Model-based Planning for State-related Changes to Infrastructure and Software as a Service Instances in Large Data Centers

Sebastian Hagen
Department of Computer Science
Technische Universität München
85748 Garching, Germany
hagen@in.tum.de

Alfons Kemper
Department of Computer Science
Technische Universität München
85748 Garching, Germany
kemper@in.tum.de

***Abstract*—To deliver 3-tier applications as a Service in the Cloud state-related constraints crossing Infrastructure- and Software as a Service boundaries need to be managed. By automating the lifecycle of applications like databases, load balancers, and web application servers rich SaaS business services can be provided in the Cloud. We propose an object oriented planning approach based on state constraints to plan for changes of SaaS and IaaS components in the Cloud. We evaluate techniques for fast storing and restoring of large object oriented Configuration Management Databases and show that enforcing constraints in a procedural instead of a declarative way offers huge performance improvements. The advantages of our approach lie within the tight integration of the planning algorithm with object oriented models frequently used for Configuration Management Databases. In addition to that, the algorithm scales to a large number of nodes and preserves its runtime even for large, heavily loaded data centers.**

***Keywords*-IT change planning, service management, application management, AI planning, state-based constraints**

I. Introduction

With the proliferation of *Cloud Computing* [1], e. g., *Infrastructure as a Service* (*IaaS*) and *Software as a Service* (*SaaS*), it becomes more and more important for cloud/service providers to migrate existing business applications to the Cloud to generate additional value for customers on top of IaaS. A typical business application very often consists of a three-tier architecture, comprising a database (DB), several application- or web servers (WAS/Apache), and a load balancer (LB). Offering and migrating such applications to the Cloud makes sense to decrease IT costs by consolidating IT infrastructure, to offer business applications as a service, and to dynamically scale the application layer to cope with varying loads. Applications following the 3-tier architecture are for example, SAP/R3 and wiki systems, such as TikiWiki.

To automatically offer these applications to customers as a service [2] in the Cloud, their lifecycle has to be automatically managed according to state-related constraints crossing SaaS and IaaS boundaries. In a previous work [3] we proposed to describe these constraints using state-transition systems:

SaaS - SaaS constraints: To start an Apache server, the database needs to be in state *running* because Apache relies on data stored in the database.

SaaS - IaaS constraints: In order to install an application in a virtual machine (VM), it needs to be in state *running* and the physical machine (PM) needs to be in state *on*.

Planning for state related changes in a cloud environment becomes very challenging due to the large amount of components comprising a data center leaving lots of possibilities to instantiate an action. Traditionally object oriented (OO) models, e. g., the *Common Information Model* (*CIM*) [4] standard, have been used to implement *Configuration Management Databases* (*CMDBs*) to describe the data center and all its hosted applications. In the presence of large data centers these models can become very big making planning extremely difficult.

Although several approaches were proposed for IT change planning [3], [5], [6], [7], [8], [9] they do not provide an adequate planning approach working over object oriented CMDBs in the presence of large data centers as imposed by Cloud Computing. For example, Trastour et al. [6] proposed a planning approach over an object oriented CMDB but did not provide adequate performance for large scale CMDBs. Others [5] used a planner over a predicate based knowledgebase (KB). However, this approach has two drawbacks: (1) The object oriented CMDB and the results of planning need to be transformed between OO models and predicates. (2) An operator familiar with the object based CMDB has to learn an unknown predicate based language to describe the knowledgebase. Our previous work [3] made use of an object oriented CMDB and a KB written in a *Domain Specific Language* (DSL) over this model. However, aiming at performance in the presence of many nodes was not a goal of this work. Furthermore, there were situations, that were in fact out of the scope of this work, were the algorithm was unable to find a solution although one existed.

In this work we propose an algorithm inspired by artificial intelligence planning techniques [10] working over object oriented models to plan according to IaaS and SaaS state constraints. We evaluate the performance of several tech-

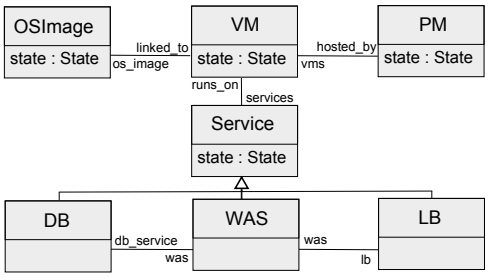

Figure 1. Object oriented CMDB model over which planning is done

niques for fast object storing and restoring to pave the way to directly use object oriented CMDBs for planning in the presence of large data centers. We apply the algorithm to a deployment case study of 3-tier applications and show that it nicely scales with the increasing size of the data center. In addition to that, we show that enforcing placement constraints in a procedural way has significant advantages over a declarative check-and-prune approach when it comes to planning speed under varying load levels in the Cloud.

Compared to previous work our approach has several advantages: (1) It scales to a large number of nodes / objects in the CMDB even in the presence of difficult placement constraints. (2) Planning is directly done over an object oriented CMDB making it easy to interface with tools and existing models, such as CIM [4]. (3) The KB can directly address properties of objects in the model. There is no need for switching between different representations as in [5]. (5) Direct interfacing with external tools can be easily achieved, e. g., to determine an optimal placement [11]. (6) The knowledgebase can be easily adapted to take different solutions for application deployment into account. For example, solving deployment using *Virtual Appliances* [12], [13] or using a finer grained domain comprising installation and configuration of applications [14].

The remainder of this paper is organized as follows: Section II introduces our notion of models and state-transition systems. The algorithm is introduced in Section III. After that, we propose and evaluate different strategies for fast object storing and restoring in Section IV. In Section V we propose a heuristic to optimize the runtime for large data centers. Section VI evaluates the influence of declarative and procedural constraints on the runtime of the algorithm. Finally, we relate our work to others in Section VII and conclude in Section VIII.

II. Modelling the Cloud

This section introduces the basic concepts our solution is built on. Object oriented models, e. g., CIM [4], have been traditionally used to describe the CMDB. The CMDB describes the configuration of a data center from its resources to its hosted applications. The CMDB can be automatically generated by scanning a data center for running applications, e. g., using HP's Discovery and Dependency Mapping software. We envision an approach in which independent tools access a shared CMDB to act upon it. Knowing the behavior of tools, scripts, and applications, we can plan for their changes directly over a CMDB to achieve a certain goal, e. g., to deploy a 3-tier architecture. Figure 1 shows the model of the CMDB used throughout this work. Note, that our approach is capable of planning over any OO model. We modeled PMs, VMs, operating system (OS) images and services. Objects are linked by references. We modeled DB, WAS, and LB with their dependencies between each other. Furthermore, the model provides methods to create and destroy references. Each class is mapped to a state-transition system describing its behavior. For example, Fig. 2 denotes the state-transition system of class *WAS* in Fig. 1. The STS describes the behavior of a WAS instance. Every class owns a state property storing its current state. More formally, our solution comprises the following concepts:

Let $O = \{o_1, ..., o_n\}$ be a set of domain objects. An object $o \in O$ is a 3-tuple, such that $o = (id, s, (p_1, ..., p_n))$ where id is the unique id of o, s the current state of o, and p_i the values of o's properties. For an object o, $id(o)$ returns the unique id of o and $state(o)$ denotes the current state of o. Let $o = (id, s, (p_1, ..., p_n))$ and $o' = (id, s', (p'_1, ..., p'_n))$ be two versions of the object with identifier id. We define $changed(o, o')$ such that $changed(o, o') = true$ iff $s \neq s' \vee \exists p_i \in \{p_1, ..., p_n\} : p_i \neq p'_i$. A model $M \subseteq O$ is a subset of O such that references between objects are covered by the objects' properties. Similarly to changed objects we can define for two models M and M': $changed(M, M')$ iff $|M| \neq |M'| \vee \exists o \in M : \exists o' \in M' : id(o) = id(o') \wedge changed(o, o')$. We define $changed_{objs}(M, M')$ as the set of all objects changed or newly created between M and M'. More formally $changed_{objs}(M, M') = \{o' \in M' | \exists o \in M : changed(o, o') \vee \neg \exists o \in M : id(o) = id(o')\}$. Deleted objects still remain in M, but are not referenced by properties any more.

Our notion of *state-transition systems* (*STS*) is the following: Let $\sum = \{\sigma_1, ..., \sigma_n\}$ be the set of all state-transition systems. A state-transition system $\sigma \in \sum$ is a 2-tuple $\sigma = (S_\sigma, T_\sigma)$ such that $S_\sigma = \{s_1, ..., s_n\}$ are the states of σ and $T_\sigma = \{t_1, ..., t_n\}$ are the transitions of σ. For each domain object $o \in O$ there exists exactly one $\sigma \in \sum$ such that $state(o) \in S_\sigma$ throughout the existence of the object. It is denoted by $sts(o)$. A transition $t \in T_\sigma$ is a tuple $t = (s_{source}, s_{sink}, precond(M), C, effects(M, c))$ such that $s_{source} \in S_\sigma$ is the source and $s_{sink} \in S_\sigma$ the sink of t. $precond(M)$ describes a precondition evaluated over model M which needs to account to execute t. $C \subseteq M$ are the candidate objects of transition t. A transition can be instantiated for every candidate object $c \in C$, e. g., to link the candidate c to the object σ describes. $effects(M, c)$ describes the effects the execution of the transition has on the underlying model M dependent on candidate c. We call

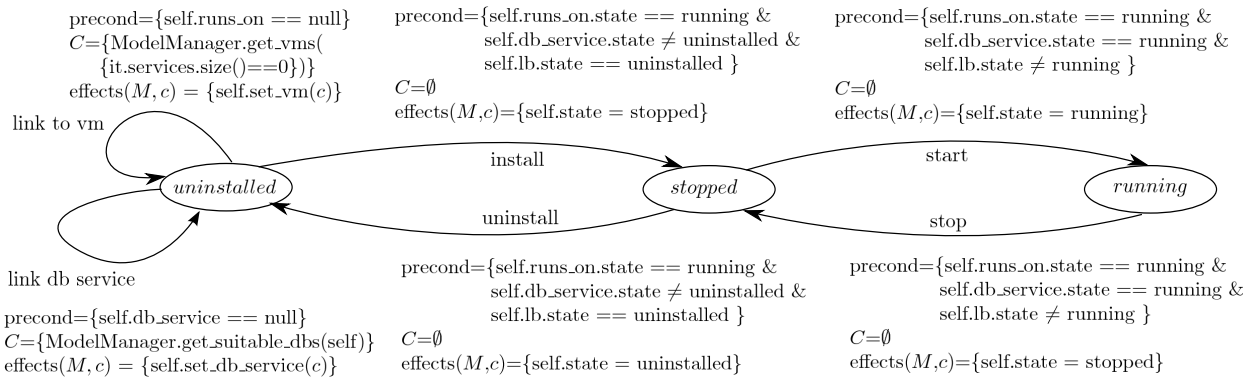

Figure 2. State transition system describing the lifecyle of a web application server. *self* refers to the object the STS describes.

$trans_{ex}(state(o)) \subseteq T_{sts(o)}$ the set of executable transitions of o. More formally, $trans_{ex}(state(o)) = \{t_i | t_i = (state(o), s_{sink_i}, precond_i(M), C_i, effects_i(c)) \in T_{sts(o)}$ and $precond_i(M)$ evaluates to true over the current model M $\}$. For planning purposes we define, as a central concept in this work, a *Logical Planning State* (*LPS*) lps as a tuple $lps = (o_1, ..., o_n) \subseteq M \times_n M$ such that all o_i are mutually disjunct. A LPS is a tuple formed of objects from the model, over which planning is done. $objects(lps) = objects((o_1, ..., o_n)) = \{o_1, ..., o_n\}$ are the objects of lps.

III. Algorithm

The Algorithm consists of a loop iterating over LPSs stored on a stack. The top of stack is read, and, if possible, a successor LPS is created and pushed on the stack. If no successor LPS exists, the planner backtracks by popping the stack and continues with the preceding LPS. Backtracking to the preceding LPS makes it necessary to restore all objects that changed. The search ends if the current top of stack satisfies the *goal check* (gc). The search space of LPSs is explored in a first-depth search order, making it necessary to check for a circle of LPS involving the newly added LPS. A LPS implements a stateful iterator shown in Alg. 1, which allows access to previously unreturned successor LPSs to explore the search space. For now we assume that lps_{init}, the initial LPS on the stack, holds all objects in M, i. e., $objects(lps_{init}) = M$. We will later relax this constraint in Section V.

Line 2 in Alg. 1 chooses an object o_i and an executable transition t of o_i in ascending order of the o_is stored in lps. The transition must not have been chosen in a previous call to the iterator, or if it has been before, there needs to be at least one candidate left ($C \neq \emptyset$). Note, that transitions can be instantiated multiple times for each candidate or only once if $C = \emptyset$. The candidate is initialized with *null* in Line 4. If a remaining candidate exists, *candidate* is initialized with a randomly chosen candidate from C (Line 6), and removed from C such that it is not used in future calls to the iterator. For example, consider transition *link to vm* in Fig. 2. The transition places a WAS server on a VM. The set of candidates C consists of all virtual machines present in the model, that have no services deployed on them. This is described by executable Groovy [15] code, dynamically executed at runtime, over the current model to retain C. The iterator randomly chooses a VM in Line 6. Note, that there are also transitions with $C = \emptyset$, e. g., transition *install* in Fig. 2, because installing a service is an internal change that makes candidate selection unnecessary. Candidates are a way to describe model transformations necessary during deployment to refine an abstract model to a fine grained full-deployment model as described in [14]. After that, the current model is assigned to M_{old} (Line 9). The effects are applied to the model and the candidate. In the case of the example in Fig. 2, $effects(M, c) = \{self.set\_vm(c)\}$ which calls method set_vm(c) on the WAS instance to establish a reference in the model between the WAS instance and the VM. The effects are described as executable Groovy Code [15] over the OO CMDB. Finally, Line 12 checks the two models for changes, as previously defined in Section II. Because VM and WAS changed, their new versions are both stored. The parent is set to lps, and suc is returned in Line 19.

IV. Fast Object Storing And Restoring

During planning objects need to be stored before they change between an LPS and its successor LPS (Alg. 1, Line 14) or they need to be restored when backtracking. Thus, fast object storing and restoring is crucial for the performance of the planner, especially in the presence of large models/data centers. In this section we evaluate the performance of four different *Model Recovery Strategies* (*MRSs*) possible in Groovy [15]. Groovy is used because in a previous work [3] we found it well suited to define DSLs that can be directly executed over object oriented models.

A. Recovery by Full Serialization

Using the *Recovery by Full Serialization* (*RbFS*) strategy the whole model is serialized when a new LPS is generated

Algorithm 1 Determine successor Logical Planning State

1: **procedure** GET_NEXT_LPS(lps) ▷ determining next lps
2: **if** $\exists o_i \in objects(lps) : trans_{ex}(state(o_i)) \neq \emptyset$ **then**
3: Choose o_i and $t = (state(o_i), s_{end}, precond(M), C, effects(M, c)) \in trans_{ex}(state_{o_i})$ in ascending order, such that t is executable and $(\neg chosen\_before(t) \vee (chosen\_before(t) \wedge C \neq \emptyset))$
4: $candidate \leftarrow null$
5: **if** $C \neq \emptyset$ **then**
6: $candidate \leftarrow c \in C$, random choice
7: $C = C - \{candidate\}$
8: **end if**
9: $M_{old} \leftarrow$ current model
10: apply_effects($effects(M, c)$)
11: $M_{new} \leftarrow$ current model
12: **if** $changed(M_{old}, M_{new})$ **then**
13: **for all** $o \in changed_{objs}(M_{old}, M_{new})$ **do**
14: o.store_new_version()
15: **end for**
16: **end if**
17: $suc \leftarrow$ new LPS($objects(lps)$)
18: suc.set_parent(lps)
19: return suc
20: **else**
21: return null
22: **end if**
23: **end procedure**

independently of changed objects. Figure 3 depicts an RbFS example in the right branch. Grey boxes denote LPSs, circles denote domain objects of the LPS ($objects(lps)$) and rectangles their properties. $lps1$ in Fig. 3 consists of three domain objects. $o1$ has a property holding value a and another property referencing $o2$. Similarly $o2$ holds b as a value and references $o1$. $o3$ references $o2$ and $o1$. During planning the effects of a transition are applied to $lps1$ (see Line 10 in Alg. 1) leading to changes in the object model. Changed object instances or properties are depicted in black. The effects of an applied transition changes a to a' and b to b'. To backtrack from $lps2$ to $lps1$ using RbFS, a previously serialized version of the model of $lps1$ is deserialized. This leads to instances $o1'$, $o2'$, and $o3'$ being semantically equivalent to the instances in $lps1$. Referential integrity is guaranteed because the original model was serialized as a whole.

B. Recovery by Partial Serialization

Recovery by Partial Serialization (*RbPS*) (Fig. 3, left branch) only serializes single objects and deserializes changed objects separate. To restore $lps1$ from $lps2$, previously serialized versions of $o1$ and $o2$ need to be deserialized. However, deserializing an instance of $o1$ to $o1'$ automatically deserializes a new version of $o2$, $o2''$ because serialization followed references held by a serialized object. The seperate deserialized versions $o1'$ and $o2'$ do not reference each other as their originals $o1$ and $o2$ did. Furthermore, $o3$ still references the outdated versions of $o1$ and $o2$ in $lps2$. Less objects need to be serialized compared to the RbFS strategy but overhead emerges because references need to be exchanged with unique identifiers that are mapped to the most recent instance of an object.

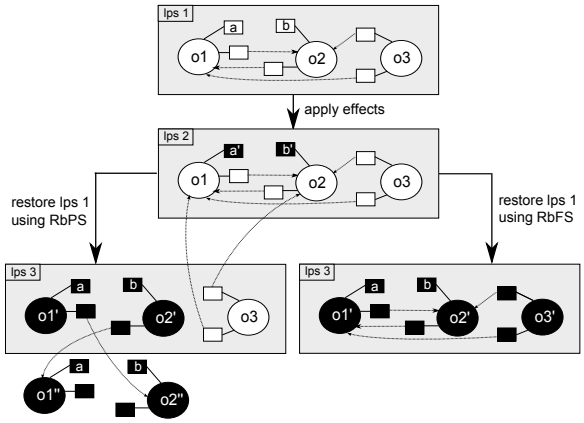

Figure 3. Restoring a LPS using Recovery by Partial Serialization (RbPS) and Recovery by Full Serialization (RbFS)

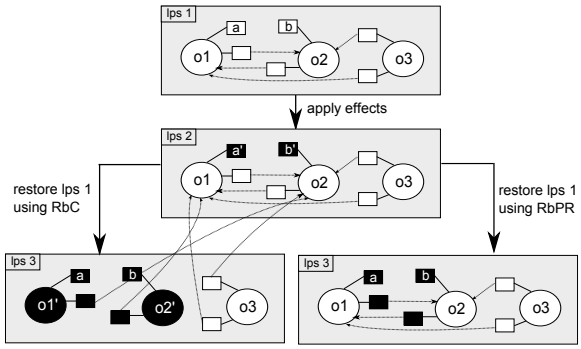

Figure 4. Restoring a LPS using Recovery by Cloning (RbC) and Recovery by Properties Renewal (RbPR)

C. Recovery by Cloning

The *Recovery by Cloning* (*RbC*) strategy restores objects by restoring a previously cloned version of the object. It is depicted in the left branch of Fig. 4. Compared to RbPS, RbC avoids the additional objects $o1''$ and $o2''$. The references of the restored objects, $o1'$ and $o2'$, reference $o1$ and $o2$ and not each other because $o1$ and $o2$ were cloned. Similarly to RbPS, $o3$ does not point to the restored objects $o1'$ and $o2'$. Thus, reference re-mapping becomes necessary.

D. Recovery by Properties Renewal

Recovery by Properties Renewal (*RbPR*) combines the advantages of RbFS (untouched references) and RbPS / RbC

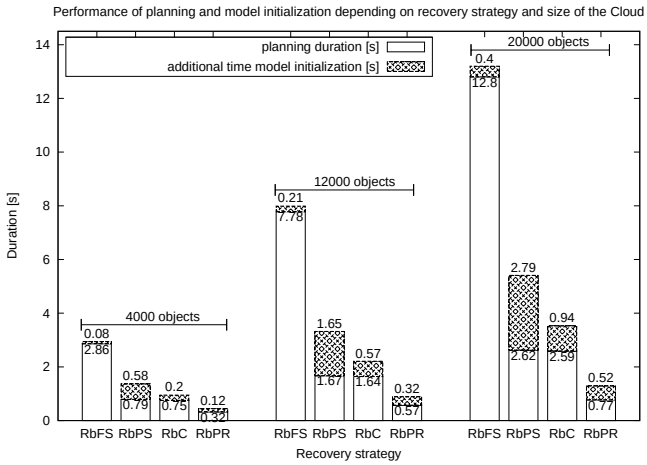

Figure 5. Planning duration and additional time needed for model initialization depending on recovery strategy in seconds

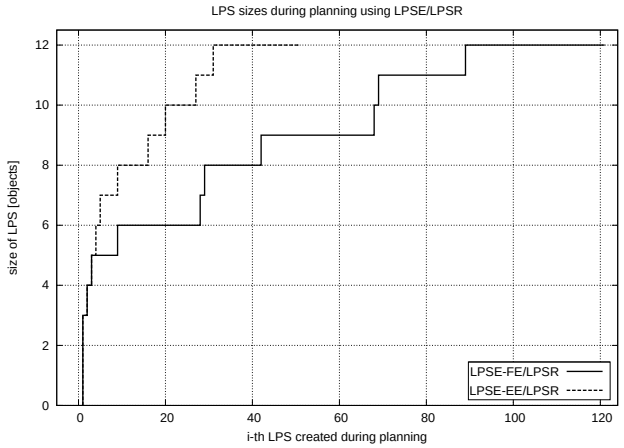

Figure 6. Size of the i-th LPS created during the run of the deployment of a 3-tier service using Front or End Expansion of LPSs

(only changed objects are stored/restored). RbPR is shown in the right branch of Fig. 4. The instance of an object stays the same during planning, references do not need to be rerouted. RbPR only stores and restores all properties of an object. To restore $lps1$ from $lps2$, all properties of $o1$ and $o2$ need to be restored. Although it would suffice to only restore changed properties, our implementation conducts stores and restores for all properties. Restoring previous references is hassle-free because there are no multiple instances of an object in different versions.

E. Performance Evaluation

Figure 5 shows the planning duration and the time needed for model initialization in seconds depending on the different model recovery techniques. Measurements were taken for model sizes between 4,000 and 20,000 domain objects. All optimization techniques (LPSE-EE/LPSR + P-CE) from Sections V and VI were applied. Each model consists to one fourth of VMs, PMs, OS images, and services (equally 1/3 of DB, WAS, and LB). The change to plan for is to achieve state *running* of a load balancer. This triggers deployment of a web application server, a database and the placement of services on VMs and VMs on PMs. Regarding solely planning time, RbPS and RbC perform nearly equally with RbC beeing marginally faster. Their runtime increases 3.5x while the model increases 5x in size. RbFS is the slowest strategy because the whole model is serialized for every LPS. It's runtime increases 4.5x, scaling worse than RbC and RbPS. RbFS needs to store and restore 1.3 million objects for the 20,000 object model, whereas RbC, RbPS, and RbPR only need to conduct 79 store/restore operations due to changed objects. RbPR scales best, with a factor of 2.4x in the presence of a 5-times larger model. Time for model initialization has to be taken into account if a planner is not running continuously. After the creation of the model, all domain objects need to be stored because it cannot be foreseen which objects will be changed during planning. Taking this time into account, RbC is between 44% and 53% faster than RbPS. RbPS is more costly than RbC regarding model initialization because each domain object has to be serialized separately. RbFS is the strategy with the smallest increase when taking model initialization into account for all model sizes because batch-serializing a whole model is cheaper than single storage operations for all objects as conducted by RbC, RbPS, and RbPR. Note, that RbC and RbPS performed equally with respect to planning duration because only 79 objects were stored/restored during planning. All in all, RbPR is the fastest strategy due to the following reasons: (1) It only stores / restores changed objects and (2) Overhead induced by serialization and cloning is avoided. If a modest amount of objects (≈ 80) are changed during planning, RbC and RbPS perform equally within milliseconds in respect to the planning duration. Note, that a continously running planning system would have to perform model initialization only once. We will use RbPR as our recovery strategy throughout the rest of this work because the other strategies have proven to be not performant enough in the context of large models or more complex changes.

V. Optimized Runtime for Large Data Centers

A highly optimized version of Alg. 1 was used to achieve the runtimes presented in Fig. 5. This section describes *Logical Planning State Expansion* (*LPSE*) and *Logical Planning Space Reduction* (*LPSR*), two techniques to prune large portions of the search space in order to make our planning approach feasible for large data centers hosted by a Cloud Computing provider.

An object $o \in M$, not part of lps, i. e., $o \notin objects(lps)$, is called *relevant with respect* (*rwr*) to an lps, if $\exists o_i \in objects(lps) : \exists t \in T_{sts(o_i)} : t$ is not executable because o violates the precondition of t. Thus, all objects violating a

precondition of an outgoing transition of the current state of an object in lps are called rwr to lps. For example, consider $lps = (was)$ and $M = \{db, was, lb\}$. was is currently in state *stopped*. The precondition to start was checks whether db is in state *running* (see transition *start* in Fig. 2). We assume db is in state *stopped* as well. Thus, the precondition of transition *stop* in Fig. 2 evaluates to *false*. $db \in M$ and $\notin objects((was))$ is thus relevant with respect to lps because it violates the precondition of transition $start$ of was, an object in $objects((was))$. Adding object db to lps will change the state of db in a subsequent planning step, finally enabling was to change its own state to *running*.
Logical Planning State Expansion (*LPSE*) is the technique of adding objects $o \in M \wedge o \notin objects(lps)$ that are rwr to lps to lps. Thus, holding $|objects(lps)|$ minimal. An LPS can be extended in two different ways: Be $lps = (o_1, ..., o_n)$ and object o_{exp} rwr to lps, then

- *End Expansion* (*LPSE-EE*) adds o_{exp} at the end of the tuple, i. e., lps is expanded to $lps' = (o_1, ..., o_n, o_{exp})$.
- *Front Expansion* (*LPSE-FE*) adds o_{exp} at the front of the tuple, i. e., lps is expanded to $lps' = (o_{exp}, o_1, ..., o_n)$.

Using LPS expansion with an initial LPS lps_{init} such that $objects(lps_{init}) = M$ does not make sense because lps_{init} cannot be extended beyond M. Determining a small, but not necessarily minimal, initial LPS can be done as follows: At the beginning of planning goal check gc is evaluated over M. Let $\{o_1, ..., o_n\} \subseteq M$ be the set of objects checked by gc. Set $lps_{init} = (o_1, ..., o_n)$. Note, that we can derive this set in our implementation because getters record access on objects in our CMDB. We call this initialization technique *LPS Reduction* (*LPSR*). It makes sense to use LPSR in combination with LPSE because LPSE will gradually extend lps_{init} with all domain objects having an indirect influence on the evaluation of the goal check.
Figure 6 shows the sizes of LPSs created during a run of the algorithm on the vertical axis. The horizontal axis denotes the i-th LPS created during planning. The underlying CMDB consisted of 3000 objects. The goal was to deploy a load balancer depending on one WAS server, which again depended on a DB. Regarding runtime and length of the generated plan LPSE-EE performs better LPSE-FE. LPSE-EE creates 50 LPSs during planning, while LPSE-FE creates 121 LPSs. LPSE-FE backtracks 50-times whereas End Expansion only backtracks 12-times. This leads to a performance loss for LPSE-FE of around 200ms. Similar results are achieved with an increased number of WASs but with a wider gap in planning duration and number of backtracks.
The superior performance of LPSE-EE is explained as follows: If no transition is executable, $\{o_{exp_1}, ..., o_{exp_n}\}$, the objects rwr to lps, are added at the end of lps such that $lps' = (o_1, ..., o_n, o_{exp_1}, ..., o_{exp_i})$ is the next LPS to be planned for. The planner will try to apply changes to an

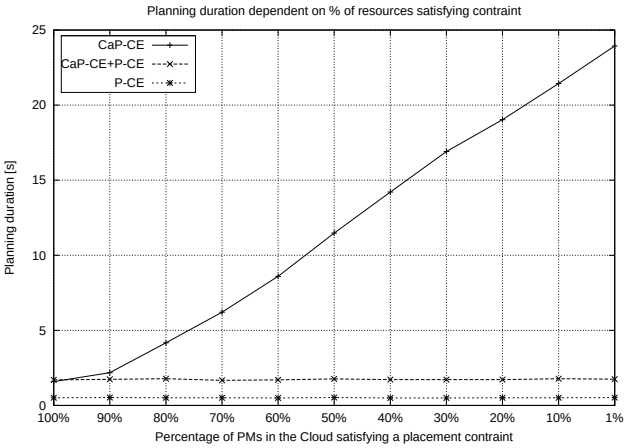

Figure 7. Planning duration in seconds depending on Constraint enforcement technique and percentage of qualifying resources

o_{exp_i} in lps' because changes to the o_i are not possible. The change to an o_{exp_i} is then directly picked up by changing an o_i in lps'' in the next iteration. Changing the order to LPSE-EE results in frequent backtracking.
Figure 6 shows phases were constant phases of lps size are followed by an expansion. Note, that constant phases are longer for LPSE-FE because backtracking occurs between newly created and old LPSs to achieve a state in $o_{exp_1}, ..., o_{exp_i}$ such that a subsequent o_i can execute a transition. Without LPSE the deployment example already leads to impractical runtimes as soon as $|M| \geq 18$. In this case runtimes vary between 500ms and several minutes, depending on the permutation of $o_1, ..., o_n$ in lps. LPSE drastically reduces the search space to objects that are directly, or indirectly relevant to reach an LPS that satisfies the goal check. Independently of the size of the model, the basic shape of the expansion curve stays the same. For example, independent of model size, planning starts with LB (added by LPSR), WAS is added, a DB, and finally VMs, images, and PMs that become rwrt to the LPS. This drastically reduces the search space and makes our approach feasible for large data centers.

VI. Declarative vs. Procedural constraint enforcement

A. Constraint enforcement strategies

Constraint enforcement during planning is important to generate valid intermediate LPSs. State-related constraints, e. g., a service (DB,WAS,LB) is not allowed to be in state *running* without its VM being *running*, are taken care of by the preconditions of transitions (see Fig. 2). However, there are other, non state-related constraints which need to be enforced. For example, constraints on the placement of services, VMs, and collocation constraints. In the context of the proposed algorithm they are equal in such a way, that a

suitable candidate (see Alg. 1, Line 6) needs to be chosen to apply a transition. Without limitation consider the constraint that VMs run exclusively on a PM. We call this constraint the *hosting constraint*. It can be enforced in two ways:
(1) *Check and Prune Constraint Enforcement* (*CaP-CE*): A CaP constraint is written in a declarative language to check a created LPS for validity. Formally, first order logic can be used to describe a CaP constraint. For instance, for the hosting constraint, we specify $\forall o \in M : o$ is a physical machine : $o.vms.size() \leq 1$. Practically, a CaP constraint is implemented as dynamically executable code over M. In case of the hosting constraint a *for* loop iterating over all PMs, and checking the size of the *vms* list.
(2) *Procedural constraint enforcement* (*P-CE*): Starting with a valid model, constraints can be enforced in a procedural way by choosing candidates that satisfy the constraint. For instance, transition *link pm* of a VM would determine the candidate PMs by querying the model for PMs with no VMs on them (similar to the *link to vm* transition in Fig. 2). Because only candidates satisfying the constraint are chosen, every created successor LPS automatically adheres to the hosting constraint.
The main difference between P-CE and CaP-CE is the following: P-CE will only create successor LPSs by choosing candidates that adhere to a constraint. CaP-CE creates successor LPS by choosing any candidate, but will backtrack from the successor LPS until a successor LPS adheres to the constraint, i. e., the right candidate was chosen.

B. Evaluation

Figure 7 shows the runtime of the planner in seconds for the LB deployment example using strategies CaP-CE, CaP-CE + P-CE, and P-CE under different percentages of resources satisfying the hosting constraint in the Cloud. M consists of each 1000 VMs, PMs, OS images and 3000 services. The horizontal axis in Fig. 7 denotes the percentage of PMs satisfying the hosting constraint.
From 90% onwards of qualifying resources, the runtime of CaP-CE increases linearily. CaP-CE randomly choses a candidate, which is more likely to be already loaded with less free resources available, leading to frequent backtracking to chose another candidate. In fact, the amount of LPS backtracked from correlates to the runtime curve of CaP-CE. For 100% qualifying PMs CaP-CE is about 100ms faster than CaP-CE + P-CE. In such lightly loaded data centers CaP-CE is better off without P-CE because P-CE poses an additional overhead and can only marginally reduce the generated candidate sets. CaP-CE + P-CE and P-CE have constant runtime (approx. 1.7s and 0.5s) independent of the amount of qualifying resources because P-CE will only chose candidates satisfying constraints, thus not generating LPSs that will later need to be pruned (i. e., backtracked from). CaP-CE in addition to P-CE poses an additional overhead because each PM is still checked for the hosting constraint whenever a LPS is chosen. However, both strategies can make sense in combination, when there are constraints difficult to be enforced by a transition, or if multiple constraints need to be enforced by a transition. Especially if lots of objects exist of a certain class, it is advised to use P-CE to propage a constraint. Besides performance CaP-CE and P-CE have other distinguishing characteristics:
(1) Maintenance : CaP-CE is easier to maintain because constraints are specified in a constraint database, separated from the behavior specification in the STS. P-CE specifies constraints implicitly by defining adequate candidate sets for a transition. Thus, the STS description has to be changed to apply different constraints.
(2) Integration of Optimizers: P-CE is well suited to easily integrate external optimizers that decide for an optimal placement. Several works propose optimization techniques [16], [11], [17] our approach can benefit from. For example, Li et al. [11] propose an approach for the energy efficient placement of virtual machines. Using procedural constraints, transition *link pm* of the VM can directly call an external application to determine an optimal candidate set. If we used CaP-CE, all placements would be tried by the planner until a placement is not pruned because it is efficient.

VII. Related Work

Related to state-based management of IaaS and SaaS constraints is the area of IT change planning [8], [9], [7], [6], [5]. None of the previously mentioned approaches aims to address planning for large scale data centers and object oriented CMDBs while reasoning about the preconditions and effects as our approach does.
Keller et al. [8] proposed CHAMPS which formalizes planning and scheduling as an optimization problem achieving a high degree of parallelism. Different to CHAMPS, we rely on an IT practitioner to define the planning domain. CHAMPS does not reason about the preconditions and effects of actions which can lead to unsound plans. Cordeiro et al. [9], [7] propose an approach to refine high-level IT changes into lower-level IT changes. Different to our work, the behavior of domain objects is not made explicit by state-transition systems. Their approach enables planning for general IT changes, while our approach focuses on state related changes and the transformation of models. Compared to CHAMPS they address the reuse of plans. Trastour et al. [6] propose a planning approach for IT changes explicitly working over OO models using a refinement version of the HTN algorithm [10]. Model storing and restoring was performed using Hibernate and the snapshot capabilities of a database. This was found to be a bottleneck. In addition to that, the application of the approach to large data-centers was out of the scope of this work. Maghraoui [5] et al. propose transforming the OO CMDB to predicates to use a variant of the POP [10] algorithm for planning. Different

to them our approach can directly plan over object oriented models. Their algorithm searches within the plan space while ours searches within the state space. We addressed plan space search over OO models in our previous work [3]. Our former work proposed a hybrid approach between state-space planning and plan-space planning to address an abstraction mismatch between the refinement of IT changes and planning according to state-dependencies. However, we did not examine the performance of different model storing / restoring techniques and the application of this algorithm to a large scale data center was out of the scope of that work. Compared to this work, our previous work was capable to plan for general IT changes, while we limit ourself to state-related changes to IaaS and SaaS components in this work.

VIII. Conclusion

We identified a performance short-coming when it comes to plan over OO CMDBs in the context of large data centers. We proposed a planning approach that is capable to plan for SaaS and IaaS constraints even for large data centers. Using a deployment case study for a 3-tier application we showed the approaches scalability to a data center in size of 5000 PMs and VMs. We found that storing and restoring of OO models can be fastest achieved by storing / restoring properties of objects. We evaluated the effect of procedural and declarative constraints on the runtime of the planner and found that procedural constraints can achieve constant runtime for placement constraints in the presence of highly loaded data centers.

For future work we would like to extend our approach to other examples besides the deployment case study, to apply it to Virtual Appliances [12], and to examine how configuration aspects [14] can be addressed using the notion of state-transition systems.

Acknowledgment

The authors like to thank Nigel Edwards and Lawrence Wilcock, both HP Labs Bristol, for their valuable comments to improve this work.

References

[1] A. Michael *et al.*, "Above the Clouds: A Berkeley View of Cloud Computing," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28, Feb 2009.

[2] V. Srinivasmurthy *et al.*, "Web2Exchange: A Model-Based Service Transformation and Integration Environment," in *Proc. IEEE Int. Conf. on Services Computing (SCC'09)*, 2009, pp. 324–331.

[3] S. Hagen, N. Edwards, L. Wilcock, J. Kirschnick, and J. Rolia, "One Is Not Enough: A Hybrid Approach for IT Change Planning," in *Proc. IFIP/IEEE Int. Workshop on Distributed Systems: Operations and Management (DSOM'09)*, 2009, pp. 56–70.

[4] DMTF. CIM Standard. [Online]. Available: http://www.dmtf.org/standards/cim/

[5] K. E. Maghraoui, A. Meghranjani, T. Eilam, M. H. Kalantar, and A. V. Konstantinou, "Model Driven Provisioning: Bridging the Gap Between Declarative Object Models and Procedural Provisioning Tools," in *Proc. CM/IFIP/USENIX Int. Middleware Conf.*, 2006, pp. 404–423.

[6] D. Trastour, R. Fink, and F. Liu, "ChangeRefinery: Assisted Refinement of High-Level IT Change Requests," in *Proc. IEEE Int. Symp. on Policies for Distributed Systems and Networks (POLICY'09)*, 2009, pp. 68–75.

[7] W. L. da Costa Cordeiro *et al.*, "A Runtime Constraint-Aware Solution for Automated Refinement of IT Change Plans," in *Proc. IFIP/IEEE Int. Workshop on Distributed Systems: Operations and Management (DSOM'08)*, 2008, pp. 69–82.

[8] A. Keller, J. Hellerstein, J. Wolf, K.-L. Wu, and V. Krishnan, "The CHAMPS System: Change Management with Planning and Scheduling," in *Proc. IEEE/IFIP Network Operations and Management Symp. (NOMS'04)*, 2004, pp. 395–408.

[9] W. L. da Costa Cordeiro *et al.*, "ChangeLedge: Change Design and Planning in Networked Systems based on Reuse of Knowledge and Automation," *Computer Networks: The Int. J. of Computer and Telecommunications Networking*, vol. 53, pp. 2782–2799, 2009.

[10] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning: Theory & Practice*. Morgan Kaufmann, 2004.

[11] B. Li *et al.*, "EnaCloud: An Energy-Saving Application Live Placement Approach for Cloud Computing Environments," in *Proc. IEEE Int. Conf. on Cloud Computing (CLOUD'09)*, 2009, pp. 17–24.

[12] C. Sun, L. He, Q. Wang, and R. Willenborg, "Simplifying Service Deployment with Virtual Appliances," in *Proc. IEEE Int. Conf. on Services Computing (SCC'08)*, 2008, pp. 265–272.

[13] A. V. Konstantinou *et al.*, "An Architecture for Virtual Solution Composition and Deployment in Infrastructure Clouds," in *Proc. of the Int. Workshop on Virtualization Technologies in Distributed Computing (VTDC'09)*, 2009, pp. 9–18.

[14] T. Eilam, M. H. Kalantar, A. V. Konstantinou, G. Pacifici, and J. Pershing, "Managing the Configuration Complexity of Distributed Applications in Internet Data Centers," *IEEE Communications Magazine*, vol. 44, pp. 166–177, 2006.

[15] D. Koenig *et al.*, *Groovy in Action*. Manning, 2007.

[16] J. Li, J. Chinneck, M. Woodside, and M. Litoiu, "Deployment of Services in a Cloud Subject to Memory and License Constraints," in *Proc. IEEE Int. Conf. on Cloud Computing (CLOUD'09)*, 2009, pp. 33–40.

[17] Y. Diao, H. Jamjoom, and D. Loewenstern, "Rule-Based Problem Classification in IT Service Management," in *Proc. IEEE Int. Conf. on Cloud Computing (CLOUD'09)*, 2009, pp. 221–228.