Supporting Undoability in Systems Operations

Ingo Weber[1,2], Hiroshi Wada[1,2], Alan Fekete[1,3], Anna Liu[1,2], and Len Bass[1,2]

[1]NICTA, Sydney
[2]School of Computer Science and Engineering, University of New South Wales
[3]School of Information Technologies, University of Sydney
[1]{*firstname.lastname*}*@nicta.com.au*

Abstract

When managing cloud resources, many administrators operate without a safety net. For instance, inadvertently deleting a virtual disk results in the complete loss of the contained data. The facility to undo a collection of changes, reverting to a previous acceptable state, is widely recognized as valuable support for dependability. In this paper, we consider the particular needs of the system administrators managing API-controlled resources, such as cloud resources on the IaaS level. In particular, we propose an approach which is based on an abstract model of the effects of each available operation. Using this model, we check to which degree each operation is undoable. A positive outcome of this check means a formal guarantee that any sequence of calls to such operations can be undone. A negative outcome contains information on the properties preventing undoability, e.g., which operations are not undoable and why. At runtime we can then warn the user intending to use an irreversible operation; if undo is possible and desired, we apply an AI planning technique to automatically create a workflow that takes the system back to the desired earlier state. We demonstrate the feasibility and applicability of the approach with a prototypical implementation and a number of experiments.

1 Introduction

Cloud computing is now a market larger than US$ 150 billion worldwide, out of which Infrastructure-as-a-Service (IaaS) is estimated at US$ 80 billion [8]. Administrators executing applications on an IaaS platform must provision, operate, and release the resources necessary for that application. Administrators perform these activities through the use of a fixed set of API operations - those made available by the IaaS provider. The APIs may be exposed as system administration commands, Web service interfaces, programming language-specific APIs, or by similar means.

When managing cloud resources, many administrators operate without a safety net – for instance, inadvertently deleting a virtual disk can result in the complete loss of the contained data. Reverting to an earlier state manually is difficult, especially for administrators with limited experience. The facility to rollback a collection of changes, i.e., reverting to a previous acceptable state, a *checkpoint*, is widely recognized as valuable support for dependability [3, 5, 13]. This paper considers the particular needs of administrators managing API-controlled systems to revert to a previous acceptable state. This reversion can be called an "undo" or "rollback".

The need for undo for administrators is partially motivated by the power of the provided APIs. Such APIs can dramatically improve the efficiency of certain aspects of system administration; but may increase the consequences of human-induced faults. These human induced faults play a large role in overall dependability [27, 28].

To support the administrator managing cloud resources through an API, we aim to provide undoability to a previously known consistent state. If the current state is unsatisfactory for any reason, a consistent earlier state (or a checkpoint) can be restored. This is similar to the concept of "transactional atomicity" in that the period from the consistent earlier state to a future consistent state can be considered as a transaction that may be committed or undone.

However, the nature of a cloud platform introduces particular difficulties in implementing undo. One fundamental issue is the **fixed set of operations:** the administrator cannot alter the set of operations provided by the cloud provider in an API nor even examine its implementation. Administrators have to accept a given API, which is not necessarily designed to support undo. One implication of this inability to change the API is that it is not totally possible to introduce locks to prevent multiple administrators from attempting concurrent undos. It is possible to introduce a lock mechanism on top of our

undo but since administrators can directly use the original API, they could circumvent the locks. We therefore assume that single user undo is enforced by administrative and not technical means.

In the following, we list particular challenges for undoability in the environment outlined above.

1. **Completely irreversible operations.** For some operations, like deleting a virtual disk, no compensating operation is provided. Although it is possible to recover data stored in a disk if the backup exists, a disk itself cannot be restored once it is deleted when no such functionality is supported by a cloud provider.

2. **Partly irreversible operations.** Some operations are mostly undoable, but not fully so. For example, starting a machine seems to be an inverse operation for stopping it. However, certain properties cannot necessarily be restored: any property that is set by the cloud provider when starting a machine cannot be guaranteed to match the value in the checkpointed state. Examples include the startup timestamp, dynamically allocated private or public IP addresses or DNS names, etc. Restoring the exact values of those properties may or may not be important in a given administration task.

3. **As-is API, requiring potentially complex undo workflows.** Since the user can manipulate the state of the resources only through the provided API, restoring a previous acceptable state can only be achieved by *calling the right API operations on the right resources in the right order*. A common assumption is that a sequence of operation calls can be undone by calling inverse operations in the reverse chronological order – e.g., as suggested in ref. [14]; but this is not always correct or optimal [16]. Further, constraints between operation calls can be non-obvious and state-specific. Contrast our accomplishing an undo via provided APIs with undos based on provider furnished tools such as ZFS snapshotting. ZFS snapshotting enables the recovery of a dataset via internal mechanisms. Administrators do not have access to such internal mechanisms and so need to cope with the features the provider made accessible.

4. **Error-prone operations.** Cloud API operations are often themselves error-prone: we frequently observed failures or timeouts on most major commercial cloud platforms. Therefore, failures may occur during the execution of an undo workflow, and need to be handled, such as flexibly executing alternative operations.

To support undoability on existing cloud platforms or similar API-controlled systems, we address the four issues listed above as follows.

1. **Introduction of Pseudo-Delete.** An irreversible operation, like delete, is replaced by corresponding pseudo variant of the operation, like *pseudo-delete* (see e.g., ref. [15]). Pseudo-delete first marks a resource for deletion while retaining ownership; the resource is actually released only when the whole operation sequence is successfully completed.

2. **Undoability Checking.** A major portion of the problem with partly irreversible operations is that their effects are not necessarily known; another portion is that the significance of changes in properties is highly context-specific. We propose a novel method to check the undoability of operations, based on a formal model from AI planning (see e.g., ref. [30]). Given the requirements on which properties need to be restored after the combination of forward and undo workflows, as well as which operations are to be executed, the proposed method determines if these operations can be undone.

3. **Generation of undo workflows.** If rollback is desired our system will automatically generate an undo workflow, i.e., create an appropriate sequence of operation calls from the given API, which, when executed, brings the resources back to their checkpointed state. To this end we use an AI planner [20], based on the formal model of operations mentioned above. Choosing a sequence of operations is a search in the space of possible solutions; highly optimized heuristics solve common cases of this computationally hard problem in reasonable time.

4. **Handling failures on rollback.** We use a particular AI planner [20] that produces plans which can handle failures by including suitable "backup plans", where such plans exist.

This paper makes the following contributions. We provide undoability support for API-controlled systems, such as cloud management, by checking the undoability of relevant operations. If accomplishing an undo is not always feasible, we can identify the specific operation and specific circumstances under which that operation cannot be undone. If accomplishing an undo is feasible, the undo system from our earlier work[1] can provide an undo workflow when desired. We further developed two prototypes: one is the undoability checker and one is an undo system for Amazon Web Services (AWS)[2] management operations. Based on these prototypes we evaluated

[1] A previous paper [31] was published at the 2012 USENIX HotDep workshop, focused on the undo system. We summarize some of these results here for completeness, but focus on the new aspects: undoability checking, its implications, and new experiments.

[2] `http://aws.amazon.com`, accessed 30/4/2013

our approach in depth. The website of the work is `http://undo.research.nicta.com.au`.

The remainder of the paper is structured as follows. In Section 2 we describe motivating examples. Section 3 gives an overview of the proposed system. Section 4 discusses the details of the domain model for AWS used in AI planning techniques. Section 5 presents the undoability checking approach, and Section 6 the undo system. In Section 7 we discuss implications for system operations in practice. In Section 8 we evaluate the feasibility of our approach. Section 9 connects and contrasts our work with related research. Section 10 concludes the paper and suggests directions for further study. A technical report contains formal details of the undoability checking omitted here, and is available through the website.

2 Motivating Examples

To showcase the problems addressed in this paper, we provide four concrete system administration scenarios next. These scenarios are mainly taken from our day-to-day system operations using public cloud resources within Yuruware[3], a NICTA spin-out offering a backup and disaster recovery solution for cloud-based systems. Some of them are also inspired by products for automating data center operations such as VMware vCenter Orchestrator[4] and Netflix Asgard[5]. In the evaluation (Section 8.3), we revisit these scenarios and discuss the findings from our evaluation of the specific undoability of each scenario.

Scenario 1. Adding a slave to an existing database server. When traffic to one database server grows, sooner or later it cannot provide enough capacity to handle the workload in time. A common technique to address the issue is to add slave database servers, serving only read queries, while a master server only serves write queries. This reconfiguration is typically performed during scheduled maintenance time, where the database server can be taken offline, and includes two steps: creating slave databases by cloning the database server, and introducing a proxy server to distribute queries among servers.

The first step, i.e., creating slave database servers, consists of the following activities.

1. Stop the instance operating a database server
2. Take a snapshot of the volume storing the database
3. Create copy volumes from the snapshot
4. Launch new instances with a copy volume attached
5. Configure the new instances as slave database servers
6. Start the original instance and configure it as a master

[3] `yuruware.com`

[4] `vmware.com/products/vcenter-orchestrator`

[5] `github.com/Netflix/asgard`

The second step, i.e., introducing a proxy server, consists of the following activities.

1. Launch a new instance
2. Re-allocate a virtual IP address from a master database to the new instance
3. Configure the new instance as a database proxy

Challenges. If a failure occurs during the reconfiguration, an administrator expects the undo mechanism to remove all newly created resources and restore the state of the instance operating the master database. It is trivial to remove new resources. Properties to be restored are the state (i.e., running), the allocation of the virtual IP address, the instance ID, and the amount of resources allocated. Many properties do not need to be restored such as the public/private IP address allocated to. (In a variant of this scenario, the system may rely on the private IP address of the master database server, such that it must be restorable as well.)

Scenario 2. Scaling up or down an instance. When the underlying platform – such as Amazon Web Services (AWS) – does not support dynamic resource reallocation (e.g., adding CPU cores without stopping a guest OS), an administrator first needs to stop an instance, change the resource allocation, and start the instance up again.

Challenges. Depending on the context, a variety of properties can be modified on a stopped machine, and need to be restorable. However, stopping a machine on AWS is most likely to change the private IP and the public DNS name.

Scenario 3. Upgrading the Web server layer to a new version. First an administrator creates a new load balancer, then launches instances operating a new version of an application and registers them with the new load balancer. Subsequently, the traffic is forwarded to the new load balancer by updating the respective DNS record. Once all activities completed in the old Web layer, the administrator creates a machine image from one of instances in the old Web layer as a backup and decommissions all resources comprising the old Web layer.

Challenges. In this scenario, all resources that comprise the old Web layer (i.e., the specific instances, a load balancer, their association, etc.) and the DNS record need to be restorable. In a variant of this scenario, instances may be designed to be stateless; hence being able to restore the original number of instances from the same image suffices to undo the changes to the machines.

Scenario 4. Extending the size of a disk volume. Increasing the size of a disk volume in a database server or a file server is a common administration task. Similar to the scenario shown in Scenario 1, an administrator stops an instance, creates a copy of the data volume with

larger size, and swaps the data volume of the instance with the new copy. Then he/she starts the instance again and deletes the old data volume and the snapshot.

Challenges. In this scenario, the original data volume and the association with the instance must be restorable.

3 System Overview

In this section, we summarize the functionality of the undoability checker and the undo system, before giving more details in the next sections. A high-level overview is given in Fig. 1.

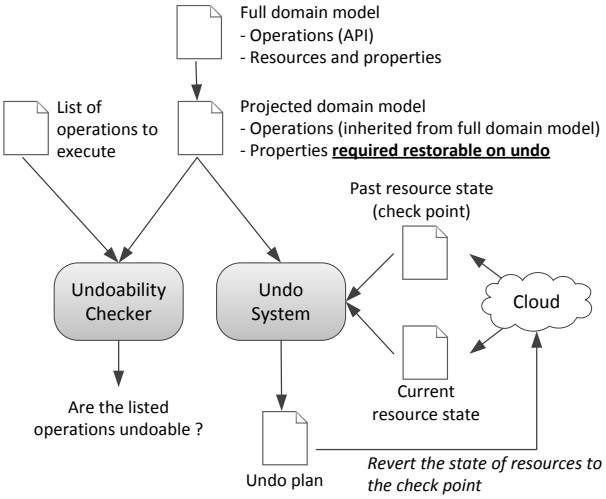

Figure 1: High-level overview of the framework

Our techniques rely on a suitable abstract model of the domain, where each operation has a precise representation of its effects on each aspect of the abstract state, as well as a precise representation of its preconditions, stating in which (abstract) states the operation can be executed. This forms the *full domain model* in Fig. 1, which can, e.g., capture operations from a management API provided by Amazon Web Services (AWS). For instance, the operation deleting a resource such as a disk volume in AWS can only be executed when the disk is available (precondition), and subsequently it is not available but deleted (effect). More details are given in Section 4.

Based on this representation, we extended techniques from AI planning (see e.g., ref. [30] for an overview) to check the undoability of a domain at design time: can each operation be undone in any situation? If no, what are the exact problems? Such problems can be solved by (i) abstracting from details or states that are irrelevant; or (ii) altering the set of operations, e.g., by replacing non-reversible forward operations such as deleting a disk volume through reversible variants, such as *pseudo-delete*. The former can be achieved by altering the domain model. The latter can be achieved by replacing the delete operation with pseudo-delete in a domain model as well as replacing the actual implementation of delete operation. In Fig. 1 the outcome of the model changes is illustrated as a *projected domain model*. A projected model is used to determine whether a given set of operations (or all operations in a domain) are undoable – whether the state of (projected) resources can be reverted from the state after executing an operation to the state before. Section 5 discusses undoability checking more detail.

When undo is desired, i.e., an administrator wants to revert the state of cloud resources from the current state to a past checkpoint, we use an extended version of the undo system from our previous work [31] to find and execute an undo plan. This is summarized in Section 6. Note that the undo system operates on the projected domain model, so that results from previous undoability checks apply.

4 Domain Model

The domain model formally captures the actions of a domain – in our case, operations defined in a cloud management API. As such, it forms the basis of the undoability checking and the undo system.

While the problem and system architecture are generic, the domain model is of course specific to the management API to be modeled. For our proof-of-concept prototype we focused on Amazon Web Services (AWS) as the domain and chose the planning domain definition language (PDDL) [25] as the planning formalism. We modeled part of the Apache CloudStack API[6] as well, namely those operations triggering state changes on machines and disk volumes. However, the AWS documentation is more detailed, and the exact effects can be observed from the single implementation of AWS. Therefore, our focus remained on AWS.

One of the most critical aspects for applying AI planning is obtaining a suitable model of the domain [22]. For the purposes of this research, we designed a domain model manually, formulated in PDDL. This model has about 1100 lines of code, and contains representations of 35 actions (see Table 1). Out of these, 18 actions are for managing AWS elastic compute cloud (EC2) resources, such as virtual machines (called *instances*) and disk volumes, and 6 actions for managing the AWS autoscaling (AS) mechanisms. These actions have been selected due to their high frequency of usage by the developers in our group. Four of the remaining actions are for system maintenance, e.g., switching a server cluster to/from maintenance modes. Those actions are not spe-

[6] `cloudstack.apache.org/docs/api/`, accessed 21/02/2013.

cific to AWS but defined for generic system administration. The final 7 actions are added undelete actions, one per resource type – which become available by replacing delete with pseudo-delete.

Resource type	API operations
Virtual machine	launch, terminate, start, stop, change VM size, undelete
Disk volume	create, delete, create-from-snapshot, attach, detach, undelete
Disk snapshot	create, delete, undelete
Virtual IP address	allocate, release, associate, disassociate, undelete
Security group	create, delete, undelete
AS group	create, delete, change-sizes, change-launch-config, undelete
AS launch config	create, delete, undelete

Table 1: AWS actions captured in the domain model – adapted from [31]

Case Study: the PDDL definition of the action to delete a disk volume is shown in Listing 1. From this example, it can be seen that parameters are typed, predicates are expressed in prefix notation, and there are certain logical operators (*not, and, oneof, ...*), also in prefix notation. The precondition requires the volume to be in state *available*, not *deleted*, and not be subject to an *unrecoverable failure*. The effect is either an *unrecoverable failure* or the volume is *deleted* and not *available* any more.

Listing 1: Action to delete a disk volume in PDDL

```
(:action Delete-Volume
 :parameters (?vol - tVolume)
 :precondition
   (and
     (volumeAvailable ?vol)
     (not (volumeDeleted ?vol))
     (not (unrecoverableFailure ?vol)))
 :effect
   (oneof
     (and
       (volumeDeleted ?vol)
       (not (volumeAvailable ?vol)))
     (unrecoverableFailure ?vol)))
```

Unrecoverable failure is a predicate we define to model the failure of an action, assuming that the affected resource cannot be brought back to a usable state using API operations. It should be noted that our planning domain model resides on a slightly higher level than the respective APIs. When a planning action is mapped to executable code, pre-defined error handlers are added as well. For example, a certain number of retries and cleanups take place if necessary. Such a pre-defined error handler, however, works only on the resource in question. If it fails to address an error, an unrecoverable failure is raised.

From the viewpoint of an AI planner the unrecoverable failure poses two challenges: *non-deterministic actions* and goal reachability. The outcome of *Delete-Volume* (success or unrecoverable failure) is observed as a non-deterministic event. In the presence of non-deterministic actions, the planner has to deal with all possible outcomes, which makes finding a solution harder than in the purely deterministic case. This requires a specific form of planning, called *planning under uncertainty* – see, e.g., Part V in [30].

Further, the question "when is a plan a solution?" arises. To cater for actions with alternative outcomes, a plan may contain multiple branches – potentially including branches from where it is impossible to reach the goal. A branch that contains no unrecoverable failure is the normal case; other branches that still reach the goal are backup branches. A plan that contains more than one branch on which the goal can be reached is called a *contingency plan*. Branches from which the goal cannot be reached indicate situations that require human intervention – e.g., if a specific resource has to be in a specific state to reach the goal, but instead raises an unrecoverable failure, no backup plan is available. This also means the action is not fully undoable (as long as unrecoverable failures are considered).

In planning under uncertainty there are two standard characterizations of plans: a *strong plan* requires *all* branches of a plan to reach the goal, whereas a *weak plan* requires *at least one* branch to reach the goal. Standard planners that can deal with uncertainty are designed to find plans satisfying either of them; however, neither is suitable in our domain. It is highly likely that no strong plan can be found: many of the actions can return an unrecoverable failure, and many of possible branches cannot reach the goal. Weak plans have the disadvantage that only the "happy path" is found: a plan that allows reaching the goal only if nothing goes wrong. When finding a weak plan, a planner does not produce a contingency plan, which we deem insufficient.

In prior work [20], a different notion of a weak plan was introduced: the goal should be reached *whenever it is possible*. This is desired in the setting given here, as it will produce as many branches (i.e., a contingency plan) as possible that still reach the goal, given such alternative branches exist. For finding plans with these solution semantics, a highly efficient standard planner, called *FF* [19], was extended in [20].

There are three discrepancies between the standard AI planning and our use of it in the undo system and the undoability checker, which require attention:

1. In the undo system, when new resources are created after a checkpoint, the resources exist in the initial

state (i.e., the state captured when a rollback is issued) but not in the goal state (i.e., the state when a checkpoint is issued). Unless treated, the AI planner simply leaves these *excess resources* intact: since they are not included in the goal state, they are irrelevant to the planner. However, to undo all changes, excess resources should be deleted. To discover plans that achieve this, we perform a step of preprocessing before the actual planning: the goal state is compared with the initial state in order to find excess resources; the goal state is then amended to explicitly declare that these excess resources should end up in the "deleted" state.

2. In the AI planning variant we employ, new resources cannot simply be created out of nowhere. Instead, we model this case through a unary predicate called *not-yet-created*. In the undoability checker, we consider *not-yet-created* to be equivalent to *deleted*.[7] Thus, we need to replace any occurrence of *not-yet-created* in an undoability checking goal with *not-yet-created OR deleted*. This allows us to undo effects of actions that create new objects, by simply deleting them.

3. The equals predicate "=" is used in PDDL to mark the equivalence of two constants. In our domain model, we use it in the preconditions of several actions. In the undoability checker, this would cause problems, since the checker derives initial and goal states for planning problems from actions' preconditions and effects and instantiates them with constants – which can cause contradictions. We circumvent the problem by applying the meaning of the equals predicate in planning tasks created by the undoability checker, and filtering out any contradictory states.

Using these special treatments in the undo system and the undoability checker, we can use a standard PDDL domain model for both purposes – so long as it follows our naming convention for *deleted* and *not-yet-created* predicates.

5 Undoability Checking

As argued in the introduction and the motivating examples, it is not *a priori* clear if all operations can be rolled back under any circumstances. In order to provide the user with confidence, we devised an undoability checker, which uses the domain model described in the previous section. We herein summarize our undoability checking approach – since the problem is highly non-trivial, a separate technical report is available, see Section 1, which provides a full formal treatment of the matter.

[7]This point is related to the above, but different in that the checkpointed state does not contain *not-yet-created* predicates.

5.1 Undoability Checking Overview

Fig. 2 provides an overview of the undoability checker. It involves two separate roles: a tool provider and a user. The tool provider defines the full domain model that formally captures the inputs and the effects of operations available in a cloud platform, such as Amazon Web Services (AWS). The user of the undoability checker is assumed to have enough knowledge to operate systems on a cloud platform; however, he/she does not need to define or know the details of the domain model.

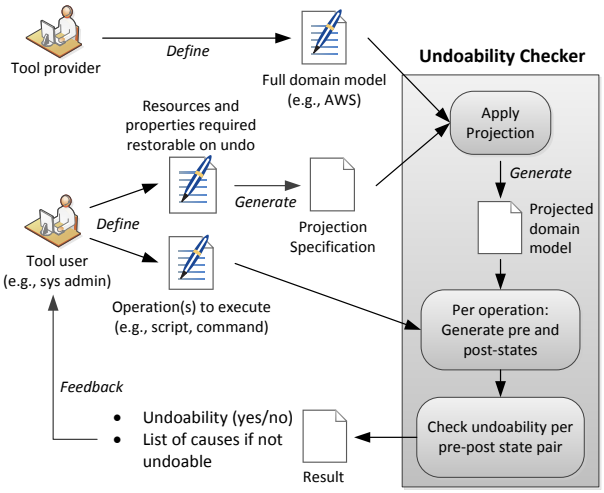

Figure 2: Overview of the undoability checker

The user defines two inputs, both are optional. We refer to these inputs as the *context*. The first one is a list of operations the user plans to execute. It can be a script defining a sequence of operations for a certain system administration task, or a single operation. The other input is a list of resources and properties the user wants to restore upon any failures. It is derived from the context of the system administration as discussed in Section 2. For example, a user may need the IP address of an instance to be restored upon a failure, but not its exact identity.

Given those inputs, the undoability checker examines whether all listed operations are undoable, i.e., the state of all listed resources and properties can be restored after executing these operations. If no operations are provided by the user, the undoability checker tests *all* operations of a domain. If no resources and properties are given by the user, all resources with all their properties from the full domain model are considered. If the output is positive, the user can be sure that the system state can be restored, as long as he/she executes operations listed in the input, or for the whole domain if no operations were specified. If the output is negative, the checker reports the causes – e.g., that a certain property cannot be restored if the user executes a specific operation. The user can consider us-

ing alternative operations or accept a weaker undoability guarantee by giving up the undoability of the property. The tool can of course be used iteratively, until lists of resources, properties, and operations are found which are fully undoable.

5.2 Domain Model Projection

Undoability of the full domain indicates that all types of resources and the properties can be restored after executing any available operations in a cloud platform. Depending on system administration tasks, the part of the domain that is required to be undoable might be significantly smaller than the full domain model, however. Therefore, before the actual undoability check is performed, the undoability checker considers the context provided by the user, if any, and extracts a subset of the full domain. To achieve this, we introduce the concept of a *projection of a domain model*.

From the list of resources and properties in the context, the undoability checker creates a projected domain that only captures resources and their properties involved in a provided context. This is done by creating a projection specification in a specific format from the user input, and applying the specification to the full domain. If no list of relevant resources and properties is provided by the user, the full domain is used instead of a projected one. Projections can also include a role-player concept, such that e.g., virtual machines can be said to play a role (e.g., Web server), where their actual identity is less relevant. A projected domain model is itself a valid domain model in PDDL, so undoability checking can proceed as per below, with or without projection.

In terms of our running example of AWS, without projection most actions in the AWS domain model would not be considered undoable, due to the unrecoverable errors discussed in Section 4. If an unrecoverable error occurs, one cannot reverse back to the previous state of that resource. Unless using a different resource to fulfill the same purpose (e.g., a new machine instead of a crashed one, from the same machine image) is acceptable, the unrecoverable error is just that: unrecoverable. More interesting is the question: *assuming no unrecoverable error occurs, do we have guaranteed undoability?* This question can be addressed by leveraging domain projection: the user can define a context that excludes unrecoverable errors.

Even so, most real-world domains do not require that each and every aspect can be restored. For example, a virtual machine in AWS has over 20 attributes such as the resource identity assigned by the provider (e.g., ID `i-d21fa486`), public and private DNS names assigned by the provider (e.g., ec2-xxx.compute.amazon.com), instance type indicating the amount of computing resources allocated (e.g., `m1.small` roughly equals to 1.2GHz CPU and 1.7GB RAM), the identity of the machine image used to launch the virtual machine (e.g., `ami-a6a7e7f4`) and launch timestamp. Although some of those properties are easy to be restored, others are not. For example, to obtain an instance whose machine image identity is the same as that of a terminated one, an administrator simply launches a new instance from the machine image used before. However, there is no way to obtain the same resource identity or public DNS name once an instance is terminated, since these are assigned dynamically by the cloud provider.

5.3 Undoability Checking Algorithm

From a (projected) domain, the checker considers each relevant action individually (either from the list provided by the user, or all actions in the domain). For each relevant action, the checker tries to find situations in which executing the action *may* result in a state from which the system *cannot* get back to the state before the action was executed. If no such situation exists, we call the action *fully undoable*. If each action can be reversed individually, then any sequence of actions can be reversed – this is related to the Sagas approach [14], which is discussed in Section 9. While such a sequence may be highly suboptimal, we only use it to prove undoability on a theoretical level – when the actual situation occurs, our undo system finds a shorter undo workflow.

To check undoability of an action, we need to ascertain undoability for any state in which the action could be executed – the *pre-states* of the action. In general, this is an infinite set. We derive a *sufficient* set of pre-states for undoability checking – i.e., a set that captures any distinct situation from the pre-states – by considering:

1. how many constants of which type may be used by any of the actions in the domain;
2. analyzing which actions may contribute to undoing the one currently being checked;
3. deriving any combination of the logical statements from the combination of the above sets.

Based on the sufficient set of pre-states, we compute the set of possible post-states – the outcomes of applying the action being checked to each of the pre-states, where non-determinism means a pre-state can result in multiple post-states. For each post-state, we then check if the corresponding pre-state is reachable, by formulating a respective planning task and feeding it into the AI planner [20]. The usual *planning problem* [30] is the following: given formal descriptions of the *initial state* of the world, the desired *goal state*, and a set of available *actions*, find a sequence of actions that leads from the initial to the

goal state. In the case of the undoability checker, the question is: can the pre-state be reached from the post-state? Thus, the post-state is the initial state of the planning problem, and the corresponding pre-state forms the goal state. If there is a plan for each pre-post-state pair, then full undoability (in the given context) is shown; else, the problematic cases are reported back to the tool user.

5.4 Checker Usage Models

There are two ways to deploy the undoability checking: offline and online. The offline deployment model is used for checking a whole domain, or a specific script / workflow implementing a system administration task. When used by administrators, this deployment model assumes that the script / workflow will be executed in future, e.g., during scheduled downtime on the following weekend, and the tool user prepares for it. In this scenario administrators usually spend enough time on developing a plan in order to get maximum leverage out of the scheduled maintenance period. Given a list of operations invoked in a script[8], the undoability checker examines if they are reversible. The checker shows a list of operations not reversible if exists (Figure 2.) The user is expected to use the undoability checker iteratively to improve the quality of scripts to execute. For example, removing all irreversible operations from the script, placing irreversible operations as late as possible to make the workflow reversible until calling one of the irreversible operations, or altering the set of attributes to be undoable such as using a virtual IP address instead of an internal IP address.

The alternative deployment model, i.e., online deployment model, is used for checking the undoability of each operation an administrator is about to execute. This deployment model provides a *safety net* to an administrator directly executing operations on a console. In this scenario, each operation is executed through a wrapper (or a proxy) rather than executed directly on cloud resources, as discussed in the next section. Before an operation is executed on cloud resources, the wrapper performs an undoability check and sends a warning to the administrator if the operation is not undoable.

The two models are built on different assumptions. The offline model assumes that the concrete state of resources such as IP addresses cannot be truthfully observed, since the analysis is performed before a script is actually executed. Therefore, it determines the undoability of all operations against *all possible* pre-states. This mode often results in very strong undoability guarantees – arguably stronger than needed, since the undoability checker examines pre-states that may never occur in the actual system. Say, for example, a machine had a fictitious attribute 'unmodified since start', which would be true initially after starting a machine, but false after the first change was applied. In general, any action modifying the machine, e.g., changing its size, would be undoable – and detected as such by the offline check. However, the online check might encounter a state where this attribute was already false – in this particular case, modifications to the machine would be undoable.

To address the limitation of the offline model, the online model assumes that it can obtain the status of resources by making calls to the API a cloud platform provides. If an operation is not known to be fully undoable in a given administration context (by looking up the result that the offline check provides), it senses the state of the resources and forwards this information to the undoability checker. The checker then takes this state as the only pre-state, and checks if all possible outcomes of executing the operation in this *specific* pre-state are undoable. Therefore, operations identified as not undoable by the offline check could be identified as undoable by the online check depending on the status of resources.

Although the online model is very targeted in terms of the performed undoability checks, it may not be practical depending on the responsiveness of APIs. It is not uncommon for a scan of the resources in public cloud platform to take longer than 30 seconds depending on the network latency, while it can be less that 1 second on an on-premise virtualized environment. Depending on the user's preferences, the slow responsiveness may be unacceptable.

6 Undo System Design

The main component of the undo system we propose concerns automatically finding a sequence of operations for realizing rollback. An earlier version of the undo system was described in [31], which we summarize here.

6.1 Overview of the Undo System

Fig. 3 shows the overview of the undo system, which is partially positioned between the user / operational script and the cloud management API. An administrator or an operational script first triggers a *checkpoint* to be taken[9]. Our undo system gathers relevant information, i.e., state of cloud resources and their relationship, at that time. After a checkpoint, the system starts to offer rollback to the checkpoint or committing the changes. Before either of these command is called, the system transparently replaces certain non-reversible operations, e.g., deleting a

[8]Our current implementation does not parse a script, e.g., a bash script, directly to extract operations. We assume the user provides a list of operations used in a script to the checker.

[9]While the resources can form a vastly distributed system, their state information is obtained from a single source of truth: AWS's API.

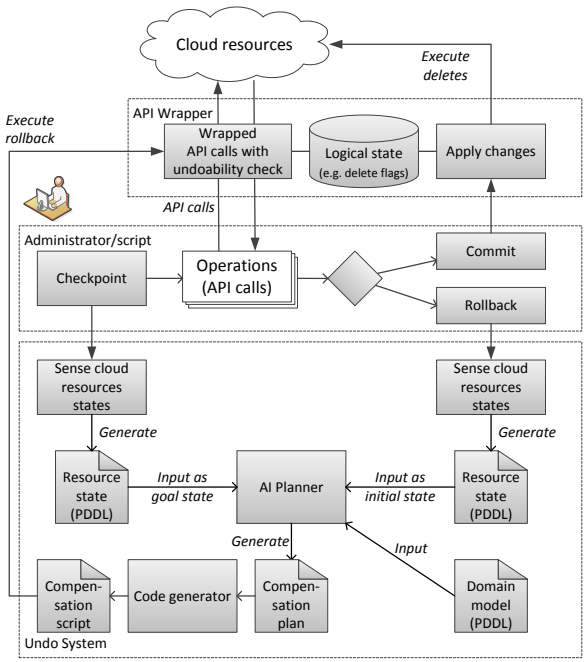

Figure 3: Overview of rollback via planning – adapted from [31]

resource, with reversible ones, like pseudo-delete. Further, it checks the undoability of each operation to be executed, as discussed in Section 5.4. When a *commit* is issued, the non-reversible changes are applied to the cloud resources – thus, rollback is not offered anymore. When a *rollback* is issued, the system again gathers the state of cloud resources and feeds the pair of state information to an AI planner to construct an undo sequence of operations, as discussed in Section 6.3.

6.2 API Wrapper

The undo system offers a wrapper for cloud APIs. After a checkpoint, when the user asks to delete a resource, the wrapper sets a *delete flag* (or *ghost flag*), indicating that the resource is logically deleted. Note that this requires altering *each* subsequent API call to the resource through the wrapper: queries to list provisioned resources need to be altered in their responses by filtering flagged resources out; changes to flagged resources need to be caught and answered, e.g., by returning a "not found" error message.

When a delete operation is to be reversed – triggered by a respective *undelete* operation – the wrapper simply removes the delete flag. When the user requests a rollback, the AI planner includes an undelete call on that resource. Only when a commit is issued, all resources with a delete flag are physically deleted.

The wrapper is also the connection point to the undoability checker in the online deployment model (Section 5.4). If an action is not undoable, it warns the user, with an option to cancel.

6.3 Rollback via AI Planning

For rollback, the goal is to return to the state of the system when a checkpoint was issued. Our undo system finds a sequence of undo actions by using the AI planner from [20]. In undo planning, the initial state is the state of the system when rollback was issued. The desired goal state is the system state captured in a checkpoint. The available actions are the API operations offered by a cloud provider, captured in the domain model.

In the undo system (Fig. 3), the planning problem (initial and goal states, set of available actions) is the input to the planner. Its output is a workflow in an abstract notation, stating which action to call, in which order, and for which resources. The abstract workflow is forwarded to a code generator, which transforms it into executable code. The final step is to execute the workflow against the wrapped API operations.

7 Implications for Practice

In this section, we discuss how the various parts of the systems can be used by practitioners in the future. The easiest case is when a domain is fully undoable. The implication is that the user can do anything, and have the confidence that the undo system can find a path back to any checkpoint. Similarly, if a script consists only of fully undoable actions, employing it can be done with high confidence.

If properties need to be excluded through projection to achieve undoability, the administrator gains explicit knowledge of the boundaries of undoability. This knowledge can be used for improving the dependability of administration tasks through awareness, or to inform changes to scripts or procedures. If no alternative action exists, a script / workflow can potentially be *re-ordered*, to schedule non-undoable actions as late as possible. The rationale for doing so is that executing an irreversible action can be seen as a form of commit: if anything goes wrong before, it can still be undone; afterwards this is not the case. As many reversible operations as possible should therefore be done before the irreversible one(s).

Another way to leverage the results is to provide a *safe mode* of an API, which includes the subset of operations that are fully undoable. This subset can e.g., be used in day-to-day operations, while using reversible operations might be reserved for users with higher privileges. Untrusted or less experienced administrators may only be allowed to operate within the boundaries of the safe

mode, so that any mistakes or malicious behavior can always be undone.

Finally, the undo system's checkpointing / commit / rollback commands can be made part of scripts or workflows. For instance, a script may always set a checkpoint before execution. If any of a set of known errors arises, rollback is automatically triggered. If a final set of automated checks succeeds, the changes can be committed before exiting the script.

8 Implementation and Evaluation

In this section, we describe our prototypical implementations and the experiments we conducted to evaluate our approach. The latter include applying the undoability checker to the AWS domain model, assessing the undoability of the scenarios from Section 2, as well as a summary of performance experiments from our earlier work [31].

8.1 Implementation

Both the undo system and the undoability checker have been implemented as prototypes. The undo system has been rolled out for internal beta-testing and used for feasibility and performance experiments. Both prototypes have been implemented in Java, making command line calls to the version of the FF planner used in our work [20]. The undo system further includes bash scripts for use as a command line tool, replacing the AWS implementations of the operations in Table 1 and providing additional operations for checkpointing, undelete, rollback, and commit.

The undoability checker has around 17,500 lines of Java code, including an extended version of an open source parser, lexer, and PDDL data model[10]. The undo system has roughly 7,300 lines of Java code, the FF planner as per above around 17,000 lines of code in C and lexer / parser definitions.

The main limitations of the current prototypes are the following.

- The undoability checker does not have a component for generating projections as yet – desired projections had to be applied by manually creating variants of the Amazon Web Services (AWS) domain model.
- The API wrapper of the undo system is not integrated with the undoability checker, although the latter has an interface that can be adapted to implement this integration (checking undoability of single actions in specific states).
- Neither tool has been optimized for performance yet, and both are still in the state of (well-tested) prototypes. Particularly for the undo system, we plan to change this in the near-term future.

[10] `http://www.zeynsaigol.com/software/graphplanner.html`, accessed 8/4/2013

8.2 Undoability of a Full Domain

Using the implementation of the undoability checker, we performed several iterations of checking the AWS domain model. As expected, the domain model including *unrecoverable errors* is not undoable. After implementing a projection to remove *unrecoverable errors* as well as several changing parameters such as internal IP address, undoability is given for most actions. Note, that this domain model includes our AWS extension of undelete actions.

In comparison to the earlier version of our domain model, as described in [31], it should be noted that we added more details in all but a few actions. This primarily concerns statements in preconditions and effects that were previously seen as not required. For instance, deleting a volume (see Listing 1) now has a precondition of *not volume deleted*, which was previously seen as implicitly following from *volume available*. We further split up certain actions, without implications for practice – e.g., instead of changing all auto-scaling group targets (min, max, desired) at once, for the purposes of undoability checking we split this action into three separate ones. This reduces the number of separate planning tasks significantly. Due to such changes, the domain model grew from 800 to 1100 lines of code.

Our current results of checking undoability for the whole domain model (Table 1) show 34 out of the 35 actions to be fully undoable, given the above projection. For this purpose, the prototype ran for 11s inside a VirtualBox VM on two (hyperthreaded) cores from an Intel i7-2600 quadcore CPU @ 3.4GHz, with 4GB RAM available to it. During that time, it called the planner with 1330 separate planning problems – i.e., pre-post-state pairs as per Section 5.3.

The action for which undoability cannot currently be shown is *create auto-scaling group*. This action creates an AS group with the respective three sizes (min, max, desired). The corresponding *delete* action has only the precondition of *max = 0* (which in practice implies *min = 0 = desired*). However, this implication is not expressed to the planner as yet. If it were, then the number of possible instantiations in pre/post-state pairs becomes too large to be handled by the current implementation. We believe that this issue can be solved in the future, and undoability can be fully shown, when applying the above-mentioned projection.

8.3 Undoability of Administration Tasks

We next present our evaluation of the undoability of concrete system administration scenarios. As described in Section 5.4, we assume that the tool user provides a list of operations to execute in system administration task. The undoability checker examines whether the list contains irreversible operations given a set of properties to be reversible. This section discusses the results that the undoability checker produces for the four scenarios introduced in Section 2.

Findings for Scenario 1. Adding a slave to an existing database server. All steps can all be undone through the AWS management API. In particular, the original configuration of the database server can be restored by restoring the volume from a snapshot taken – however, while this step is on top of our future work list, it is currently not implemented in our undo system. The variant relying on the private IP address of the database server can be undone, so long as *stop machine* is replaced with *pseudo-stop*, and assuming that reconfiguring the database server can be done on a running machine. Note that replacing *stop* with *pseudo-stop* has the side-effect that no changes such as changing the machine size can be done: such changes require the machine to be actually stopped, in which case the private IP cannot be restored.

Findings for Scenario 2 Scaling up or down an instance. In our current model, we support changing the instance size. Other aspects could be modeled analogously: from studying the AWS documentation of `ec2-modify-instance-attribute`[11], we believe any aspect that can be changed can also be undone. As for the implicit changes to private IP / public DNS name, the undo system could be configured to warn the user.

Findings for Scenario 3. Upgrading the Web server layer to a new version. DNS records are outside the scope of our domain, and need to be undone manually. Load balancers and the creation/deletion of machine images are currently not captured, but could be in the future. The remaining actions are all shown to be undoable in our domain model.

Findings for Scenario 4. Extending the size of a disk volume. This scenario uses solely operations shown to be undoable. Since the machine is stopped by the administrator at any rate, its private IP will change, so that undoability is not affected. The warnings mentioned in Scenario 3 can be applied here as well.

[11] `http://docs.aws.amazon.com/AWSEC2/latest/CommandLineReference/ApiReference-cmd-ModifyInstanceAttribute.html`, last accessed 29/4/2013

8.4 Undo System Performance

We summarize the performance results from [31], and discuss performance changes resulting from the changes to the domain explained in Section 8.2. For this experiment, we assembled over 70 undo planning tasks. When tasked with finding the solution to undo problems requiring up to 20 undo actions, the execution time of the planner was often close to or below the resolution of measurement (10 ms) – i.e., all such plans were found in less than 0.01 seconds. Even plans with more than 60 actions were found in less than 2 seconds, using the domain model from [31]. However, after altering the domain model, planning times changed as follows. Plans with less than 30 steps were still found in less than 10ms. More complex plans took 8-15 times longer than with the original domain model. To put this into perspective: scripts with over 20 steps are unusual in our groups' experience. In comparison to the execution time of scripts on AWS – for instance, the average execution time over 10 runs of an 8-step undo plan of AWS operations was 145 seconds – the cost for planning is marginal in many situations.

9 Related Work

Our undo approach is a checkpoint-based rollback method [13, 29]. Alternative rollback methods are log-based [13, 15] or using "shadow pages" in databases [15]. In much research, checkpoints store a relevant part of the state on memory and/or disk, and for rollback it suffices to copy the saved information back into. In contrast, rollback in our setting means achieving that the "physical state" of a set of virtual resources matches the state stored in a checkpoint – i.e., achieving rollback requires executing operations, not only copying information.

Thus, our setting is similar to long-running transactions in workflows or business processes. [16] gives an overview of approaches for failure and cancellation mechanisms for long-running transactions in the context of business process modeling languages, where typical mechanisms are flavors or combinations out of the following: (i) Sagas-style reverse-order execution of compensators [14]; (ii) short-running, ACID-style transactions; and (iii) exception handling similar to programming languages like C++ and Java. As for (i), on cloud platforms, compensating operations may not be available. Even when an operation is an inverse for another, there may be non-obvious constraints and side-effects, so that executing the apparently compensating operations in reverse chronological order would not restore the previous system state properly; a different order, or even different operations, might be more suitable [16]. This argument supports our approach to use an AI planner for

coming up with a targeted undo plan. ACID cannot be achieved so (ii) is unsuitable in our setting: the state of the cloud resources is externalized instantaneously, so consistency and isolation are impossible to retain here. As for (iii), hand-coded exception handling for whole workflows can be implemented, but is error-prone; per-operation exception handling is unsuitable, since the behavior of an action is non-deterministic, context-specific, and the target state is not static. In summary, the traditional approaches are not a good fit for the problem at hand.

Besides AI Planning, other techniques were used to achieve dependability in distributed systems management. [21] uses POMDPs (partially-observable Markov decision processes) for autonomic fault recovery. An architecture-based approach to self-repair is presented in [4]. [26] is a self-healing system using case-based reasoning and learning. The focus in these works is self-repair or self-adaptation, not undo for rollback.

AI Planning has been used several times for system configuration, e.g., [1, 2, 7, 9, 10, 11, 23], and for cloud configuration, e.g., [17, 18, 24]. Some works aim at reaching a user-declared goal, e.g., [1, 7, 17, 18, 23], whereas others target failure recovery, e.g., [2, 9, 10, 24]. The goal in the latter case is to bring the system back into an operational state after some failure occurred.The closest related work are [17, 18, 24]. In [18], planning is applied in a straight-forward fashion to the problem of reaching a user-specified goal.The work is well integrated with cloud management tools, using Facter, Puppet and ControlTier – all of which experience some level of popularity among administrators nowadays. [24] applies hierarchical task network (HTN) planning on the PaaS level for fault recovery. [17] uses HTN planning to achieve configuration changes for systems modeled in the common information model (CIM) standard.

As for the undoability checking, Burgess and Couch [6] have shown that well-defined rollback is impossible to achieve in realistic systems, mostly due to non-determinism and openness. This result concurs with our argumentation for the need of projection: only fully closed and deterministic systems have a chance of being fully undoable without projection. For any other case, our approach first can point out what prevents undoability, and iteratively those issues can be projected away, so as to understand under which assumptions undoability is given. Our undo tool takes non-determinism of actions into account directly.

Our undoability checking extends a model of reversible actions from Eiter et al. [12]. Our work differs from [12] in two major ways: we consider undoability of full and projected domains, and we assume the state to revert to is known. This changes the underlying formalism (described in the technical report – see Section 1), and hence most formal results built on top of it.

10 Conclusions

In this paper we describe our support for undoability, building on two approaches: (i) an undoability checker analyses to what degree operations can be rolled back; and (ii) an undo system that automatically generates rollback workflows, when the need arises. The latter can essentially provide transactional atomicity over API-controlled environments, like cloud management. We evaluated the approaches through the prototypes we developed, with performance experiments and by applying them to real-world examples of cloud management APIs and best practices. An intrinsic limitation of our approaches is that they operate on a manually created model of the available operations. While we took care to assess that the model truthfully captures the implementation, this cannot be formally guaranteed without access to AWS's API implementation, deployment, and operation of the cloud platform. Further, the model is not going to be aware of future changes to the API and its implementation.

The prototype for the undo system is in the process of being developed into a more mature tool, and we may make it available for public use through our website – see Section 1. It will likely feature different levels of undoability from which the users can choose, as established through the undoability checker.

In future work, we plan to extend the undoability checker with an approach to find projections leading to full undoability automatically, such that the removed properties are minimal. For the undo system, we plan an extension to capture the internal state of resources when checkpointing and to restore the internal state on rollback. For example, the content of a disk volume can be captured by taking a snapshot. Finally, the undo system will be extended to handle multiple checkpoints and manage them by their names, where administrators can then choose to rollback to checkpoint P_1 or commit all changes up to checkpoint P_2.

Acknowledgements

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program. This work is partially supported by the research grant provided by Amazon Web Services in Education [12]. We thank Gernot Heiser for his helpful remarks on this paper.

[12] `http://aws.amazon.com/grants/`

References

[1] ARSHAD, N., HEIMBIGNER, D., AND WOLF, A. L. Deployment and dynamic reconfiguration planning for distributed software systems. In *ICTAI'03: Proc. of the 15th IEEE Intl Conf. on Tools with Artificial Intelligence* (2003), IEEE Press, p. 3946.

[2] ARSHAD, N., HEIMBIGNER, D., AND WOLF, A. L. A planning based approach to failure recovery in distributed systems. In *WOSS'04: 1st ACM SIGSOFT Workshop on Self-Managed Systems* (2004), pp. 8–12.

[3] AVIZIENIS, A., LAPRIE, J.-C., RANDELL, B., AND LANDWEHR, C. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing 1*, 1 (2004), 11–33.

[4] BOYER, F., DE PALMA, N., GRUBER, O., SICARD, S., AND STEFANI, J.-B. A self-repair architecture for cluster systems. In *Architecting Dependable Systems VI*, vol. 5835 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2009, pp. 124–147.

[5] BROWN, A., AND PATTERSON, D. Rewind, repair, replay: three R's to dependability. In *10th ACM SIGOPS European workshop* (2002), pp. 70–77.

[6] BURGESS, M., AND COUCH, A. L. On system rollback and totalized fields: An algebraic approach to system change. *Journal of Logic and Algebraic Programming 80*, 8 (2011), 427–443.

[7] COLES, A. J., COLES, A. I., AND GILMORE, S. Configuring service-oriented systems using PEPA and AI planning. In *Proceedings of the 8th Workshop on Process Algebra and Stochastically Timed Activities (PASTA 2009)* (August 2009).

[8] DA ROLD, C., JESTER, R., MAURER, W., CHAMBERLIN, T., ENG, T. C., AND PETRI, G. Data center services: Regional differences in the move toward the cloud. Tech. Rep. G00226699, Gartner Research, 29 February 2012.

[9] DA SILVA, C. E., AND DE LEMOS, R. A framework for automatic generation of processes for self-adaptive software systems. *Informatica 35* (2011), 3–13. Publisher: Slovenian Society Informatika.

[10] DALPIAZ, F., GIORGINI, P., AND MYLOPOULOS, J. Adaptive socio-technical systems: a requirements-driven approach. *Requirements Engineering* (2012). Springer, to appear.

[11] DRABBLE, B., DALTON, J., AND TATE, A. Repairing plans on-the-fly. In *Proc. of the NASA Workshop on Planning and Scheduling for Space* (1997).

[12] EITER, T., ERDEM, E., AND FABER, W. Undoing the effects of action sequences. *Journal of Applied Logic 6*, 3 (2008), 380–415.

[13] ELNOZAHY, E. N. M., ALVISI, L., WANG, Y.-M., AND JOHNSON, D. B. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv. 34*, 3 (Sept. 2002), 375–408.

[14] GARCIA-MOLINA, H., AND SALEM, K. Sagas. In *SIGMOD'87: Proc. Intl. Conf. on Management Of Data* (1987), ACM, pp. 249–259.

[15] GRAEFE, G. A survey of b-tree logging and recovery techniques. *ACM Trans. Database Syst. 37*, 1 (Mar. 2012), 1:1–1:35.

[16] GREENFIELD, P., FEKETE, A., JANG, J., AND KUO, D. Compensation is not enough. *Enterprise Distributed Object Computing Conference, IEEE International 0* (2003), 232.

[17] HAGEN, S., AND KEMPER, A. Model-based planning for state-related changes to infrastructure and software as a service instances in large data centers. In *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing* (Washington, DC, USA, 2010), CLOUD '10, IEEE Computer Society, pp. 11–18.

[18] HERRY, H., ANDERSON, P., AND WICKLER, G. Automated planning for configuration changes. In *LISA'11: Large Installation System Administration Conference* (2011).

[19] HOFFMANN, J., AND NEBEL, B. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research (JAIR) 14* (2001), 253–302.

[20] HOFFMANN, J., WEBER, I., AND KRAFT, F. M. SAP speaks PDDL: Exploiting a software-engineering model for planning in business process management. *Journal of Artificial Intelligence Research (JAIR) 44* (2012), 587–632.

[21] JOSHI, K. R., SANDERS, W. H., HILTUNEN, M. A., AND SCHLICHTING, R. D. Automatic model-driven recovery in distributed systems. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems* (Washington, DC, USA, 2005), SRDS '05, IEEE Computer Society, pp. 25–38.

[22] KAMBHAMPATI, S. Model-lite planning for the web age masses: The challenges of planning with incomplete and evolving domain models. In *AAAI'07: 22nd Conference on Artificial Intelligence* (2007).

[23] LEVANTI, K., AND RANGANATHAN, A. Planning-based configuration and management of distributed systems. In *Proceedings of the 11th IFIP/IEEE international conference on Symposium on Integrated Network Management* (Piscataway, NJ, USA, 2009), IM'09, IEEE Press, pp. 65–72.

[24] LIU, F., DANCIU, V., AND KERESTEY, P. A framework for automated fault recovery planning in large-scale virtualized infrastructures. In *MACE 2010, LNCS 6473* (2010), pp. 113–123.

[25] MCDERMOTT, D., ET AL. *The PDDL Planning Domain Definition Language*. The AIPS-98 Planning Competition Comitee, 1998.

[26] MONTANI, S., AND ANGLANO, C. Achieving self-healing in service delivery software systems by means of case-based reasoning. *Applied Intelligence 28* (2008), 139–152.

[27] OPPENHEIMER, D., GANAPATHI, A., AND PATTERSON, D. A. Why do internet services fail, and what can be done about it? In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4* (Berkeley, CA, USA, 2003), USITS'03, USENIX Association, pp. 1–1.

[28] PATTERSON, D., AND BROWN, A. Embracing failure: A case for recovery-oriented computing (ROC). In *HPTS'01: High Performance Transaction Processing Symposium* (2001).

[29] RANDELL, B. System structure for software fault tolerance. *IEEE Transactions On Software Engineering 1*, 2 (1975).

[30] TRAVERSO, P., GHALLAB, M., AND NAU, D., Eds. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2005.

[31] WEBER, I., WADA, H., FEKETE, A., LIU, A., AND BASS, L. Automatic undo for cloud management via AI planning. In *HotDep'12: Proceedings of the Workshop on Hot Topics in System Dependability* (Oct. 2012).