PAPER Special Section on Network Virtualization and Network Softwarization for Diverse 5G Services

# Scalable State Space Search with Structural-Bottleneck Heuristics for Declarative IT System Update Automation

**Takuya KUWAHARA**[†a)], **Takayuki KURODA**[†b)], **Manabu NAKANOYA**[†c)], **Yutaka YAKUWA**[†d)], ***Nonmembers***, ***and*** **Hideyuki SHIMONISHI**[†e)], ***Member***

**SUMMARY** As IT systems, including network systems using SDN/NFV technologies, become large-scaled and complicated, the cost of system management also increases rapidly. Network operators have to maintain their workflow in constructing and consistently updating such complex systems, and thus these management tasks in generating system update plan are desired to be automated. Declarative system update with state space search is a promising approach to enable this automation, however, the current methods is not enough scalable to practical systems. In this paper, we propose a novel heuristic approach to greatly reduce computation time to solve system update procedure for practical systems. Our heuristics accounts for structural bottleneck of the system update and advance search to resolve bottlenecks of current system states. This paper includes the following contributions: (1) formal definition of a novel heuristic function specialized to system update for A* search algorithm, (2) proofs that our heuristic function is consistent, i.e., A* algorithm with our heuristics returns a correct optimal solution and can omit repeatedly expansion of nodes in search spaces, and (3) results of performance evaluation of our heuristics. We evaluate the proposed algorithm in two cases; upgrading running hypervisor and rolling update of running VMs. The results show that computation time to solve system update plan for a system with 100 VMs does not exceed several minutes, whereas the conventional algorithm is only applicable for a very small system.

***key words:*** *orchestration, task planning, system update automation, automated planning, change management, model-based engineering, declarative provisioning, network function virtualization*

## 1. Introduction

Recent development and wide acceptance of SDN [1], [2] and NFV [3], [4] technologies have made network infrastructure greatly flexible, i.e. deployment, modification, scale-in and out, as well as destruction of (virtual) network systems can be done in a software-defined way. Network functions, such as router, firewall, load-balancer, IDS/IPS, and so on, are deployed as a virtual entity on any available physical servers, rather than manually installing physical boxes into the system. Scaling-up the number of such functions, for instance, can be done by instructing orchestrators or NFV platform software, rather than purchasing any new hardware boxes. Researches on automation of NFV resource optimization can be found in many literatures, such as [5], for example. In addition, in plumbing among these functions, there's no need to physically wire among them, or set up complex VLAN/routing configurations with a lot of constraints, but SDN technology enables ideal plumbing among these functions with arbitrary logical topology, appropriate path selection, and logical separation from other services, regardless of physical network installations. Optimization of co-design of NFV function placement and SDN route optimization can also be found in many literatures, such as [6], for example.

These technologies have made operator's tasks for deployment and maintenance of the system significantly ease and agile, however, these tasks have to be well-prepared for every situations. Workflows for deployment and maintenance tasks have to be carefully designed so that the order of each step in the workflow ensures no system failure nor effects on running services. This is an another burden for operators to design such workflows in advance of the service operation, as network systems become more complex and demands for network services adapted to allow diverse customization requirement. In addition, such pre-defined workflow can only be applied for well-prepared regular tasks, such as increasing the number of functions within a prepared resource pool, or preparing a new virtual network based with common templates. Let us suppose a case, when a resource pool in some small area has been exhausted and needs to allocate extra resource in different area, there would be a manual configurations to fetch resource in remote area, set up a network path, or extend virtual network span to that area, reconfigure service chain topology or job dispatch policy, etc. Someone can imagine another case where any network hardware or service halted unexpectedly and simple switching to stand-by system is not applicable, or there's any trouble at a single point of failure, the operator is urged to prepare recovery workflow on-demand as quickly as possible. We can also imagine a workflow to patch or upgrade hypervisor, which requires delicate treatment of services running on top of that hypervisor because such services have mutual effects with other services running at other part of the system.

Based on above discussions, we are facing to new frontier of SDN/NFV research towards automation of workflow generation. This will make manual and labor jobs of operation expert fully automated and possibly autonomous self-management of a network infrastructure. In this paper, we discuss automated system update to eliminate human task

Manuscript received April 12, 2018.
Manuscript revised July 26, 2018.
Manuscript publicized September 20, 2018.

[†]The authors are with System Platform Research Laboratories, NEC, Kawasaki-shi, 211-8666 Japan.
a) E-mail: t-kuwahara@me.nec.jp.com
b) E-mail: t-kuroda@ax.nec.jp.com
c) E-mail: m-nakanoya@bc.nec.jp.com
d) E-mail: y-yakuwa@ap.nec.jp.com
e) E-mail: h-shimonishi@cd.jp.nec.com

DOI: 10.1587/transcom.2018NVP0009

of generation of system update plan. To this end, we are proposing to employ *Declarative system update* [7]–[12] for this issue. Taking this approach, system operators have only to input *the desired system state*, and planning engines automatically generate system-update plans on behalf of the system operators.

Some declarative system update approaches [7], [8] adopt a "state-space search" as an automated planning method. Although it has been suggested that a state space of a system grows exponentially with increasing number of system components [11], our previous work proposed a kind of a divide-and-conquer technique to solve planning problems concerning updating large-scale systems [8].

However, in some cases, exponential growth of the state space remains an issue concerning planning problems. Accordingly, the notion of *global constraints* was introduced to planning system updates [12]. This notion makes it possible to impose certain system requirements on the transient and desired states of a system update. For example, when system operators need to update a service that manages critical infrastructure and cannot be stopped even during system update procedures, planning engines accounting for global constraints can generate a system-update plan while keeping the system operating normally.

Unfortunately, if we introduce global constraints to our previous approach, our divide-and-conquer strategy may not sufficiently break an original planning problem down to fine grained sub-problems. In such cases, system update would not be finished in realistic time, even if the size of the system is realistic, for example, consisting of $50 \sim 100$ components.

In our preliminary report, we have proposed a basic method for efficient planning of declarative system update [13]. In this paper, we further propose complete method which can omit recomputation of values of heuristic function to be more efficient and present mathematical proof of justification for this optimization. We also present various experimental results to verify the effectiveness of the method.

The proposed method suppresses the increase in time for planning and makes declarative system update a more powerful tool for system management. The proposed planning method is based on the observation that most system updates contain *a "bottleneck" component*, which takes many steps to update due to dependencies with other components. We found that a system-update plan can be quickly formulated by setting the update of the bottleneck components as a priority. We formalized this finding as *a heuristic function* for the A* search algorithm, and we designed an efficient planning method tailored for declarative system update.

In the rest of this paper, related work is introduced in Sect. 2. In Sect. 3, declarative system update with state-space search is overviewed. In Sect. 4, state models and planning problems are formalized in the similar way to a reported method [8]. In Sect. 5, a heuristic function is formulated and validated, and an efficient algorithm for calculating that heuristic function is devised and validated. In Sect. 6, the results of an experimental evaluation of the performance of the algorithm are presented and discussed, and in Sect. 7, the conclusions of this study are presented.

## 2. Related Work

A fundamental framework for declarative system update, which accepts a desired state as input and generates workflows that execute system update by automatic planning, has been proposed by El Maghraoui et al. [11].

The planning technique used in [11] is called *partial-order planning (POP)*. In short, a POP engine receives a set of *actions*, which are elemental operations for system update (and similar to *the transitions* in our models), and constructs *a partial plan*.

In contrast to a totally ordered plan specifying a complete order of actions, a partial plan is a *partially ordered* set of actions and specifies only necessary and sufficient ordering of actions. A partial plan can be more efficiently executed than a totally ordered plan because two or more actions can be executed *in parallel* when they are not related by their relative orders.

In a comparative study on several planning techniques, the most-suitable technique for planning an IT-system update was discussed [14]. It was concluded that *hierarchical task network (HTN)* algorithms, which decompose a high-level action into several finer-grained actions and finally elementary actions by applying decomposition to the actions hierarchically, are the most suitable.

It was also pointed that a general HTN algorithm is insufficient to solve a planning problem concerning a large-scale IT system. In another work, to tailor an HTN algorithm to IT system planning [15], a hybrid approach to a HTN and state-space search and several optimization were proposed [16].

In contrast to those works, in our previous work [8]–[10], we chose state-space search for making it possible to explicitly deal with system states and easily taking *global constraints* (cf. [12]) into account. State-space search can be adapted to account for global constraints by restricting a state space to states satisfying given global constraints. This feature is difficult to integrate into methods such as those that handle states of IT systems implicitly (like POP and HTN).

As mentioned in Sect. 1, the size of state spaces grows exponentially with increasing number of components in a system. To address this issue, our previous work [8] adopted a divide-and-conquer strategy, namely, dividing systems into strongly connected components by dependencies. By our divide-and-conquer strategy, the main factor of scalability of a state-space search reduces from the size of *a whole system* to the maximum size of *strongly connected components of a system*. However, global constraints make related system components depend on each other, and our divide-and-conquer strategy does not work due to these dependencies.

## 3. Illustration of Fully Declarative System Update

This section illustrates declarative system update. Figure 1 shows our motivating example, that is a situation in which $n$

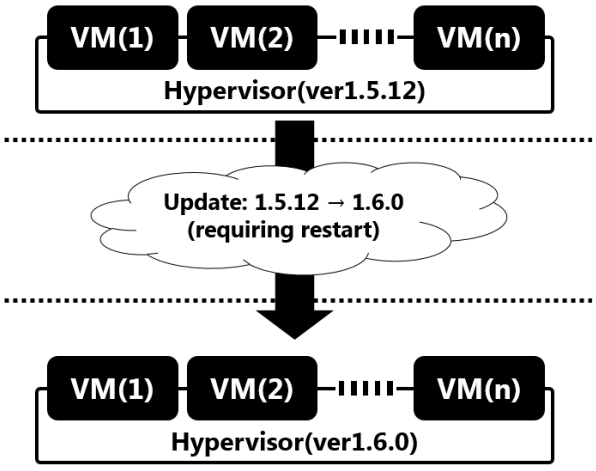


**Fig. 1** Motivating example: updating a running hypervisor.

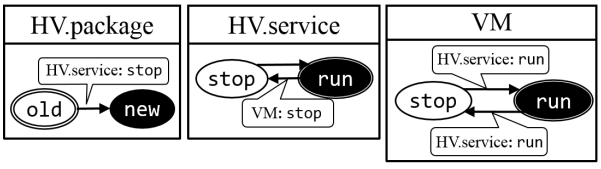


**Fig. 2** An example of state model planning.

virtual machines are hosted on a hypervisor and the hypervisor needs to be stopped and restarted for version update. In the rest of this section, for brevity, we assume that $n = 1$, that is, only one VM is hosted on the hypervisor.

In this example, it is supposed that the hypervisor cannot be upgraded while in operation and that the hypervisor cannot be stopped while it is hosting the VM. In this case, system update must be executed in correct order, namely, (1) stop the VM, (2) stop the hypervisor, (3) upgrade the hypervisor, (4) restart the hypervisor, and (5) reboot the VM.

### 3.1 Modeling Systems

In this study, each system component is modeled by *state models*. State models consist of several transition systems with dependencies on their transitions. Here, a transition system consists of *states* of a system and *transitions* between them, and represents behaviors of a system with discrete internal status.

System updates are modeled by the planning problems on state models, which is called *state model planning*. A state model planning is defined by a state model as well as initial and desired states of each transition system of the state model. Figure 2 shows an example modeled by state-model planning.

Each rectangle in the figure represents a transition system, henceforth called *a state element* to emphasize that it is *an element* of a whole state model. Each circle in a state element is a state of it, and each arrow between two states is a transition between them. Each double-lined state is *an initial state* of each state element, and each filled-with-black state is *a desired state* of each state element. The goal of system update is for all components to transit from an initial state to a desired state.

In Fig. 2, three state elements are shown. "HV.package" has two states, `old` and `new`, which represent an old version and a new version of the package of the hypervisor, respectively. "HV.service" has two states, `run` and `stop`, which represent running and stopped states of the hypervisor, respectively. "VM" also has two states, `run` and `stop`, which represent running and stopped states of the VM, respectively[†].

In Fig. 2, some transitions have a label with a name of an element and its state. For example, the label "VM:`stop`" is attached to the transition from `run` to `stop` in HV.service. These labels represent *dependencies* on transitions. A dependency on a transition represents *a requirement to use the transition*. That is, dependency "$d$:$s$" on a transition from $src$ to $dst$ of state element $e$ means that "$e$'s state can move from $src$ to $dst$ *only if* component $d$ is at state $s$."

We note that system operators do not have to directly represent a state model of their systems. Representing state model is equally or more time-consuming tasks for system operators. We assume that our declarative system update tool uses high-level models to hide details of states, transitions and dependencies. For example, the component model [9], [10] is proposed, which consists of system components, connections between them, and property of them. In this model, the model encapsulates the details of components and connections, and are converted into a detailed model that corresponds to the state model.

### 3.2 State Model Planning

The goal of state model planning is to discover the shortest sequence of transitions that transfers states of all components to desired states while keeping all requirements on labels fulfilled. A system-update plan is obtained as a solution of state model planning, and system update can be correctly performed by following the system-update plan. For example, one solution of the problem shown in Fig. 2 is given as the following plan:

1. VM: `run` → `stop`
2. HV.service: `run` → `stop`
3. HV.package: `old` → `new`
4. HV.service: `stop` → `run`
5. VM: `stop` → `run`

(Here, notation $e : a \to b$ denotes a transition from $a$ to $b$ in $e$.) This solution gives a correct system-update plan directly.

A way to formulate state-model planning as a shortest-path problem on *a state graph*, which is a transition system integrating all system components, was shown by Kuroda and Gokhale [8]. For example, Fig. 3 shows a state graph of our running example. A system-update plan can be found as

[†]Precisely, the "VM" component should be named "VM.service", but it is omitted for brevity.

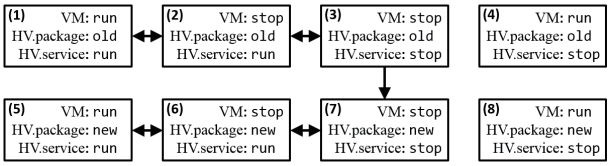


**Fig. 3** State graph of the system shown in Fig. 2.

paths from the initial state (1) to the desired state (5) over this state graph as the following path: (1) → (2) → (3) → (7) → (6) → (5).

## 4. Formal Definitions

In this section, state models and state model planning are formalized.

### 4.1 Notations

The following useful notations are introduced first.

- Let $I$ be a set. When tuple $\mathcal{X} = (\mathcal{X}_i)_{i\in I}$ is an element of $\prod_{i\in I} X_i$, $\mathcal{X}[i]$ denotes $\mathcal{X}_i$. That is, $\_[i]$ is an $i$-th projection map.
- Suppose that $I = \{i_1, \ldots, i_n\}$. Tuple $\mathcal{X}$ such as $\mathcal{X}[i] = X_i$ can also be denoted as $\{i_1 : X_{i_1}, \ldots, i_n : X_{i_n}\}$.
- Sequence of $x_1, \ldots, x_n$ arranged in this order is denoted as $\langle x_1, \ldots, x_n\rangle$.
- When $s = \langle x_1, \ldots, x_n\rangle$, length of $s$ is defined as $n$ and denoted as $\|s\|$.

### 4.2 State Model

System components are modeled as unlabeled transition systems, which called *state elements*.

**Definition 4.1** (State element)**.** *A state element $e$ is defined as a pair $(S, T)$ where $S$ is called states, and $T \subseteq S \times S$ is called transitions. $S$ can be denoted as $\mathbb{S}(e)$ and $T$ can be denoted as $\mathbb{T}(e)$.*

State elements of a state model represent behavior of components (i.e., finer-grained parts of systems), and the state model represents the behavior of the whole system that consists of these components. Thus, to formally define a state model, interactions between state elements, called *"dependencies"*, need to be defined. For this purpose, *global states and global transitions* of a set of state elements are first introduced.

**Definition 4.2** (Global state and global transition)**.** *Let $E$ be a set of state elements.*

*Global states over $E$ are defined as a set of all tuples of states over $E$. Formally, global states $\mathcal{S}_E$ are defined as $\prod_{e\in E} \mathbb{S}(e)$.*

*Global transitions over $E$ are all transitions $\bigcup_{e\in E} \mathbb{T}(e)$ extended to global states. Formally, let $e \in E$, $t \in \mathbb{T}(e)$ and $\sigma_1, \sigma_2 \in \mathcal{S}_E$. $(\sigma_1, t, \sigma_2)$ is a global transition when $t = (\sigma_1[e], \sigma_2[e])$ holds and $\sigma_1[e'] = \sigma_2[e']$ holds for each $e' \in E \setminus \{e\}$.*

*A set of all global transitions over $E$ is denoted as $\mathcal{T}_E$, and $(\sigma_1, t, \sigma_2) \in \mathcal{T}_E$ is denoted as $\underline{t}_{\sigma_1}$.*

Intuitively, global states represent *states of a whole system*, and global transitions represent *component-wise transitions of a whole system*.

The notion of *dependency* among state elements can be introduced as follows. As mentioned in Sect. 3.2, dependencies represent restrictions on transitions of state elements, such as "The VM instance can be launched only when the hypervisor is running."

**Definition 4.3** (Dependency)**.** *Let $E$ be a set of state elements.*

*A dependency over $E$ is a mapping $D$ from each transition in $E$ to a tuple of (non-empty) subsets of states in $E$:*

$$D : t \mapsto (B_e)_{e\in E} \quad (\text{where } B_e \subseteq \mathbb{S}(e) \wedge B_e \neq \emptyset)$$

*State element $e$ is said to be depending to $e'$ when $D(t)[e'] \neq \mathbb{S}(e')$ holds for some $t \in \mathbb{T}(e)$. Dependency from $e$ to $e'$ can be denoted as $e \overset{D}{\circ\cdots\blacktriangleright} e'$. When it is clear from the context, $D$ can be omitted, and $e \circ\cdots\blacktriangleright e'$ can be simply written.*

A dependency maps from each transition to a set of global states that the transition is executable. Formally, A transition $t$ is said to be *executable at a global state* $\sigma$ when $\sigma[e]$ is in $D(t)[e]$ for all $e \in E$.

The following notation is also introduced:

- A set of all dependencies over $E$ is denoted as $\mathcal{D}_E$.
- When $B[e] = \mathbb{S}(e)$ holds, "$e : \mathbb{S}(e)$" can be omitted from $\{e_1 : B[e_1], e_2 : B[e_2], \ldots, e_n : B[e_n]\}$ for simplicity of expression.
- The symbol $\top$ denotes $\{e_1 : \mathbb{S}(e_1), \ldots, e_n : \mathbb{S}(e_n)\}$.
- When $D$ is a dependency and $(s, d)$ is a transition, $D((s, d))$ can be denoted as $D(s, d)$.

Then, state models are formally defined as follows. Let $E$ be a set of state elements and $D$ be a dependency over $E$. A state model is defined as pair $\mathcal{M} = (E, D)$.

**Example 4.1.** *State elements of the system shown in Fig. 4 are formalized as follows:*

- *HV.package* = $(\{\texttt{old}, \texttt{new}\}, \{(\texttt{old}, \texttt{new})\})$
- *HV.service* = $(\{\texttt{run}_{\text{hv}}, \texttt{stop}_{\text{hv}}\},$

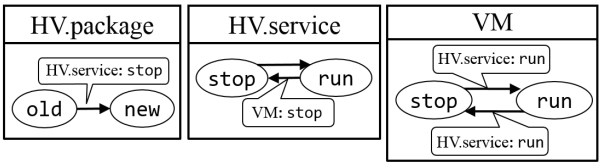


**Fig. 4** State model of the system used in Sect. 3.

$\{(\mathtt{run}_{\mathrm{hv}}, \mathtt{stop}_{\mathrm{hv}}), (\mathtt{stop}_{\mathrm{hv}}, \mathtt{run}_{\mathrm{hv}})\})$

- $VM = (\{\mathtt{run}_{\mathrm{VM}}, \mathtt{stop}_{\mathrm{VM}}\},$
  $\{(\mathtt{run}_{\mathrm{VM}}, \mathtt{stop}_{\mathrm{VM}}), (\mathtt{stop}_{\mathrm{VM}}, \mathtt{run}_{\mathrm{VM}})\})$

*Dependency $D$ between these state elements is defined as*

$$\begin{cases} D(\mathtt{old}, \mathtt{new}) = \{HV.service : \{\mathtt{stop}_{\mathrm{hv}}\}\} \\ D(\mathtt{run}_{\mathrm{hv}}, \mathtt{stop}_{\mathrm{hv}}) = \{VM : \{\mathtt{stop}_{\mathrm{VM}}\}\} \\ D(\mathtt{run}_{\mathrm{VM}}, \mathtt{stop}_{\mathrm{VM}}) = D(\mathtt{stop}_{\mathrm{VM}}, \mathtt{run}_{\mathrm{VM}}) = \\ \quad \{HV.service : \{\mathtt{run}_{\mathrm{hv}}\}\} \end{cases}$$

*; otherwise, $D(t) = \top$.*

*Finally, the state model of the system in Fig. 4 is given as $\mathcal{M} = (\{HV.package, HV.service, VM\}, D)$.*

### 4.3 State Graph

As mentioned in Sect. 3.2, a state model is translated into one transition system called a *state graph* [8].

**Definition 4.4** (State graph)**.** *Let $\mathcal{M} = (E, D)$ be a state model. A state graph $\mathcal{G}(\mathcal{M}) = (\mathcal{S}, \mathcal{T})$ is a labeled state transition system defined as*

- $\mathcal{S} = \mathcal{S}_E$
- $\mathcal{T} = \left\{ \underline{t}_{\sigma} \in \mathcal{T}_E \;\middle|\; t \text{ is executable at } \sigma. \right\}$

State graph is a graph whose nodes are states of a system, and edges are (executable) transitions between them.

A global transition $(\sigma_1, t, \sigma_2)$ in $\mathcal{T}$ can be denoted as $\sigma_1 \xrightarrow{t}_{\mathcal{T}} \sigma_2$, or simply $\sigma_1 \xrightarrow{t} \sigma_2$ when $\mathcal{T}$ is clear from the context.

**Example 4.2.** *Let $\mathcal{M}$ be a state model defined in the example 4.1. Figure 5 shows the state graph $\mathcal{G}(\mathcal{M})$, which consists of six global states $\sigma_0, \ldots, \sigma_5$. Here, "hv.p" (resp. "hv.s") abbreviates "HV.package" (resp. "HV.service"). In Fig. 5, two "unreachable" global states: $(hv.p : \mathtt{new}, hv.s : \mathtt{stop}_{\mathrm{hv}}, VM : \mathtt{run}_{\mathrm{VM}})$ and $(hv.p : \mathtt{old}, hv.s : \mathtt{stop}_{\mathrm{hv}}, VM : \mathtt{run}_{\mathrm{VM}})$ are omitted.*

### 4.4 State Model Planning

*State model planning* and its *solution* can now be defined. Formally, *state model planning* is defined as a planning problem defined for a state graph of a state model from *a global initial state* to *a global desired state*.

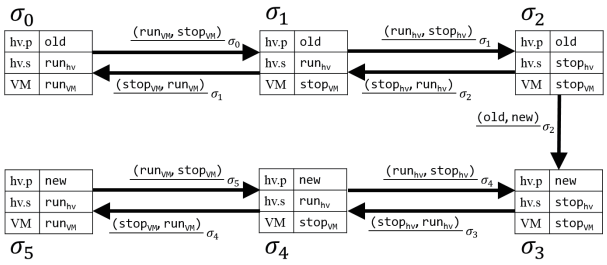


**Fig. 5** State graph of the state model shown in Fig. 4.

**Definition 4.5** (State model planning)**.** *State model planning is a triple $\mathcal{P} = (\mathcal{M}, \sigma_0, \sigma_f)$.*

- *$\mathcal{M}$: state model*
- *$\sigma_0 \in \mathcal{G}(\mathcal{M})$: global initial state*
- *$\sigma_f \in \mathcal{G}(\mathcal{M})$: global desired state*

*Paths* on state models are defined as:

**Definition 4.6** (Path on state models)**.** *Let $\mathcal{M} = (E, D)$ be a state model, and $\mathcal{G}(\mathcal{M}) = (\mathcal{S}, \mathcal{T})$ be the state graph of $\mathcal{M}$. If $\sigma_0, \sigma_1, \ldots, \sigma_n \in \mathcal{S}$ and $\underline{t_1}, \underline{t_2}, \ldots, \underline{t_n} \in \mathcal{T}$ satisfy the following relations:*

$$\sigma_0 \xrightarrow{t_1}_{\mathcal{T}} \sigma_1, \sigma_1 \xrightarrow{t_2}_{\mathcal{T}} \sigma_2, \ldots, \sigma_{n-1} \xrightarrow{t_n}_{\mathcal{T}} \sigma_n$$

*sequence $\langle t_1, t_2, \ldots, t_n \rangle$ is called a path on $\mathcal{M}$ from $\sigma_0$ to $\sigma_n$.*

*A solution* to state-model planning $(\mathcal{M}, \sigma_0, \sigma_f)$ is defined as one of the shortest paths on $\mathcal{M}$ from $\sigma_0$ to $\sigma_f$. Hence, $L(\mathcal{P})$ denotes:

- the length of a solution to state model planning $\mathcal{P}$ when $\mathcal{P}$ has a solution, or
- $\infty$ when $\mathcal{P}$ has no solution.

By definition of a solution to state-model planning, a solution to state model planning $(\mathcal{M}, \sigma_0, \sigma_f)$ can be obtained by solving *shortest path search* problem defined for a graph $\mathcal{G}(\mathcal{M})$ from $\sigma_0$ to $\sigma_f$. A heuristic search algorithm, a.k.a. the A* algorithm [17], is adopted to solve a shortest-path search problem defined for a state graph with the proposed heuristic function.

## 5. Heuristic Function for State Model Planning

### 5.1 Challenging Case in Realistic Use

For brevity of examples, we only discussed about a simple and small problem consisting a VM and a hypervisor so far. However, in many cases, hypervisors manage two or more VMs. It is supported that $n$ is taken as a natural number more than two, and $n$ VMs are hosted on the hypervisor. A state model of a system containing $n$ VMs has $2^{n+1} + 2$ global states. Unfortunately, in "hop count" measurement, the desired state of the running example is situated at the farthest point from the initial state of the running example.

Therefore, if Dijkstra's algorithm [18] is used to find a path from an initial state to a desired state, then all over the global states are traversed and the problem will never be solved in a practical time when $n$ is large (e.g., when $n > 100$). Accordingly, to deal with large systems, a heuristic function tailored for state model planning must be adopted.

### 5.2 Critical Element Heuristic Function

The proposed heuristic function is based on the following observation. In many cases of system update, a "bottleneck" component occurs in the entire system update. That is, a component has many dependencies to other components directly or indirectly, and resolving its dependencies takes most of the updating time. The element of that bottleneck component is called *the critical element*. For example, in the system shown in Fig. 2, HV.package is said to be the critical element of the planning problem, because transferring states of all other elements are caused by the update of the HV.package.

The proposed heuristic function can be regarded as one of variants of relaxed planning heuristics [19]. That is, dependencies of a given state model planning are relaxed by extracting dependencies that only relate to the critical element and removing all other dependencies. The length of a solution of the *relaxed* state model planning under approximates that of the original problem. Thus, this *approximate length of a solution* can be used as an admissible heuristics of the A* search algorithm.

The critical element of given state model planning must be identified. The proposed algorithm searches for the critical element by using a *brute-force* algorithm. Namely, for all elements $e$, given state model planning is relaxed by $e$, an approximate length of a solution is computed by solving each relaxed state model planning, and the element that produces maximum approximate length is taken as the critical element.

### 5.3 Definition of Relaxation of State Model Planning and Critical Element Heuristic Function

First, *e-derived dependency* is introduced. Intuitively, $e$-derived dependency based on dependency $D$ is dependency $D$ restricted on a rooted tree whose root is $e$, nodes are $E$, and edges are a part of "$\circ\cdots\blacktriangleright$" relations.

**Definition 5.1** ($e$-derived dependency)**.** *Let $\mathcal{M} = (E, D)$ be a state model. A dependency $D^e_\downarrow$ is called an $e$-derived dependency based on $D$ when the following conditions hold (1) - (4) for all $e \in E$:*

*(1) For each transition $t$ and each state $e'$, $D(t)[e'] \subseteq D^e_\downarrow(t)[e']$ holds.*
*(2) $e$ is not depended from any other $e \in E$.*
*(3) Dependency $D^e_\downarrow$ does not form a cycle.*
*(4) $\forall e_1, e_2, e_3 \in E \setminus \{e\}$.*

$$\left[ e_2 \underset{D^e_\downarrow}{\circ\cdots\blacktriangleright} e_1 \wedge e_3 \underset{D^e_\downarrow}{\circ\cdots\blacktriangleright} e_1 \right] \Rightarrow e_2 = e_3$$

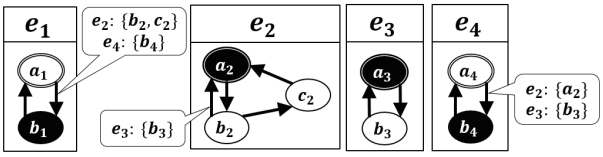


**Fig. 6** State model planning $\mathcal{P}_\varepsilon$.

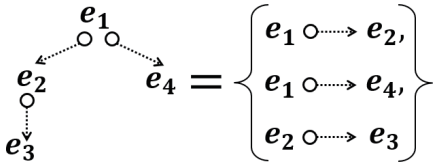


**Fig. 7** $e_1$-rooted tree of dependencies over $\mathcal{P}_\varepsilon$.

**Example 5.1.** *Problem $\mathcal{P}_\varepsilon$ shown in Fig. 6 is used in Sect. 5. Dependency $D_\varepsilon$ of $\mathcal{P}_\varepsilon$ is given as:*

$$\begin{cases} D_\varepsilon(a_1, b_1) & = \{e_2 : \{b_2, c_2\}, e_4 : \{b_4\}\} \\ D_\varepsilon(b_2, a_2) & = \{e_3 : \{b_3\}\} \\ D_\varepsilon(a_4, b_4) & = \{e_2 : \{a_2\}, e_3 : \{b_3\}\} \\ D_\varepsilon(\cdot, \cdot) & = \top \quad (otherwise) \end{cases}$$

*The following two $e_1$-derived dependencies are introduced as*

*1) If all dependencies except for direct dependencies are removed from $e_1$, the following $e_1$-derived dependencies $D_\varepsilon{}^{e_1}_{\downarrow direct}$ are obtained:*

$$\begin{cases} D_\varepsilon{}^{e_1}_{\downarrow direct}(a_1, b_1) & = \{e_2 : \{b_2, c_2\}, e_4 : \{b_4\}\} \\ D_\varepsilon{}^{e_1}_{\downarrow direct}(\cdot, \cdot) & = \top \quad (otherwise) \end{cases}$$

*2) If all dependencies except for those along with an $e_1$-rooted tree of dependencies shown in Fig. 7 are removed, the following $e_1$-derived dependencies $D_\varepsilon{}^{e_1}_{\downarrow derived}$ are obtained:*

$$\begin{cases} D_\varepsilon{}^{e_1}_{\downarrow derived}(a_1, b_1) & = \{e_2 : \{b_2, c_2\}, e_4 : \{b_4\}\} \\ D_\varepsilon{}^{e_1}_{\downarrow derived}(b_2, a_2) & = \{e_3 : \{b_3\}\} \\ D_\varepsilon{}^{e_1}_{\downarrow derived}(\cdot, \cdot) & = \top \quad (otherwise) \end{cases}$$

*relaxed state model* and *relaxed state-model planning*can thus be defined on the basis of an $e$-derived dependency.

**Definition 5.2** (Relaxed state model $\mathcal{M}|_{D^e_\downarrow}$ and relaxed state-model planning $\mathcal{P}|_{D^e_\downarrow}$)**.** *Let $\mathcal{M} = (E, D)$ be a state model, $e$ be in $E$, $D^e_\downarrow$ be an $e$-derived dependency, $\mathcal{P} = (\mathcal{M}, \sigma_0, \sigma_f)$ be a state model planning.*

*$\mathcal{M}|_{D^e_\downarrow} = (E, D^e_\downarrow)$ is called an $D^e_\downarrow$-relaxed state model of $\mathcal{M}$ and $\mathcal{P}|_{D^e_\downarrow} = (\mathcal{M}|_{D^e_\downarrow}, \sigma_0, \sigma_f)$ an $D^e_\downarrow$-relaxed state model planning of $\mathcal{P}$.*

Problem $\mathcal{P}|_{D^e_\downarrow}$ is said to be a *relaxed* problem of $\mathcal{P}$ in the sense that a graph $\mathcal{G}(\mathcal{M})$ is a subgraph of $\mathcal{G}(\mathcal{M}|_{D^e_\downarrow})$.

The problem $\mathcal{P}|_{D^e_\downarrow}$ takes only dependencies derived from $e$ into account. Accordingly, a critical element and

*critical element heuristics* can be formally defined as follows.

The following notation is introduced for brevity. Let $\mathcal{P} = (\mathcal{M}, \sigma_0, \sigma_f)$ be state model planning and $\sigma$ be a global state of $\mathcal{M}$. $\mathcal{P}@\sigma$ is written for state-model planning $(\mathcal{M}, \sigma, \sigma_f)$.

**Definition 5.3.** *Critical-element heuristic function* $\vartheta_{\mathcal{P}}(\sigma) : \mathcal{S} \to \mathbb{N}$ *is defined as*

$$\vartheta_{\mathcal{P}}(\sigma) = \max_{e \in E} L(\mathcal{P}|_{D^e_{\downarrow}}@\sigma)$$

*Element* $\arg\max_{e \in E} L(\mathcal{P}|_{D^e_{\downarrow}})$ *is called a critical element of* $\mathcal{P}$. *(Subscript* $\mathcal{P}$ *is omitted from* $\vartheta_{\mathcal{P}}$ *and* $\vartheta$ *is simply written when it is clear from context.)*

The following theorem shows that the function $\vartheta$ is *consistent*. Consistency is important property for heuristic functions in the following two points; (1) a consistent function is also an *admissible* function, which guarantees A* algorithm calculates a collect shortest path, and (2) values of a consistent function are invariant through A* algorithm. We can optimize A* algorithm using this property.

**Theorem 5.1.** *Let* $\mathcal{P}$ *be state-model planning. Function* $\vartheta_{\mathcal{P}}$ *is consistent, namely,*

$$\vartheta_{\mathcal{P}}(\sigma) \leq \vartheta_{\mathcal{P}}(\sigma') + 1$$

*holds for all global states* $\sigma, \sigma'$ *such as* $\sigma \to \sigma'$.

*Proof.* Let $e_c, e'_c$ be critical elements of $\mathcal{P}@\sigma, \mathcal{P}@\sigma'$ respectively. When no path from $\sigma'$ to $\sigma_f$ exists on $\mathcal{P}|_{D^{e_c}_{\downarrow}}$, $\vartheta_{\mathcal{P}}(\sigma') = \infty$ and Theorem 5.1 holds. Therefore, the case that a path from $\sigma'$ to $\sigma_f$ exists on $\mathcal{P}|_{D^{e_c}_{\downarrow}}$ is considered.

The shortest path $\pi$ on $\mathcal{P}|_{D^{e_c}_{\downarrow}}$ from $\sigma'$ to $\sigma_f$ is chosen. Because transition $t$ such that $t_{\_\sigma} = (\sigma, t, \sigma')$ exists, path $\pi'$ exists on $\mathcal{P}|_{D^{e_c}_{\downarrow}}$ such that $t, \underbrace{t_0, t_1, \ldots, t_n}_{\pi}$. Because $\vartheta(\sigma)$ is the length of the shortest path from $\sigma$ to $\sigma_f$ on $\mathcal{P}|_{D^{e_c}_{\downarrow}}$, the following equation is obtained:

$$\vartheta(\sigma) \leq |\pi'| = |\pi| + 1 \tag{1}$$

Because $e'_c$ is a critical element of $\mathcal{P}|_{D^e_{\downarrow}}@\sigma'$, the following holds:

$$L(\mathcal{P}|_{D^{e_c}_{\downarrow}}@\sigma') \leq L(\mathcal{P}|_{D^{e'_c}_{\downarrow}}@\sigma') \tag{2}$$

$$|\pi| \leq \vartheta(\sigma') \tag{3}$$

By (1) and (3), $\vartheta(\sigma) \leq \vartheta(\sigma') + 1$ holds. □

The *consistency* property makes the proposed algorithm more efficient by eliminating recalculation of the value of the proposed heuristic function. The detail of optimization of A* search algorithm is discussed below.

## 5.4 Algorithm for Computing $\vartheta(\sigma)$

An algorithm for computing $L(\mathcal{P}|_{D^e_{\downarrow}}@\sigma)$ for calculating $\vartheta(\sigma)$ is proposed in the following. We fix $e$-deriving dependencies for each $e$ in the rest of this section, and for brevity, $\mathcal{P}|_{D^e_{\downarrow}}$ is abbreviated as $\mathcal{P}|_e$.

We note that the number of states of each state element is assumed to be enough small (e.g., 2~5) to finish the preprocess explained in the following subsection in negligibly short time to whole planning process.

### 5.4.1 Preprocess

First, the proposed algorithm requires *simple e-local paths of each element e*. Here, *an e-local path* is a sequence of transition: $(s_0, d_0), (s_1, d_1), \ldots, (s_k, t_k) \in \mathbb{T}(e)$ such that $d_{i-1} = s_i$ holds for all $i$, and a local path is said to be *simple* when it visits each state at most once. Second, the algorithm requires *the length of the shortest e-local paths of each element e*. Note that since shortest local paths are simple, these values can easily be obtained after all simple $e$-local paths of each element $e$ are obtained.

Before A* search algorithm is executed, the following preprocesses are executed. *All* simple $e$-local paths of each state pair $(s, s')$ in each element $e$ are calculated, and stored in an array $sp_e$. Then the shortest path of each state pair $(s, s')$ is obtained by choosing the shortest path from $sp_e[s, s']$, and its length in an array $dist_e$.

**Example 5.2.** *Let* $\mathcal{P}_{\mathcal{E}}|_{e_1}$ *be* $\mathcal{P}_{\mathcal{E}}|_{D_{\mathcal{E}}{}^{e_1}_{\downarrow derived}}$. *After the preprocess is applied to* $\mathcal{P}|_e$, *the following prerequisite data is obtained. (*$sp_e[s, s] = \{\langle\rangle\}$ *and* $dist_e[s, s] = 0$ *are omitted for all e.)*

- $e_i$ $(i = 1, 3, 4)$*:*

$$sp_{e_i}[s, s'] = \{\langle (s, s') \rangle\}, dist_{e_i}[s, s'] = 1 \quad (\textit{for all } s, s' \textit{ s.t. } s \neq s')$$

- $e_2$*:*

$$\begin{cases} sp_{e_2}[b_2, a_2] = \{\langle (b_2, a_2) \rangle, \langle (b_2, c_2), (c_2, a_2) \rangle\}, \\ sp_{e_2}[a_2, c_2] = \{\langle (a_2, b_2), (b_2, c_2) \rangle\}, \\ sp_{e_2}[c_2, b_2] = \{\langle (c_2, a_2), (a_2, b_2) \rangle\}, \\ sp_{e_i}[s, s'] = \{\langle (s, s') \rangle\} \quad (\textit{for all other } s, s' \textit{ s.t. } s \neq s') \end{cases}$$

$$\begin{cases} dist_{e_2}[a_2, c_2] = dist_{e_2}[c_2, b_2] = 2 \\ dist_{e_2}[s, s'] = 1 \quad (\textit{for all other } s, s' \textit{ s.t. } s \neq s') \end{cases}$$

### 5.4.2 Main Algorithm

The value of $\vartheta_{\mathcal{P}}(\sigma)$ is the maximum value of $L(\mathcal{P}|_{e_p}@\sigma)$ for all states in $\mathcal{P}$. In the proposed algorithm for calculating the value of $\vartheta_{\mathcal{P}}(\sigma)$, all of the values of $L(\mathcal{P}|_{e_p}@\sigma)$ are calculated by the following algorithm **calc**$(\mathcal{P}, e, \sigma)$:

Function **costOn**$(\mathcal{P}, e, \mathcal{R})$ calculates the minimum cost of transferring states of $e$ and its dependent elements from initial states to desired states *under a requirement* $\mathcal{R}$ *on* $e$. Here, a requirement on an element $e$ is a sequence of a set of states of $e$. Under requirement $\langle R_1, \ldots, R_m \rangle$, $e$ needs to visit one required state in each $R_i$ in the order of $R_1, \ldots, R_m$.

**Algorithm 1** "**calc**($\mathcal{P}$, $e_p$, $\sigma$)": Calculate $L(\mathcal{P}|_{e_p}@\sigma)$

```
1: for e ∈ E do
2:     visit[e] ← false
3: end for
4: L ← costOn(P|_{e_p}, e, σ, ⟨⟩)
5: for e ∈ E do
6:     if not visit[e] then
7:         L ← L + dist_e(σ[e], σ_f[e])
8:     end if
9: end for
10: return L
```

**Algorithm 2** "**costOn**($\mathcal{P}$, $e$, $\sigma$, $\langle R_0, R_1, \ldots, R_m\rangle$)"

```
1: visit[e] ← true
2: spc ← ∞
3: for ⟨s_0, s_1, ..., s_m⟩ ∈ trace(⟨R_0, ..., R_m⟩) do
4:     for π ∈ pathThrough(σ_0[e], s_0, s_1, ..., s_m, σ_f[e]) do
5:         spc' ← ||π||
6:         for dep ∈ {e' | e ○···▸ e'} do
7:             req ← pathReqs(P, dep, π)
8:             spc' ← spc' + costOn(P, dep, σ, req)
9:         end for
10:        if spc > spc' then spc ← spc'
11:    end for
12: end for
13: return spc
```

Algorithm **costOn** is defined as **Algorithm 2**.

Function **trace**($\langle R_0, \ldots, R_m\rangle$) returns a set of sequences of states and be formally defined as:

$$\textbf{trace}(\langle\rangle) = \{\langle\rangle\}, \textbf{trace}(\langle R_0, \ldots, R_m\rangle) = R_0 \times \cdots \times R_m$$

Function **pathThrough**($s_0, s_1, \ldots, s_k$) returns $e$-local paths from $s_0$ to $s_k$ visiting $s_0, s_1, \ldots, s_k$ and be formally defined as:

$$\begin{aligned}&\textbf{pathThrough}(s_0, s_1, \ldots, s_k)\\ &= \{\text{concatenation of } \pi_1, \ldots, \pi_k\\ &\qquad\qquad \mid \pi_1 \in sp_e[s_0, s_1], \ldots, \pi_k \in sp_e[s_{k-1}, s_k]\}\end{aligned}$$

Function **pathReqs**($\mathcal{P}$, $e$,$\pi$) returns a requirement that is a sequence consisting of dependencies of $\pi$ and be formally defined as:

$$\begin{aligned}&\textbf{pathReqs}(\mathcal{P}, e, \pi) = \langle D(t'_1)[e], \ldots, D(t'_l)[e]\rangle\\ &\left(\text{where } \langle t'_1, \ldots, t'_l\rangle\text{: a sequence } \pi \text{ with all } \top \text{ removed}\right)\end{aligned}$$

Finally, the value $\vartheta_{\mathcal{P}_\mathcal{E}}(\sigma)$ is obtained as the maximum value of **calc**($\mathcal{P}_\mathcal{E}$, $e$, $\sigma$) for all $e$.

**Example 5.3.** *Let* $\mathcal{P}_\mathcal{E}|_{e_1} = (\mathcal{M}, \sigma_0, \sigma_f)$ *be* $\mathcal{P}_\mathcal{E}|_{D_\mathcal{E}{}^{e_1}\downarrow_{derived}}$. *First, an execution process of* ***costOn***$(\mathcal{P}_\mathcal{E}|_{e_1}, e_1, \sigma_0, \langle\rangle)$ *is shown below. In the following explanation, for simplicity, we omit common arguments* $\mathcal{P}_\mathcal{E}|_{e_1}$ *and* $\sigma_0$ *in* ***costOn***$(\mathcal{P}_\mathcal{E}|_{e_1}, e, \sigma_0, \pi)$ *and simply write* ***costOn***$(e, \pi)$.

*First,* ***costOn*** *calls two recursive calls of itself:*

$$\begin{aligned}&\textbf{\textit{costOn}}(e_1, \langle\rangle)\\ &= \textbf{\textit{costOn}}(e_4, \langle\{b_4\}\rangle) + \textbf{\textit{costOn}}(e_2, \langle\{b_2, c_2\}\rangle) + \|\langle(a_1, b_1)\rangle\|\end{aligned}$$

*The value of* ***costOn***$(e_4, \langle\{b_4\}\rangle)$ *is calculated immediately:* ***costOn***$(e_4, \langle\{b_4\}\rangle) = \|\langle(a_4, b_4), (b_4, a_4)\rangle\| = 2$.

*In the calculation of the value of* ***costOn***$(e_2, \langle\{b_2, c_2\}\rangle)$, *The cost of paths* $\langle(a_1, b_1), (b_1, c_1), (c_1, a_1)\rangle$ *and* $\langle(a_1, b_1), (b_1, a_1)\rangle$ *are compared in line 10, and returns the smaller as the value of* ***costOn***$(e_2, \langle\{b_2, c_2\}\rangle)$.

$$\begin{aligned}&\textbf{\textit{costOn}}(e_2, \langle\{b_2, c_2\}\rangle)\\ &= \min\{\|\langle(a_1, b_1), (b_1, c_1), (c_1, a_1)\rangle\|,\\ &\qquad \textbf{\textit{costOn}}(e_3, \langle\{b_3\}\rangle) + \|\langle(a_1, b_1), (b_1, a_1)\rangle\|\}\\ &= \min\{3, \|\langle(a_3, b_3), (b_3, a_3)\rangle\| + 2\} = \min\{3, 2 + 2\} = 3\end{aligned}$$

*Finally, the value of* ***costOn***$(e_1, \langle\rangle)$ *is calculated as:*

$$\begin{aligned}&\textbf{\textit{costOn}}(e_1, \langle\rangle)\\ &= \textbf{\textit{costOn}}(e_4, \langle\{b_4\}\rangle) + \textbf{\textit{costOn}}(e_2, \langle\{b_2, c_2\}\rangle) + \|\langle(a_1, b_1)\rangle\|\\ &= 2 + 3 + 1 = 6\end{aligned}$$

*In the above execution process, each visit*$[e]$ *is set to* ***true*** *for all* $e$. *Thus,* ***calc***$(\mathcal{P}_\mathcal{E}, e_1, \sigma_0)$ *returns* 6, *which is the return value of* ***costOn***$(\mathcal{P}_\mathcal{E}|_{e_1}, e_1, \sigma_0, \langle\rangle)$.

### 5.5 Optimize A* Search with Critical Element Heuristics

The A* search algorithm was proposed by Hart et al. [17] and is described as **Algorithm 3**. Here, the detail of backtracking procedure for picking up the found path is omitted for brevity.

**Algorithm 3** "**A***(G = $(S, T)$, $s_0$, $s_f$, $h$)": Search the shortest path on the graph $G = (S, T)$ from $s_0$ to $s_f$ with the heuristics $h$

```
1: cost[s_0] ← h(s_0); Open.add(s_0)
2: while Open ≠ ∅ do
3:     s_min ← arg min_{s∈Open} f(s)
4:     if s_min = s_f then return a shortest path obtained by backtracking.
5:     Open.remove(s_min); Closed.add(s_min)
6:     for n: all neighbor nodes of s do
7:         f(n) ← cost[s_min] − h(s_min) + h(n) + 1
           if (n ∉ Open ∧ n ∉ Closed)
8:             ∨(n ∈ Open ∧ f(n) < cost[n])
               ∨(n ∈ Closed ∧ f(n) < cost[n]) then
9:             cost[n] ← f(n); Open.add(n)
10:        end if
11:    end for
12: end while
```

Additionally, we can adopt the following optimization techniques of A* search algorithm and $\vartheta$:

1) According to Theorem 5.1, the value $f(s)$ in **Algorithm 3** is not updated. Therefore, the calculation in lines 7 – 9 of **Algorithm 3** can be omitted when $n \in$ Closed.

2) A *tie breaking* technique can be adopted. That is, $s_{\min}$ can be freely chosen from nodes whose cost is minimum in all nodes in Open in line 3 of **Algorithm 3**. The proposed algorithm choose the farthest node to $s_0$.

3) When $\forall t, e.D^{e_1}_{\downarrow}(t)[e] \subseteq D^{e_2}_{\downarrow}(t)[e]$ holds, we can remove $e_1$ from the candidates of critical elements. Additionally, when $\sigma[e_1] = \sigma_f[e_1]$ and $\forall t \notin \mathbb{T}(e_1).D^{e_1}_{\downarrow}(t)[e] \subseteq D^{e_2}_{\downarrow}(t)[e]$ hold for all $e$, we have $L(\mathcal{P}|_{D^{e_1}_{\downarrow}}@\sigma) \leq L(\mathcal{P}|_{D^{e_2}_{\downarrow}}@\sigma)$ and have only to calculate the value of $L(\mathcal{P}|_{D^{e_2}_{\downarrow}}@\sigma)$.

4) A *memoization* technique can be adopted. Because **pathReqs** is often called many times with the same arguments, the result of each **pathReqs** call is stored and reused.

5) Let $\mathcal{P} = (\mathcal{M}, \sigma_0, \sigma_f)$ be a state model planning. When we regard a state model $\mathcal{M}$ as a graph with nodes of state elements and edges of dependencies, the graph of $\mathcal{M}$ is sometimes divided into some connected components $C_1, C_2, \ldots, C_k$ and we can construct an another state model planning $\mathcal{P}_i$ by restricting $\mathcal{P}$ over state elements of each $C_i$. Then, we can use more (or equal) accurate heuristics than $\vartheta$. That is, we adopt sum of values of $\vartheta_{\mathcal{P}_i}(\sigma)$ for each $\mathcal{P}_i$ instead of the value of $\vartheta_{\mathcal{P}}(\sigma)$.

Finally, the optimized **Algorithm 3** provides a system-update plan, $\mathcal{P} = (\mathcal{M}, \sigma_0, \sigma_f)$, as the return value of $\mathbf{A^*}(\mathcal{M}, \sigma_0, \sigma_f, \vartheta_{\mathcal{P}})$.

## 6. Performance Evaluation

We applied the proposed algorithm to solve system update plan of SDN/NFV systems. We start with very basic examples to safely upgrade running hypervisor without risking VMs, or virtual network functions, running on top of the hypervisor. And we also tested the algorithm for a system that needs rolling update to avoid fatal incidents and zero downtime update for high availability.

We implemented our heuristic function $\vartheta$ with $D^{e}_{\downarrow\text{derived}}$ in Scala and evaluated performance of our planning algorithm that uses critical element heuristics. Experiments were conducted on a machine with Intel(R) Xeon(R) E5-1620 0 (3.60GHz, 32GB memory) with timeout of 1 hour. We use scala runtime options "-J-Xms16G -J-Xmx16G" in performing all experiments.

### 6.1 Upgrade of Running Hypervisor

As we mentioned in the prior sections, the problem shown in Fig. 1, which is to upgrade a hypervisor, is a difficult problem for declarative system update because this induces search in state space of exponential size. Our first experiment is performed on the problem shown in Fig. 1, which includes a hypervisor and $n$ VMs.

Figure 8 depicts the state model planning `HV-VM`($n$), which is a state model representing the system of Fig. 1. The state model planning `HV-VM`($n$) is similar to our motivating example shown in Fig. 2 but has $n(> 1)$ VMs. The current state of `HV-VM`($n$) is that the hypervisor of the old version package is hosting $n$ VMs and the desired state is that the hypervisor is hosting $n$ VMs similarly but the package of the hypervisor is updated.

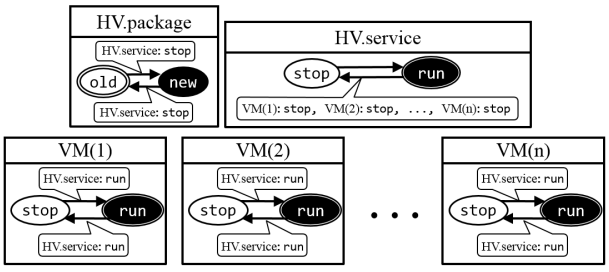


**Fig. 8** `HV-VM`($n$): planning problem of updating a hypervisor.

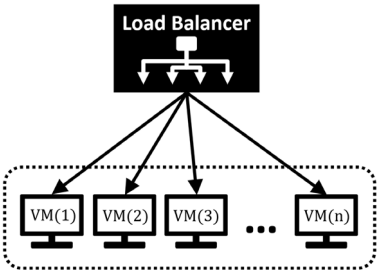


**Fig. 9** A load balanced cluster.

### 6.2 Rolling Update

As we mentioned in Sect. 1, in system updates, operators are often urged to design workflows so that the system update raise no fatal incidents of running services. If a system consists of two more VMs installed the same function, operators can adopt *rolling update strategy* to update these VMs with no downtime. Namely, in rolling update strategy, operators will not update all VMs simultaneously, but only *subset* of VMs at a time.

Figure 9 shows a load balanced cluster, which is an example of system that we can update each VMs by rolling update. $VM(1)$, $VM(2)$, ..., and $VM(n)$ are application servers where the same application is installed. The load balancer distributes attached VMs and, in most cases, all VMs are necessarily needed to keep a service in operation. Therefore, if the enough number of VMs is in operation, the service on Fig. 9 keeps alive even in system update.

Figure 10 shows the state model planning `Rolling`($n$), which represents system update in Fig. 9 with $n$ VMs. The load balancer is not appeared in `Rolling`($n$) because the state of the load balancer doesn't affect the update and apparently keeps in the "running" state. Here, we note that Fig. 10 doesn't take the condition for keeping service running into account. In order to obtain rolling update plans, we need to add a kind of global condition to the planning problem `Rolling`($n$) and solve the problem with the additioncal condition. We add the condition that "at least $m(> 0)$ VMs are running" to the problem `Rolling`($n$). Our planner can easily deal with such an additional conditions by excluding global states violating the condition.

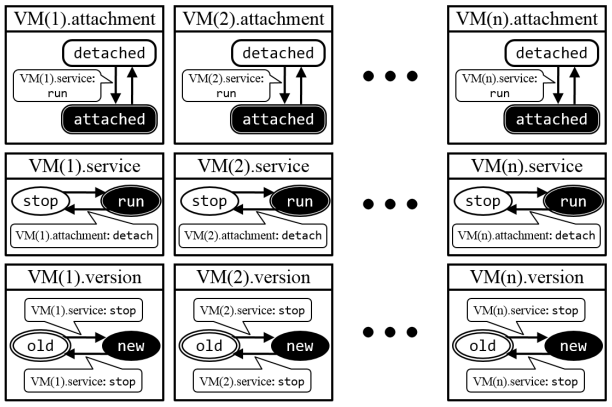


**Fig. 10** `Rolling`($n$): planning problem of rolling update.

## 6.3 Zero Downtime Updates of ToR Switches in a Data Center

We consider an another example such that uses standby components for update. As we mentioned in Sect. 1, declarative system update tools uses extra resources and standby components when system update cannot be completed in main systems. Let us suppose a case of duplicated ToR switches in a data center. Important network appliances, including ToR switches, are often duplicated for high availability. Even when main switches fall down, standby switches activate as a substitute for the main switch and keep servers mounted on the rack accessible. Additionally, even in firmware update of ToR servers with requireing system reboot, the update can be completed with no downtime by detouring traffic to standby switches.

Figure 11 shows the latter case. There are $n$ racks with eight servers and two ToR switches. In each rack, one ToR switches is an active switch, and other is a standby switch. Now, the administrator plans firmware update of active switches with no downtime.

Figure 12 shows a part of state model planning of Fig. 11, which represents just one pair of active-standby switch and contains state elements that show a firmware of active switch (SW[$i$].main.firmware), services of active and standby switch (SW[$i$].main and SW[$i$].sub), routing (SW[$i$].routeBy), connection to core switch (SW[$i$].connection), and connection to each server (SW[$i$, 1].connection, $\cdots$, SW[$i$, 8].connection). The state model planning `UpdToR`($n$) is defined by $n$ switch pairs SW[1], SW[2], ..., SW[$n$]. Similarly to `Rolling`($n$), we need to add global condition that "all switches keep connection among connecting components". That is, all state elements of ".connection" keep in the state "`on`".

## 6.4 Results

The line graph Fig. 18 shows the results of experiments to solve `HV-VM`($n$) by Dijkstra algorithm and our method. Fig. 19 shows the results of experiments using `Rolling`($n$)

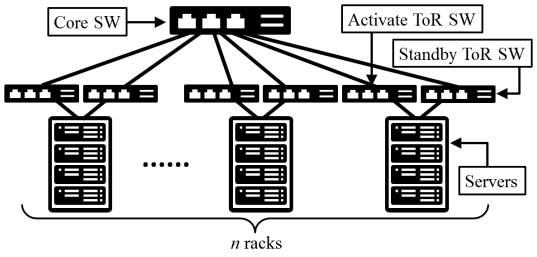


**Fig. 11** A network system on a data center.

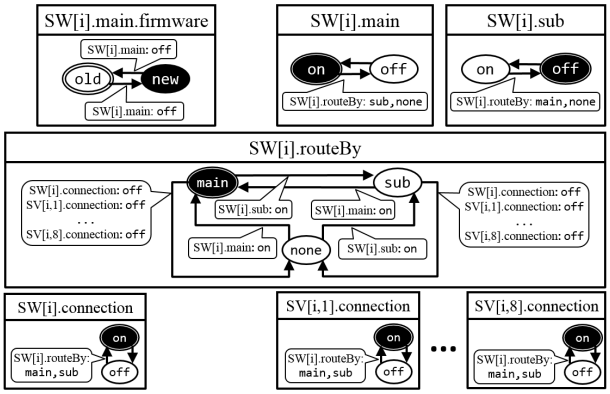


**Fig. 12** A state model planning of active/standby switches of `UpdToR`($n$).

Plan

1. VM(1): `run` → `stop`
2. VM(2): `run` → `stop`
3. VM(3): `run` → `stop`
4. HV.service: `run` → `stop`
5. HV.package: `old` → `new`
6. HV.service: `stop` → `run`
7. VM(1): `stop` → `run`
8. VM(2): `stop` → `run`
9. VM(3): `stop` → `run`

Workflow

1. Stop VM(1)
2. Stop VM(2)
3. Stop VM(3)
4. Stop hypervisor
5. Upgrade hypervisor
6. Restart hypervisor
7. Restart VM(1)
8. Restart VM(2)
9. Restart VM(3)

**Fig. 13** A plan for `HV-VM`(3) and a workflow for Fig. 1 of $n = 3$.

with the condition that "at least 1 VMs are running". The horizontal line shows $n$ and the vertical one shows elapsed time in seconds to find a solution of each problem. The line of "no heuristic" shows results of Dijkstra algorithm and the line of "our method" shows results of A* algorithm with critical-element heuristic function.

The line graph Fig. 18 shows that the computing time of state space search using Dijkstra algorithm drastically increases and exceeds an hour with $n = 24$ on one hand. On the other hand, our method can calculate a plan even with $n = 100$ in 5 minutes.

Figure 13 shows a plan for `HV-VM`(3) and the corresponding workflow. The plan correctly avoid accidental system stop by stopping the underlying hypervisor.

The results of experiments using `Rolling`($n$) shows us striking difference of efficiency between two algorithm. The line graph Fig. 19 shows the computing time of state

| Plan | Workflow |
|---|---|
| 1. VM(1).attachment: `detached` → `attached` | 1. Detach VM(1) from load balancer |
| 2. VM(1).service: `run` → `stop` | 2. Stop VM(1) |
| 3. VM(1).version: `old` → `new` | 3. Update VM(1) |
| 4. VM(1).service: `stop` → `run` | 4. Restart VM(1) |
| 5. VM(1).attachment: `attached` → `detached` | 5. Attach VM(1) to load balancer |
| 6. VM(2).attachment: `detached` → `attached` | 6. Detach VM(2) from load balancer |
| 7. VM(2).service: `run` → `stop` | 7. Stop VM(2) |
| 8. VM(2).version: `old` → `new` | 8. Update VM(2) |
| 9. VM(2).service: `stop` → `run` | 9. Restart VM(2) |
| 10. VM(2).attachment: `attached` → `detached` | 10. Attach VM(2) to load balancer |

**Fig. 14** A plan for `Rolling`(2) and a workflow for Fig. 9 of $n = 2$.

| Plan | Workflow |
|---|---|
| 1. SW[1].sub: `off` → `on` | 1. Activate standby switch SW[1].sub |
| 2. SW[1].routeBy: `main` → `sub` | 2. Switch route from SW[1].main to SW[1].sub |
| 3. SW[1].main: `on` → `off` | 3. Stop SW[1].main |
| 4. SW[1].main.firmware: `old` → `new` | 4. Update firmware of SW[1].main |
| 5. SW[1].main: `off` → `on` | 5. Activate SW[1].main |
| 6. SW[1].routeBy: `sub` → `main` | 6. Switch route from SW[1].sub to SW[1].main |
| 7. SW[1].sub: `on` → `off` | 7. Stop SW[1].sub |

**Fig. 15** A plan for `UpdToR`(1) and a workflow for Fig. 11 of $n = 1$.

space search using Dijkstra algorithm drastically increases and exceed an hour with $n = 10$ whereas our method can calculate a plan even with $n = 100$ in 10 minutes.

Figure 14 shows a plan for `Rolling`(2) and the corresponding workflow. From begining step 1 to finishing step 5, VM(2) keeps in operation and from begining step 6 to finishing step 10, VM(1) keeps in operation. Therefore, while performing this workflow, the load balancer can distribute workload to at least one of VM(1) and VM(2) and the system of Fig. 9 keeps alive even during the update.

The line graph Fig. 20 also shows that notable difference between Dijkstra algorithm and our heuristic search. The computing time of state space search using Dijkstra algorithm exceeds an hour with $n = 8$ on one hand. On the other hand, our method can calculate a plan even with $n = 60$.

Figure 15 shows a plan for `UpdToR`(1) and the corresponding workflow. From begining step 1 to finishing step 7, core switch and servers in the rack 1 keep connected. Therefore, while performing this workflow, the rack 1 in Fig. 11 keeps in operation.

### 6.5 Evaluation of Estimated Path Length by $\vartheta$

In the above experiments, our heuristic function can estimate precise value of shortest paths of these inputs by calcutating its value of global initial states. That is, our heuristics estimated length of solution of `HV-VM`($n$) as $2n+3$, `Rolling`($n$) as $5n$, and `UpdToR`($n$) as $7n$ at global initial states of each problem. We note that because `Rolling`($n$) (or `UpdToR`($n$)) can be separated into sub-problems $\mathcal{P}_i$ corresponding to each VM(i) (or SW[i]), as me mentioned in Sect. 5.5, we use $\sum_i \vartheta_{\mathcal{P}_i}(\sigma)$ instead of $\vartheta_{\texttt{Rolling}(n)}(\sigma)$ (or $\vartheta_{\texttt{UpdToR}(n)}(\sigma)$).

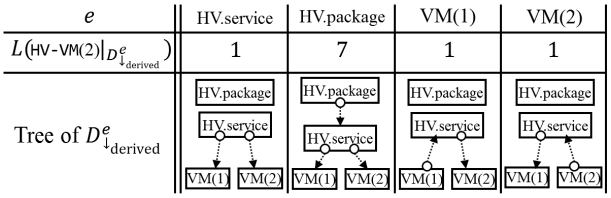


**Fig. 16** The computation of $\vartheta_{\texttt{HV-VM(2)}}$.

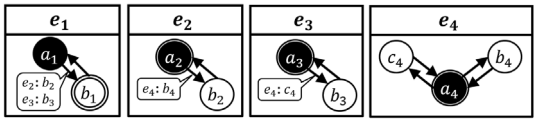


**Fig. 17** Example problem $\mathcal{P}_c$ that $\vartheta_{\mathcal{P}_c}(\sigma_0) \neq L(\mathcal{P}_c)$.

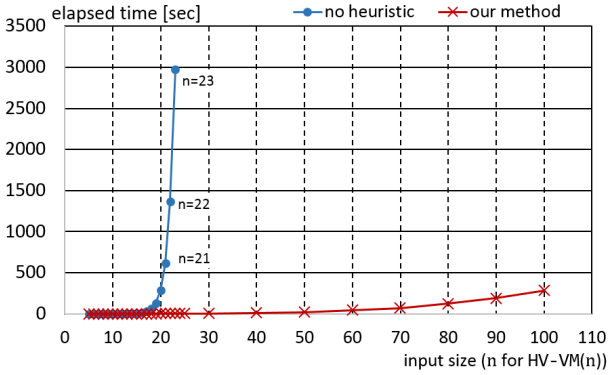


**Fig. 18** The results of experiments using `HV-VM`($n$).

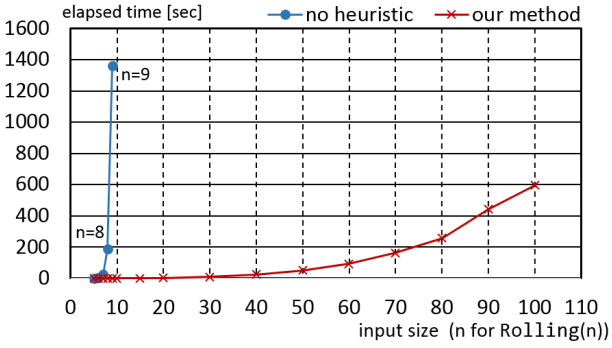


**Fig. 19** The results of experiments using `Rolling`($n$).

Figure 16 illustrates computation of $\vartheta_{\texttt{HV-VM(2)}}$ by showing the value of $L(\texttt{HV-VM(2)}|_{D^e_{\downarrow\text{derived}}})$ and $e$-rooted trees of $D^e_{\downarrow\text{derived}}$ for each element $e$ in `HV-VM`(2). The critical element of `HV-VM`(2) is HV.package because it follows the maximum value of $L(\texttt{HV-VM(2)}|_{D^e_{\downarrow\text{derived}}})$.

In contrast, we introduce an example problem $\mathcal{P}_c$ that $\vartheta_{\mathcal{P}_c}(\sigma_0) \neq L(\mathcal{P}_c)$ by Fig. 17. Any relaxed $\mathcal{P}_c$ have *strictly shorter* solution than the original $\mathcal{P}_c$. Because any derived dependencies cannot depend on both $e_4.b_4$ and $e_4.c_4$, solutions of any relaxed problems don't pass through $e_4.b_4$ and

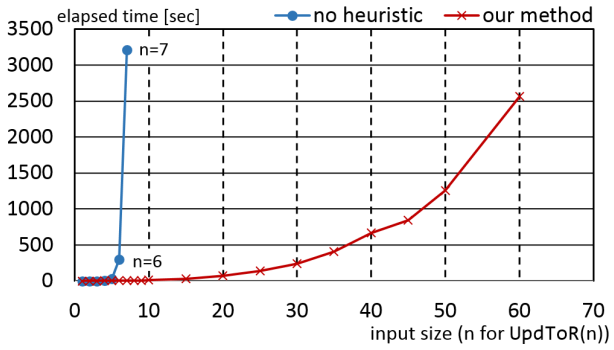


**Fig. 20** The results of experiments using UpdToR($n$).

$e_4.c_4$ at the same time, but solutions of $\mathcal{P}_c$ pass through both of them.

## 7. Conclusion

To alleviate cost and time for management of SDN and NFV systems, or IT systems more generally, we proposed declarative system update scheme which automatically generates system update plans. A novel heuristic function of A* search for declarative system update was proposed to solve the system update planning problem in practical time. An efficient algorithm for calculating the value of the heuristic function was also proposed. Under the assumption that state elements in models are finely enough grained, the proposed algorithm generates system-update plans in sufficiently practical time. We took upgrading running hypervisor and rolling update of running VMs as examples of system update, and showed that our proposed algorithm delivered the update plan within several minutes for a system with 100 VMs, whereas the conventional algorithm is only applicable for very small systems.

## Acknowledgments

This work was partly supported by the Ministry of Internal Affairs and Communications, Japan.

### References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," SIGCOMM Comput. Commun. Rev., vol.38, no.2, pp.69–74, March 2008.

[2] K. Suzuki, K. Sonoda, N. Tomizawa, Y. Yakuwa, T. Uchida, Y. Higuchi, T. Tonouchi, and H. Shimonishi, "A survey on openflow technologies," IEICE Trans. Commun., vol.E97-B, no.2, pp.375–386, Feb. 2014.

[3] E.I. NFV, "Network functions virtualisation - white paper #3." https://portal.etsi.org/Portals/0/TBpages/NFV/Docs/NFV_White_Paper3.pdf

[4] Y.C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computing-a key technology towards 5g," ETSI white paper, vol.11, no.11, pp.1–16, Sept. 2015.

[5] J.G. Herrera and J.F. Botero, "Resource allocation in nfv: A comprehensive survey," IEEE Trans. Netw. Serv. Manag., vol.13, no.3, pp.518–532, Sept. 2016.

[6] X. Cheng, S. Su, Z. Zhang, H. Wang, F. Yang, Y. Luo, and J. Wang, "Virtual network embedding through topology-aware node ranking," SIGCOMM Comput. Commun. Rev., vol.41, no.2, pp.38–47, April 2011.

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

[8] T. Kuroda and A. Gokhale, "Model-based it change management for large system definitions with state-related dependencies," Proc. 2014 IEEE 18th International Enterprise Distributed Object Computing Conference, EDOC'14, pp.170–179, IEEE Computer Society, Washington, DC, USA, 2014.

[9] T. Kuroda, M. Nakanoya, A. Kitano, and A.S. Gokhale, "The configuration-oriented planning for fully declarative IT system provisioning automation," 2016 IEEE/IFIP Network Operations and Management Symposium, NOMS 2016, pp.808–811, Istanbul, Turkey, April 2016.

[10] M. Nakanoya, T. Kuroda, and A. Kitano, "Automated change planning for differential update IT systems with state constraint," 2016 IEEE 20th International Enterprise Distributed Object Computing Workshop (EDOCW), pp.1–9, Sept. 2016.

[11] 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," Proc. 7th ACM/IFIP/USENIX International Conference on Middleware, Middleware'06, pp.404–423, Springer-Verlag, Berlin, Heidelberg, 2006.

[12] H. Herry and P. Anderson, "Planning with global constraints for computing infrastructure reconfiguration," Proc. 2012 AAAI Workshop on Problem Solving Using Classical Planners, AAAI Press, pp.44–50, 2012.

[13] T. Kuwahara, T. Kuroda, M. Nakanoya, Y. Yakuwa, and H. Shimonishi, "Scalable declarative IT system update automation by A* search with critical-element heuristics," submitted to 7th IEEE International Conference on Cloud Networking, CloudNet 2018, Tokyo, Japan, Oct. 2018.

[14] S. Hagen and A. Kemper, "A performance and usability comparison of automated planners for it change planning," Proc. 7th International Conference on Network and Services Management, CNSM'11, pp.143–151, International Federation for Information Processing, Laxenburg, Austria, Austria, 2011.

[15] S. Hagen, N. Edwards, L. Wilcock, J. Kirschnick, and J. Rolia, "One is not enough: A hybrid approach for IT change planning," Integrated Management of Systems, Services, Processes and People in IT, pp.56–70, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[16] S. Hagen, W.L. da Costa Cordeiro, L.P. Gaspary, L.Z. Granville, M. Seibold, and A. Kemper, "Planning in the large: Efficient generation of it change plans on large infrastructures," Proc. 8th International Conference on Network and Service Management, CNSM'12, pp.108–116, International Federation for Information Processing, Laxenburg, Austria, Austria, 2013.

[17] P.E. Hart, N.J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," IEEE Trans. Syst. Sci. Cybern., vol.4, no.2, pp.100–107, July 1968.

[18] E.W. Dijkstra, "A note on two problems in connexion with graphs," Numer. Math., vol.1, no.1, pp.269–271, Dec. 1959.

[19] J. Hoffmann and B. Nebel, "The FF planning system: Fast plan generation through heuristic search," J. Artif. Intell. Res., vol.14, p.2001, 2001.

**Takuya Kuwahara** received his master's degree of information science ad technology from Graduate School of Information Science and Technology, The University of Tokyo in 2015 and has been engaged in research on formal methods for program verification. He joined in NEC Corporation in 2015. Now he is working on researches for automation technology for ICT system design and operation.

**Takayuki Kuroda** received M.E. and Ph.D. degrees from the Graduate School of Information Science, Tohoku University, Sendai, Japan in 2006 and 2009. He joined NEC Corporation in 2009 and has been engaged in research on model-based system management for Cloud applications and Software-Defined networks. As a visiting scalar in the Electrical Engineering and Computer Science department at the Vanderbilt University at Nashville, he studied declarative workflow generation for ICT system update. Now he is working on researched for automation technologies for system design, optimization and operation.

**Manabu Nakanoya** received his B. Engineering degree from Keio University in 2006. He joined NEC Corporation in 2006 and has been engaged in system integration of governmental ICT system. He now works in NEC's System Platform Research Laboratories and is working on researches for automation technologies for ICT system design, optimization, and operation.

**Yutaka Yakuwa** received his M. Engineering degree from Waseda University in 2009. He joined NEC Corporation in 2009 and has been undertaking research on formal methods. He now works in NEC's System Platform Research Laboratories and is engaged in research on software-based automation technologies for ICT system operation and management.

**Hideyuki Shimonishi** received M.E. and Ph.D. degrees from the Graduate School of Engineering Science, Osaka University, Osaka, Japan, in 1996 and 2002. He joined NEC Corporation in 1996 and has been engaged in research on traffic management in high-speed networks, switch and router architectures, and traffic control protocols. As a visiting scholar in the Computer Science Department at the University of California at Los Angeles, he studied next-generation transport protocols. Since then, he engaged in researches on networking technologies including SDN, OpenFlow and NFV for carrier, data center and enterprise networks from their early stages. Now he is working on researches for IoT system platform and automation technologies for system design, optimization, and operation.