Automated Planning of System Rollback in Declarative IT System Update

Takuya Kuwahara, Takayuki Kuroda, Manabu Nakanoya, Yutaka Yakuwa, Yoichi Sato, Yasuhiro Matsunaga
System Platform Research Laboratories, NEC
1753 Shimonumabe, Nakahara-ku, Kawasaki, Kanagawa, Japan
Email: {t-kuwahara@me, t-kuroda@ax, m-nakanoya@bc, y-yakuwa@ap, y-sato@jw, y-matsunaga@bl}.jp.nec.com

***Abstract*—The automation of system management has been expanding, and there has been interest lately in an automated workflow generation that automatically generates the workflows of system updates. However, because these automation technologies operate under the assumption that systems work in accordance with their underlying system model, they are not good at handling unexpected behaviors of target systems.**

In this paper, we propose a way to incorporate unexpected handling into our declarative system update mechanism by automatically generating a "recovery workflow" to roll back a target system in the event of abnormal system stops. We evaluate our tool through a practical three-tier architecture system operating a simple Web service, and found that our method can complete generation of a recovery workflow in one second, and roll back the system from all system states.

***Index Terms*—provisioning automation; declarative; ai planning; change management; system rollback; model-driven management; fault management**

I. Introduction

System update automation has been developed to release system operators from complicated manual jobs during operation. Declarative system update [1], [2], [3] is one of the approaches for system update automation. The tools that adopt this approach (e.g., Puppet [4] and Ansible [5]) receive *the desired state* of a managed system from system operators, generate a change plan from the current state to the desired state automatically, and apply the change plan to the managed system. Declarative system update enables system operators to update systems in an intuitive and direct manner by allowing them to focus on the goal of system updates without having to worry about the actual update process.

However, model-based approaches have an essential weakness to *abnormal behaviors*. In most cases, generated change plans aren't designed to address exceptional behaviors (e.g., physical hazards). Because generated change plans is guaranteed to be correct only under the assumption that managed systems will behave according to the models, abnormal incidents will cause an abnormal termination, or other unexpected behaviors.

978-3-903176-15-7

In order to address abnormal system behaviors, we need to prepare extra change plans to restore a managed system from all abnormal system states. However, it is inefficient to individually prepare plans for each abnormal state in advance, because it often takes a long time and a large amount of memory to generate and store all plans for restoration.

In this work, we presented a way to incorporate rollback supports into our declarative system update tool by generating a *recovery workflow* to roll back target systems from error states to the initial state. Our proposed tool prepares recovery workflows for all abnormal states in advance, even if the workflow for an update is partially ordered and has many possible abnormal states. Our proposed tool doesn't compute recovery workflows individually but rather integrates information for recovery from all abnormal states, which we call a *recovery scheme*. When a system update halts abnormally, our tool generates a recovery workflow by combining the recovery scheme with the executed tasks in the workflow.

In Section II, we provide an overview of our declarative system update tool and the proposed tool incorporated with rollback support. In Section III, we introduce system and workflow models. In Section IV-VII, we describe the technical details of rollback support. In Section VIII, we evaluate our tools through a realistic case scenario. In Section IX, we discuss related work. We conclude in Section X with a brief summary.

II. Overview

Figure II.1 shows our declarative system update tool proposed in an earlier work [6], [3], which consists of the

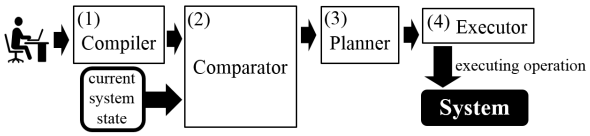


Figure II.1. Overview of declarative system update.

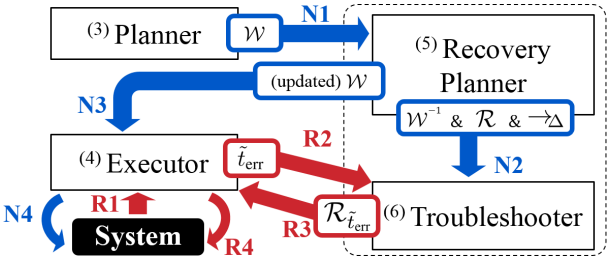


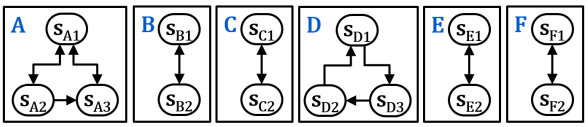
Figure II.2. A part of the declarative system update with recovery.

following four components. (1) *Compiler* converts from a human-editable system model to a finer-grained system model. (2) *Comparator* compares *current and desired states of systems*, and generates planning problem instances whose solutions are workflows of system update. (3) *Planner* generates a workflow of system updates by solving the planning instance. (4) *Executor* executes the workflow generated by the planner and updates the target system.

Figure II.2 shows a part of our declarative system update tool supporting automated system rollback. We add two components to our tool: (5) *Recovery planner* prepares *a recovery scheme*, which is information of recoveries for all errors during updates, and the execution order of the workflow updates for recovery procedure. (6) *Troubleshooter* receives executed tasks from the executor and then immediately builds a recovery workflow using the recovery scheme.

As Figure II.2 shows, the declarative system update with recovery follows a normal procedure with steps **N1,...** for update and follows a recovery procedure with steps **R1,...** for rollback. The normal procedure runs as follows: **(N1)** the planner generates a workflow $\mathcal{W}$, **(N2)** the recovery planner generates a recovery scheme and **(N3)** updates $\mathcal{W}$, and **(N4)** the executor executes $\mathcal{W}$.

When a system update stops abnormally in Step **N4**, the recovery procedure runs as follows: **(R1)** the executor detects abnormal stops, **(R2)** the troubleshooter receives *an error trace* $\tilde{t}_{\text{err}}$ and **(R3)** generates a recovery workflow $\mathcal{R}_{\tilde{t}_{\text{err}}}$, and **(R4)** the executer executes $\mathcal{R}_{\tilde{t}_{\text{err}}}$.



Dependencies:

source	target	dependency
s_{A1}	s_{A2}	{D:{s_{D1}}}
s_{A1}	s_{A3}	{D:{s_{D1}}}
s_{A2}	s_{A1}	{D:{s_{D1}}}
s_{A2}	s_{A3}	{D:{s_{D1}}}
s_{A3}	s_{A1}	{D:{s_{D1}}, E:{s_{E1}}}

source	target	dependency
s_{B2}	s_{B1}	{A:{s_{A2},s_{A3}}}
s_{C2}	s_{C1}	{A:{s_{A3}}}
s_{E1}	s_{E2}	{B:{s_{B2}}, C:{s_{C2}}}
s_{E2}	s_{E1}	{B:{s_{B2}}, C:{s_{C2}}, F:{s_{F2}}}
(others)		{}

Figure III.1. A running example $\mathcal{S}_{\text{expl}}$.

III. System Model and Workflow Model

First, we introduce our model of systems (cf. [3]).

We first introduce *system components* and *dependencies*. System components are modeled by a finite set of their *states* and *transitions*. A system component e has its current state and moves to another state by its transition. When a transition τ transits e's current state "a" to another state "b", τ is denoted as "$e : a \to b$".

A dependency is a condition for executing a transition τ, denoted as $\{e_1 : S_1, \ldots, e_k : S_k\}$; each e_i is a system component and S_i is a nonempty proper subset of states of e_i. This means τ can be executed only when e_i's current state is in S_i for all e_i.

Definition III.1 (System). *A system $\mathcal{S}$ is defined as a set of system components. A state σ of a system $\{e_1, \ldots, e_k\}$ is defined as a mapping from each element to its state, which is denoted by $\{e_1 : s_1, \ldots, e_k : s_k\}$.*

Example III.1. *Figure III.1 shows a system $\mathcal{S}_{\text{expl}}$. As shown in the upper part of Figure III.1, $\mathcal{S}_{\text{expl}}$ includes six system components $A, B, \ldots, F$, which are shown as rectangles containing labeled circles and arrows representing the states and transitions, respectively.*

The table in the lower part of Figure III.1 shows all dependencies in $\mathcal{S}_{\text{expl}}$. For example, "$B : s_{B2} \to s_{B1}$" has a dependency $\{A : \{s_{A2}, s_{A3}\}\}$.

Next, we define workflow for system update. First, we introduce the notion of *tasks* to distinguish the same transition occurred in system update twice or more.

Definition III.2 (Task). *A task $t = \tau_i$ is a transition τ indexed by a natural number $i = 0, 1, 2, \ldots$.*
When τ is e's transition, t is called an e-task.

When t is τ_i and τ's dependency is $\{e_1 : S_1, \ldots, e_n : S_n\}$, we use the following notation; (1) "$\text{tgt}(t)$" is the target state of τ and (2) "$\mathbb{D}(t, e_i)$" is the states S_i.

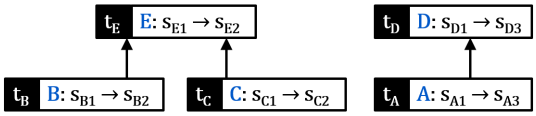


Figure III.2. Executable workflow $\mathcal{W}_{\text{expl}}$ of $\mathcal{S}_{\text{expl}}$

Then, we can define system update from a state σ by transitions of system components as follows.

Definition III.3 (System update)**.** *An e-task $t = \tau_i$ is said to be executable at σ when τ is executable at σ.*

When t is executable at σ, we define the "updated" state $(\sigma \downarrow t)$ as follows; (1) $(\sigma \downarrow t)(e) = \text{tgt}(t)$ and (2) $(\sigma \downarrow t)(e') = \sigma(e')$ for all $e' \neq e$.

When $((\sigma \downarrow t_1) \downarrow t_2) \dots \downarrow t_n$ can be defined, we denote this state by $\sigma \downarrow \langle t_1, t_2, \dots, t_n \rangle$, and the sequence $\langle t_1, t_2, \dots, t_n \rangle$ is said to be executable at σ.

Finally, we define the workflow as a partially ordered set of tasks, and *the executability* of a workflow as that all topological sorting of the workflow is executable.[1]

Definition III.4 (Workflow)**.** *An ordered set $\mathcal{W} = (T, \to_{\mathcal{W}})$ is said to be a workflow of $\mathcal{S}$ where T is a set of tasks of $\mathcal{S}$ and $\to_{\mathcal{W}}$ is a partial order on T.*

$\mathcal{W}$ is said to be executable at σ when all topological sorting of $\mathcal{W}$ is executable at σ. Then, σ updated by $\mathcal{W}$ is denoted by $\sigma \downarrow \mathcal{W}$.

When a sequence $\langle t_1, \dots, t_n \rangle$ is a prefix of a topological sorting of $\mathcal{W}$, we call it a trace on $\mathcal{W}$.

Example III.2. *Figure III.2 shows a workflow $\mathcal{W}_{\text{expl}}$ of $\mathcal{S}_{\text{expl}}$ that consists of five tasks $t_A, \dots, t_E$ represented by rectangles. Arrows in Figure III.2 specify execution order. That is, an arrow from t_C to t_E means that t_E can be executed after the execution of t_C. For example, $\langle t_B, t_A, t_C, t_D, t_E \rangle$ is one of the trace from σ_{expl}.*

The workflow $\mathcal{W}_{\text{expl}}$ is executable at $\sigma_{\text{expl}} = \{A : s_{A1}, B : s_{B1}, C : s_{C1}, D : s_{D1}, E : s_{E1}, F : s_{F1}\}$.

IV. Reversing Workflow for Rollback

Suppose that we plan to update system $\mathcal{S}$ from σ by a workflow $\mathcal{W} = (T, \to_{\mathcal{W}})$.In this section, we discuss how to plan "recovery workflow" for all traces of $\mathcal{W}$.

A. Reversed workflow

As stated in Section I, planning recovery workflows for all traces often requires an unrealistically large amount of time and memory for computating and storing the workflows. So we apply the rollback technique by invoking reversing operations [7], which perform *undo* tasks corresponding to executed tasks in a workflow. That is, we build *a reversed workflow* as follows.

Definition IV.1. *When τ_i is a task, a task τ_i^{-1} is defined as $(\tau^{-1})_i$ and called the reversed task of τ_i.*

A reversed workflow of $\mathcal{W} = (T, \to_{\mathcal{W}})$ can be defined as $\mathcal{W}^{-1} = (\{T^{-1}, \to_{\mathcal{W}^{-1}}\})$ where $T^{-1} = \{t^{-1} \mid t \in T\}$ and $\to_{\mathcal{W}^{-1}} = \{(t_2^{-1}, t_1^{-1}) \mid (t_1, t_2) \in \to_{\mathcal{W}}\}$.

If $\mathcal{W}^{-1}$ is executable, for a trace $\tilde{t}_{\text{err}}$, $\mathcal{W}^{-1}$ can roll $\mathcal{S}$ back from $\sigma \downarrow \tilde{t}_{\text{err}}$ by only executing the part of $\mathcal{W}^{-1}$ restricted to reversed tasks of $\tilde{t}_{\text{err}}$. We denote the restricted $\mathcal{W}^{-1}$ to reversed $\tilde{t}_{\text{err}}$ by $\mathcal{W}^{-1}|_{\tilde{t}_{\text{err}}}$.

B. Rollback for irreversible tasks

Unfortunately, $\mathcal{W}^{-1}$ is, in most cases, not executable. Namely, some tasks aren't inversely executable in $\mathcal{W}^{-1}$. In this paper, we focus on *reversibility of tasks*.

Definition IV.2 (Reversiblity)**.** *A task t is said to be reversible on $\mathcal{W}$ if for all traces $\langle t_1, \dots, t_k, t \rangle$ on $\mathcal{W}$, t^{-1} is executable at $\sigma \downarrow \langle t_1, \dots, t_k, t \rangle$.*

Intuitively, when t is reversible, t^{-1} is executable at all states immediately after t is executed.

Example IV.1. *All tasks in $\mathcal{W}_{\text{expl}}$ are irreversible.*

Note that t_A can be rolled back before execution of t_E, t_B and t_C can be rolled back after execution of t_A, but t_D and t_E cannot be executed in any execution.

The recovery planner addresses these irreversible tasks by two prosesses — *reordering* and *backward planning*. In reordering process, the recovery planner makes some irreversible tasks reversible by changing the execution order of $\mathcal{W}$. In backward planning process, the recovery planner plans workflows from states after execution of remaining irreversible tasks to states before it.

V. Reordering to Maximize Reversibility

The term "reordering" [8] refers modifications to change ordering relations in workflows. Formally, a reordered workflow of $(T, \to)$ is defined as a workflow $(T, \to')$. Suppose that we obtain an executable workflow $\mathcal{W} = (T, \to_{\mathcal{W}})$, and some tasks in T are irreversible. Our goal is to maximize the number of reversible tasks in T by reordering $\mathcal{W}$.

We were inspired by Muise et al.'s studies [9] and adopted the reordering technique based on *partial maximum satisfiability problem (partial Max-SAT)*. Partial

[1] A topological sorting of $(A, \to)$ is defined as a sorted sequence $\langle a_1, \dots, a_n \rangle$ of A in the order where $a_j \to a_i$ does not hold for any $0 \leq i < j \leq n$.

Max-SAT is the problem given by set of variables and two sets of formulae called *hard clauses* and *soft clauses* and determining assignments **true** or **false** to the variables such that all hard clauses and the maximum number of soft clauses evaluate **true**. For example, when $(A \wedge C) \vee B$ and $B \Rightarrow (\neg C)$ are given as hard clauses and A, B and C as soft clauses, the assignment $(A, B, C) = (\textbf{true}, \textbf{false}, \textbf{true})$ is a solution.

We can encode the problem that maximizing the number of reversible tasks in a given workflow as a partial Max-SAT problem. First, we encode the validity as an exectable workflow and the compatibility with the original workflow as hard clauses. Second, we encode the reversibility of tasks as soft clauses.

We introduce the notion of *backward support* to encode reversibility of tasks. For brevity, we discuss a task t such that t^{-1}'s dependency is $\{e : \{s_e\}\}$. In order to guarantee reversibility of t, we need to ensure that e's current state is s_e immediately after execution of t. Specifically, either of the following two condition needs to be satisfied; (1) the last e-task executed before execution of t is $t_e = (e : \_ \to s_e)$. (2) $\sigma(e) = s_e$ holds and no e-task executed before execution of t. Additionally, we can integrated the conditions (1) and (2) into (1) by adding "imaginary" task $t_{(\sigma,e)} = (e : \_ \to \sigma(e))$, which are executed before any other tasks in $\mathcal{W}$. We denote T with tasks $t_{(\sigma,e)}$ for all e as $\hat{T}$. When (1) is satisfied, t_e is said to be *backwardly supporting* t.

We introduce three variables, κ, β, and θ, as follows:

- For all $t_1, t_2 \in \hat{T}$, $\kappa(t_1, t_2)$ indicates t_1 executes before t_2.
- For all $t_1 \in \hat{T}$, $t_2 \in T$, such that t_2 has its reversed task t_2^{-1} and $\mathrm{tgt}(t_1) \in \mathbb{D}\left(t_2^{-1}, e\right)$, $\beta(t_1, t_2)$ indicates t_1 backwardly supports t_2.
- For all $t \in T$, such that t has its reversed task t^{-1}, $\theta(t)$ indicates t is reversible in $\mathcal{W}$.

An assignment to κ induces a binary relation over T. We denote it $\to_\kappa$. As explained in detail later, we strengthen the ordering $\to_\mathcal{W}$ to $\to_\kappa$ so that the number of reversible tasks is maximized. Then, we formalize the conditions for that $(T, \to_\kappa)$ is *executable*, $\to_\kappa$ is a subset of $\to_\mathcal{W}$, and $\theta(t)$ indicates t's reversibility.

First, we define formulae that ensure the relation $\to_\kappa$ is partial ordering compatible with $\to_\mathcal{W}$ as follows:

$$\textbf{(1)}\neg\kappa(t,t) \qquad \textbf{(2)}\kappa(t_1,t_2) \wedge \kappa(t_2,t_3) \Rightarrow \kappa(t_1,t_3)$$
$$\textbf{(3)}\neg\kappa(t_2,t_1) \quad (\forall t_1, t_2 \in T, \text{ s.t. } t_1 \to_\mathcal{W} t_2)$$

Second, we add the conditions of the property of $t_{(\sigma,e)}$.

$$\textbf{(4)}\neg\kappa(t, t_{(\sigma,e)}) \wedge \kappa(t_{(\sigma,e)}, t)$$

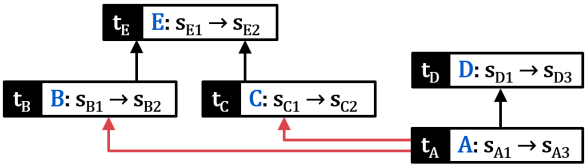


Figure V.1. A workflow $\mathcal{W}'_{\mathrm{expl}}$ obtained by reordering $\mathcal{W}_{\mathrm{expl}}$.

Third, we define formulae that ensure variables $\beta(t_1, t_2)$ correctly indicate that t_1 backwardly supports t_2. The following condition (5) ensures that there is no e-task t_{del} such that $t_1 \to_\mathcal{W} t_{\mathrm{del}} \to_\mathcal{W} t_2$.

$$\textbf{(5)}\beta(t_1,t_2) \Rightarrow \bigwedge_{\substack{t_{\mathrm{del}} \in T \text{ s.t.} \\ t_{\mathrm{del}} = \mathrm{tgt}(t_1) \to \_}} \kappa(t_{\mathrm{del}}, t_1) \vee \kappa(t_2, t_{\mathrm{del}})$$

Finally, we define a formula that ensures variables θ correctly indicate that reversibility of tasks.

$$\textbf{(6)}\theta(t) \Rightarrow \bigwedge_{\substack{e \in \mathcal{S} \text{ s.t.} \\ t \text{ depends } e}} \bigvee_{s \in \mathbb{D}(t,e)} \bigvee_{\substack{t_{\mathrm{sup}} \in \hat{T} \text{ s.t.} \\ \mathrm{tgt}(t_{\mathrm{sup}}) = s}} \kappa(t_{\mathrm{sup}}, t) \wedge \beta(t_{\mathrm{sup}}, t)$$

By these formulae, we can maximize the number of reversible tasks in a workflow $(T, \to_\mathcal{W})$ by using partial Max-SAT solvers as follows: **(Step 1)** Construct formulae $(1), \ldots, (6)$, **(Step 2)** by a partial Max-SAT solver, solve the problem whose hard clauses are $(1), \ldots, (6)$ and soft clauses are $\theta(t)$ for all t, and **(Step 3)** construct $(T, \to_\kappa)$ from the solution.

Example V.1. *After Step 1 for $\mathcal{W}_{\mathrm{expl}}$, the formulae (5) and (6) for t_A, t_B and t_C is obtained as follows:*

$$\beta(t_{(\sigma_0,E)}, t_A) \Rightarrow \kappa(t_E, t_{(\sigma_0,E)}) \vee \kappa(t_A, t_E)$$
$$\beta(t_A, t_B) \Rightarrow \textbf{true}, \beta(t_A, t_C) \Rightarrow \textbf{true}$$
$$\theta(t_A) \Rightarrow \kappa(t_{(\sigma_0,E)}, t_A) \wedge \beta(t_{(\sigma_0,E)}, t_A)$$
$$\theta(t_B) \Rightarrow \kappa(t_A, t_B) \wedge \beta(t_A, t_B)$$
$$\theta(t_C) \Rightarrow \kappa(t_A, t_C) \wedge \beta(t_A, t_C)$$

*After Step 2, we obtain an assignment such that $\kappa(t_A, t_B) = \kappa(t_A, t_B) = \kappa(t_A, t_C) = \kappa(t_A, t_E) =$ **true** and $\theta(t_A) = \theta(t_B) = \theta(t_C) =$ **true** hold.*

Figure V.1 shows a reordered workflow $\mathcal{W}'_{\mathrm{expl}} = (T, \to_\kappa)$, whose ordering is strengthened by adding $(t_A, t_B), (t_A, t_C)$ and (t_A, t_E). In contrast to $\mathcal{W}_{\mathrm{expl}}$, $\mathcal{W}'_{\mathrm{expl}}$ enables t_A, t_B, and t_C to be backwardly executed.

VI. Backward Planning

Even after the reordering process, some tasks are left irreversible. In this section, we denote these irreversible tasks by $\mathcal{I}_\mathcal{W}$, and discuss a method to roll them back.

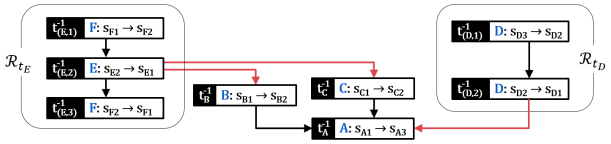


Figure VI.1. A workflow generated by combining $\mathcal{W}_{\text{expl}}^{-1}$, $\mathcal{R}_{t_D}$ and $\mathcal{R}_{t_E}$ by $\to_{\Delta\{t_D,t_E\}}$.

A. Plan workflows for rolling back irreversible tasks

In backward planning process, the recovery planner plans *reversing workflows* for each $t_{\text{I}} \in \mathcal{I}_{\mathcal{W}}$ to roll $\mathcal{S}$ back to the state before executing t_{I}. However, the recovery planner cannot plan reversing workflows in the same way as the planner planning $\mathcal{W}$, because the $\mathcal{S}$'s state after execution of t_{I} isn't determined uniquely. For example, in Example V.1, D's state can be either s_{D1} or s_{D2} after execution of t_E. Thus, we focus on a *part* of $\mathcal{S}$, whose state can be uniquely determined immediately after execution of t_{I}, and then plan a reversing workflow $\mathcal{R}_{t_{\text{I}}}$ rolling back the part to the state before executing t_{I}.

Example VI.1. *Let us consider the state of $\mathcal{S}_{\text{expl}}$ immediately after execution of t_D in the execution of $\mathcal{W}_{\text{expl}}$ from σ_{expl}. First, the D's state changed to s_{D3}. Second, the A's state must be s_{A3}, because $t_A \to_{\mathcal{W}_{\text{expl}}} t_D$ is specified. Third, the F's state stays the initial state s_{F1}. Finally, states of B, C and E are not uniquely determined. Thus, in the backward planning process, $\mathcal{R}_{t_D}$ is planned as a workflow from $\{A : s_{A3}, D : s_{D3}, F : s_{F1}\}$ to $\{A : s_{A3}, D : s_{D1}, F : s_{F1}\}$, which shows in Figure VI.1 as a rounded rectangles labeled as $\mathcal{R}_{t_D}$.*

B. Combine reversing workflow and reversed workflow

When the troubleshooter receives a trace $\tilde{t}_{\text{err}}$ including $T_{\text{I}} = \{t_{\text{I}(1)}, \ldots, t_{\text{I}(m)}\} \subseteq \mathcal{I}_{\mathcal{W}}$, then the troubleshooter *combines* the restricted reversed workflow $\mathcal{W}^{-1}|_{\tilde{t}_{\text{err}}}$ and the reversing workflows $\mathcal{R}_{t_{\text{I}(1)}}, \ldots, \mathcal{R}_{t_{\text{I}(m)}}$ into an executable workflow $\mathcal{R}_{\tilde{t}_{\text{err}}}$. This *combining* process consists of two phases: (1) replacing $t_{\text{I}(1)}^{-1}, \ldots, t_{\text{I}(m)}^{-1}$ in $\mathcal{W}^{-1}|_{\tilde{t}_{\text{err}}}$ by $\mathcal{R}_{t_{\text{I}(1)}}, \ldots, \mathcal{R}_{t_{\text{I}(m)}}$, (2) adding such ordering relations as the combined workflow become executable.

For the second phase, in advance, the recovery planner computes the ordering relation $\to_{\Delta T_{\text{I}}}$ for all T_{I} such that the combined workflow $\mathcal{R}_{\tilde{t}_{\text{err}}}$ is executable for all traces $\tilde{t}_{\text{err}}$ including T_{I}. We omit the details of the algorithm to compute $\to_{\Delta T_{\text{I}}}$ due to space limitations.

Example VI.2. *By combining $\mathcal{W}_{\text{expl}}^{-1}$, $\mathcal{R}_{t_D}$ and $\mathcal{R}_{t_E}$, we can obtain an executable workflow shown in Figure VI.1.*

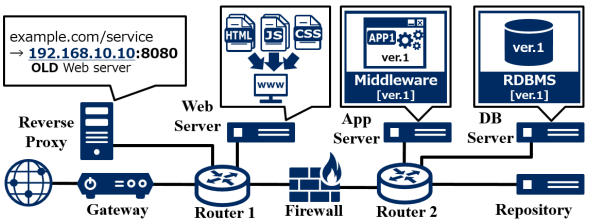


Figure VIII.1. Case scenario: A Web service on three-tier architecture.

VII. Generate Recovery Workflow

The recovery planner computes (1) a reversed workflow $\mathcal{W}$, (2) reversing workflows $\mathcal{R}_{t_{\text{I}}}$ for each $t_{\text{I}} \in \mathcal{I}_{\mathcal{W}}$ and (3) additional ordering relation $\to_{\mathcal{W}}$ to combine them. The recovery planner passes them to the troubleshooter as *a recovery scheme for $\mathcal{W}$*.

When system update halts abnormally, the executor passes a trace $\tilde{t}_{\text{err}}$ of executed tasks to the troubleshooter. Then the troubleshooter, first, takes all irreversible tasks T_{I} in $\tilde{t}_{\text{err}}$, second, obtains reversing workflows for T_{I} and $\to_{\Delta T_{\text{I}}}$ from the recovery scheme, and finally, combines reversing workflows for T_{I} and $\mathcal{W}^{-1}|_{\tilde{t}_{\text{err}}}$ into $\mathcal{R}_{\tilde{t}_{\text{err}}}$.

VIII. Evaluation

We evaluate our method to roll systems back using a more complex and large-scale case study than $\mathcal{S}_{\text{expl}}$.

A. Case scenario: Reforming a three-tier Web system

Our case scenario is based on a Web service operated by a system on three-tier architecture, which is shown in Figure VIII.1. We assume the system administrator plan to reform this system entirely as follows: (1) replace the Web server, (2) reconfigure the reverse proxy, (3) add a newly developed application `App2`, (4) update the database, and (5) upgrade versions of `App1`, the middleware and the relational database management system.

The package of applications `App1`, `App2` and the database are managed by a version control system on a repository server. In the current state, each repository refers to the latest version. When we downgrade them, each repository needs to refer to the previous version.

B. Planning workflow

Figure VIII.2 shows a workflow planned by the planner. (Note that the workflow doesn't include dotted arrows.) The workflow includes five irreversible tasks: (A) undeploy `App1` of the previous version, (B) remove the unpacked `App1.war` of the previous version, (C) update `APP1.war`, (D) remove database of the previous version, and (E) stop the httpd server.

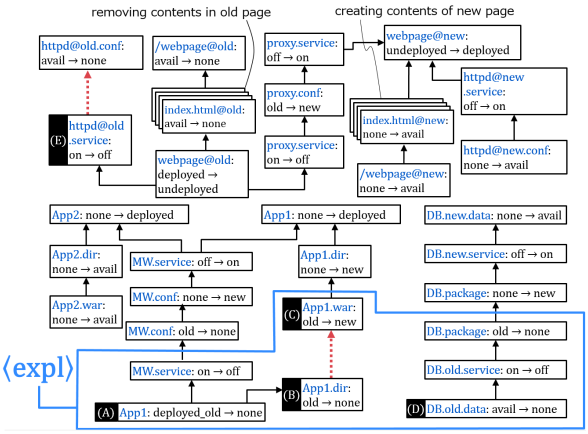


Figure VIII.2. Planned workflow for case scenario and additional ordering by our reordering technique.

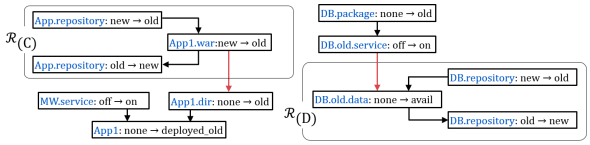


Figure VIII.3. Recovery workflow for ⟨expl⟩ in Figure VIII.2.

C. Recovery scheme and Recovery workflow

The two dotted arrows in Figure VIII.2 shows additional ordering added by the reordering phase of the recovery planner. After reordering, the three tasks (A), (B) and (E) become reversible, but the tasks (C) and (D) left irreversible. Thus, the recovery planner plans reversing workflows $\mathcal{R}_{(C)}$ and $\mathcal{R}_{(D)}$ and additional ordering relations for $\{(C)\}$, $\{(D)\}$ and $\{(C),(D)\}$. Figure VIII.3 shows the recovery workflow for the tasks surrounding by the border ⟨expl⟩ in Figure VIII.2.

Remarkably, reversed workflows generated by reversing executed part of the workflow in Figure VIII.2 aren't executable at 99.5 percent of their error states. This result emphasized importance of reordering and backward planning of the recovery planner.

The experiment was performed by a prototype of our approach implemented by Scala [10]. On a machine with Intel(R) Xeon(R) E5-1620 0 (3.60 GHz, 32 GB memory), our prototype implementation computes the workflow of Figure VIII.2 in about 740 milliseconds and with 48.7 MB memory, and prepares for rollback by reordering and backward planning in about 210 milliseconds and with 15.3 MB memory. These results demonstrate that our approach can plan workflows with rollback support for practical systems such as three-tier architecture.

IX. Related Work

Machado et al. [7] proposed an IT change management system with rollback support. Their architecture enables IT change management systems to roll back the managed systems with *atomicity marks* specifying which rollback procedure to use. Their rollback support works properly under the assumption that all *undo* actions for executed tasks can be performed in reversed order of the execution order. However, as shown in Section VIII, undo tasks cannot be performed in some cases.

Hagen and Kemper [11] proposes another approach to address unexpected incidents in system changes. Like our study, their approach named *parameter adaption* also uses the structure of the original workflow planned for normal procedure, and only adapts the parameters in the workflow to become feasible in the system after unexpected incidents. Their approach can continue updating system even after unexpected incidents, but cannot address such an error state as the workflow needs to change its structure for recovery.

X. Conclusion and Future Work

In this paper, we proposed an IT system update automation that can roll back its target IT system when updates halt due to abnormal incidents. By our approach, a workflow for a system update is augmented by additional execution ordering such that we can execute inversely as many tasks as possible. When some tasks are left irreversible even after this process, workflows to roll back such tasks are individually planned. Additionally, we show that our approach can be applied to the update of a three-tier architecture system in one second.

We assume that each workflow can be rolled back to the state before its execution. However, some system states are potentially irreversible for all recovery workflows. In such cases, we can only roll back the system to transient states of the update, where it may be unsafe to stay. As a future work, we will improve our approach to address this issue. Specifically, system operators will specify states where the system can safely stay, and the target system will be rolled back to these specified states when the system abnormally halts and cannot be rolled back to its initial state.

Acknowledgments

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

REFERENCES

[1] 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, 2006.

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

[3] T. Kuroda, M. Nakanoya, A. Kitano, and A. S. Gokhale, "The configuration-oriented planning for fully declarative IT system provisioning automation," in *2016 IEEE/IFIP Network Operations and Management Symposium, NOMS 2016*.

[4] "Puppet," https://puppet.com/.

[5] "Ansible," https://www.ansible.com/.

[6] T. Kuroda and A. Gokhale, "Model-based it change management for large system definitions with state-related dependencies," in *Proceedings of the 2014 IEEE 18th International Enterprise Distributed Object Computing Conference*, ser. EDOC '14, 2014.

[7] G. S. Machado, F. F. Daitx, W. L. d. C. Cordeiro, C. B. Both, L. P. Gaspary, L. Z. Granville, C. Bartolini, A. Sahai, D. Trastour, and K. Saikoski, "Enabling rollback support in it change management systems," in *NOMS 2008 - 2008 IEEE Network Operations and Management Symposium*, April 2008, pp. 347–354.

[8] C. Bäckström, "Computational aspects of reordering plans," *J. Artif. Int. Res.*, vol. 9, no. 1, pp. 99–137, Sep. 1998. [Online]. Available: http://dl.acm.org/citation.cfm?id=1622797.1622800

[9] C. Muise, S. A. McIlraith, and J. C. Beck, "Optimally relaxing partial-order plans with maxsat," in *Proceedings of the Twenty-Second International Conference on International Conference on Automated Planning and Scheduling*, ser. ICAPS'12. AAAI Press, 2013, pp. 358–362. [Online]. Available: http://dl.acm.org/citation.cfm?id=3038546.3038591

[10] "The scala programming language," https://www.scala-lang.org/.

[11] S. Hagen and A. Kemper, "Facing the unpredictable: Automated adaption of it change plans for unpredictable management domains," in *2010 International Conference on Network and Service Management*, Oct 2010, pp. 33–40.