Planning with Global Constraints for Computing Infrastructure Reconfiguration

Problem Solving Using Classical Planners
AAAI Technical Report WS-12-12

Planning with Global Constraints for Computing Infrastructure Reconfiguration

Herry Herry and **Paul Anderson**
School of Informatics
University of Edinburgh
Edinburgh, UK

Abstract

This paper presents a prototype system called SFplanner which uses an automated planning technique to generate workflows for reconfiguring a computing infrastructure. The system allows an administrator to specify a configuration task which consists of current state, desired state and global constraints. This task is compiled to a grounded finite-domain representation as the input for the standard (unmodified) Fast-Downward planner in order to automatically generate a workflow. The execution of the workflow will bring the system into the desired state, preserving the global constraints at every stage of the workflow.

Introduction

The growing size and complexity of computing infrastructures has increased awareness of the need for system configuration tools. These help system administrators to manage large scale systems, such as data centers, by automating the configuration tasks in order to satisfy a particular specification.

Out of several proposed approaches, the declarative approach has become widely accepted as the most appropriate solution - the administrator describes the "desired" state of the system and the tool computes the necessary actions to bring the system from its current state into the desired state. Most of the currently popular tools apply a broadly declarative approach - for example, Puppet (Puppet Labs 2012), BCFG (Desai et al. 2003) and LCFG (Anderson and Scobie 2002).

However, none of the above tools make any guarantees about the order of the actions when implementing a configuration change. (Herry, Anderson, and Wickler 2011) demonstrates that the actions are often executed in an essentially indeterminate order which is highly likely to bring the system into a state that violates some essential constraints.

One approach to this problem has been the use of provisioning tools - the administrator defines and stores workflows so they can be invoked and scheduled automatically by a central controller to satisfy the ordering constraint. IBM Tivoli Provisioning Manager (IBM Corp. 2012), Microsoft Opalis (Microsoft Corp. 2012) and ControlTier (DTO Solutions 2012) are examples which provide this capability. Unfortunately, this still requires that the workflows are computed manually. Even in a small system, a very large number of workflows could be required to cater for every eventually. In addition, choosing an appropriate workflow to suit a particular desired state is not always obvious.

SmartFrog (Goldsack et al. 2009) is a configuration tool which has been augmented with "behavioural signatures" to support a different approach. This allows us to explicitly define state-dependencies between system components, and a change of state in one component may depend on changes of state in other components. This could produce a cascading effect of state changes with a particular order. However, we must compute the state-dependencies manually which is error-prone and time consuming.

In our previous work (Herry, Anderson, and Wickler 2011), we proposed a solution based on automated planning techniques which generates workflows "on the fly". This allows us to define the current state as well as the desired state, together with a set of constraints defined in actions. The workflow is generated and executed to implement the transitions of the system without violating the specified constraints. Unfortunately, this system does not allow the administrator to define global constraints i.e. constraints that must be satisfied either in the intermediate and goal states[1]. All constraints must be defined explicitly as preconditions associated with some actions. A change to a constraint forces us to modify the actions.

Clearly, we could modify the action in order to satisfy the global constraints, but in real situations this is impractical; the specification is commonly written by a software engineer or expert who has a deep knowledge of the software which the administrator does not have. Determining whether an action must be modified or not may be as hard as the planning itself, since the constraints for execution could require arbitrary states to be achieved by previous actions. In addition, a modification may not be allowed due to a lack of permission or a license violation, for example.

This paper describes a prototype system called SFplanner

[1] Global and goal constraints in SFp are equivalent to the *always* and *at end* modalities of state trajectory constraints in PDDL3 (Gerevini and Long 2005) respectively.

```
class Service {
   running false
   action start {
      precondition { }
      postcondition {
         $this.running true
      }
   }
   action stop {
      precondition { }
      postcondition {
         $this.running false
      }
   }
}
class Client {
   refer as *Service
   action changeReference(s as *Service) {
      precondition { }
      postcondition {
         $this.refer $s
      }
   }
}
```

Figure 1: Class definition in file *class.sfp*.

```
#include file("class.sfp")
//--- current state ---//
web1 as Service {
   running true
}
web2 as Service {
   running false
}
pc1 as Client {
   refer $web1
}
pc2 as Client {
   refer $web1
}
pc3 as Client {
   refer $web1
}
//--- goal constraint ---//
constraint goal {
   $web1.running false
}
//--- global constraint ---//
constraint global {
   $pc1.refer.running true
   $pc2.refer.running true
   $pc3.refer.running true
}
```

Figure 2: An SFp configuration task.

which has the ability to generate a workflow automatically between two system states. In contrast to previous work, this allows the administrator to define a set of global constraints explicitly in the configuration specification. This specification is compiled to a grounded finite-domain representation (FDR) (Helmert 2009) as the input for the standard (unmodified) Fast-Downward planner (Helmert 2006). After post-processing, the generated workflow can achieve the desired state while preserving the global constraints at every intermediate stage.

We start by introducing a conceptual model of system configuration, and we then describe the approach used by SFplanner to find a solution to a planning task in this domain. This is followed by a more detailed description of the SFplanner architecture. Finally, we present some experimental results on the "cloud-burst problem", and we discuss some related work and possible future directions.

Modelling the System Configuration

Object-oriented models, such as the Common Interface Model (CIM), have been widely used in industry to model artifacts of computing infrastructure. We adapt a similar approach to describe the configuration of the managed system. Each system resource (artifact) is modelled as an object which may have one or more attributes. Each attribute may be assigned a value, and the collection of attribute/value pairs represents the state of an object. An object may also have one or more actions. The execution of an action may change the object's state by modifying its attribute values.

SFplanner has a domain specific language called SFp which adopts this model. It is an extension of the SmartFrog (SF) (Goldsack et al. 2009) language, which is a prototype-based language, that allows us to define an object as a member of particular class. In SF and SFp, each object may have a set of attributes with primitive, non-primitive or reference values. SF does not have any notation for declaring an action. However in SFp, we can define actions on an object which have parameters, preconditions and postconditions. Furthermore, SFp also allows us to define the current state of the system, as well as the goal and the global constraints, as part of configuration task.

The SFp language allows the model of the system to be divided into several modules represented by a set of files. In creating the model, an administrator can use any module simply by including the file which contains the corresponding classes or resources into the main specification. The module specification can be reused on another system simply by including the specification in the software distribution. A software engineer or expert can create a new class (abstract resource) that inherits another class behaviour using the inheritance notation. This feature allows re-usability of specifications which is a common practice in the real world.[2]

Figure 2 shows a model of a system which consists of two services (*web1* and *web2*) and three clients (*pc1*, *pc2* and *pc3*). It uses the class definition in figure 1 where each service has an attribute *running* and two actions i.e. *start* and *stop*. The client has an attribute *refer* with reference value

[2]There are some features under-development which are not covered in this paper such as array/set data structures and creating/deleting objects in actions.

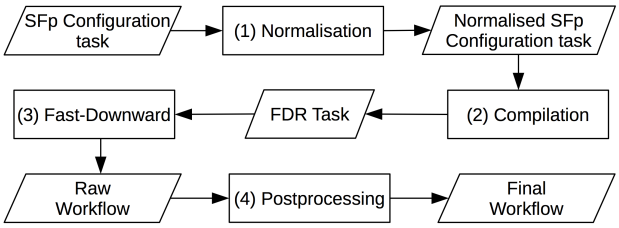


Figure 3: Steps for solving SFp configuration task.

to a service, and an action *changeReference* which changes *refer*'s value to the service defined in the parameter.

An SFp configuration task is defined by specifying the current state as well as the goal state and the global constraints. In figure 2, the current state of *web1* and *web2* is *running* and *stopped* respectively, and *pc1*, *pc2* and *pc3* are referring to *web1*. The goal constraint of the system is that *web1* is stopped. And its global constraint is that *pc1*, *pc2* and *pc3* must always refer to a running service.

Generating the Workflow

Intuitively, the state of a system which is defined in SFp can be described by a set of state variables where each variable represents an object's attribute. Each variable has a finite domain of possible values, and the action modifies its value to attain the goal state. Formally, this can be described as a normalised SFp configuration task.

This is similar to the definition of a planning task in finite-domain representation (FDR task). In (Helmert 2009), an FDR task could be defined as a 5-tuple $\Pi = \langle V, s_0, s_g, A, O \rangle$ where:

- $V = \{v_1, \ldots, v_n\}$ is a set of state variables (fluents), each is associated with a finite domain D_v. If $d \in D_v$ we call the pair $v = d$ an atom.
  A partial variable assignment over V is a function s on some subset of V such that $s(v) \in D_v$, wherever $s(v)$ is defined. If $s(v)$ is defined for all $v \in V$, s is called a state.
- s_0 is a state called an initial state, and s_g is a partial variable assignment called the goal.
- A is a set of axioms over V.
- O is a set of operators, where an operator is a triple $\langle name, pre, eff \rangle$, where name is a unique symbol to distinguish an operator from others, and pre and eff are partial variable assignments called preconditions and postconditions, respectively.

A global constraint in SFp configuration task could be considered as a partial variable assignment. Thus, we could define a normalised SFp configuration task as a 6-tuple $\Theta = \langle V, s_0, s_g, s_c, A, O \rangle$ where:

- s_c is a partial variable assignment called the global constraint.

Based on this definition, we can solve an SFp configuration task by compiling it into an FDR task as the input for the Fast-Downward planner (Helmert 2006) to find the solution[3]. Figure 3 illustrates a step-by-step process for solving an SFp configuration task in SFplanner. The details of each step can be summarized as follows:

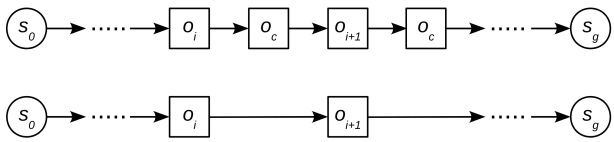


Figure 4: A raw workflow (top) and a final workflow (bottom).

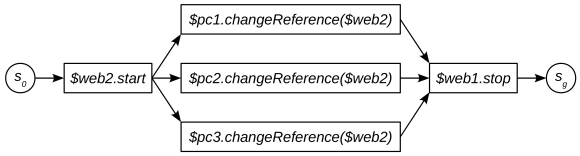


Figure 5: The generated workflow for SFp configuration task defined in figure 2.

1. **Normalisation:** transforms an SFp configuration task into a normalised SFp configuration task. This involves the following sub-steps:
   (a) all objects' attributes are identified and each of them is replaced by an unique state variable;
   (b) all possible values of a particular type, either primitive or non-primitive, are populated and grouped into a finite domain. Then this domain is assigned to all state variables which have this type;
2. **Compilation:** compiles a normalised SFp configuration task into FDR task via several sub-steps i.e.:
   (a) each action is translated into one or more grounded operators, where each grounded operator is assumed to violate the global constraint and cannot be selected (executed) if the previous state does not satisfy this constraint;
   (b) compute a set of axioms which are related to the action constraints as well as the goal and global constraints;
   (c) an artificial grounded operator *verify-global-op* is introduced which must be selected directly by the planner after selecting another grounded operator in order to verify whether the resulting state satisfies the global constraint, the global constraint is set as its prevail condition, and its effect is to assign *true* to the state variable *satisfied-global*;
   (d) a new state variable *satisfied-global* with boolean type is introduced as the flag of the global constraint. It is assigned *true* by *verify-global-op* if the intermediate or goal state satisfies the global constraint, otherwise *false*;
   (e) in the initial state, each state variable is assigned a unique value as defined in the initial state of the normalised SFp configuration task, while the variable *satisfied-global* is assigned *true*;

[3]The compilation result is as normally produced by Fast-Downward's PDDL to FDR translator.

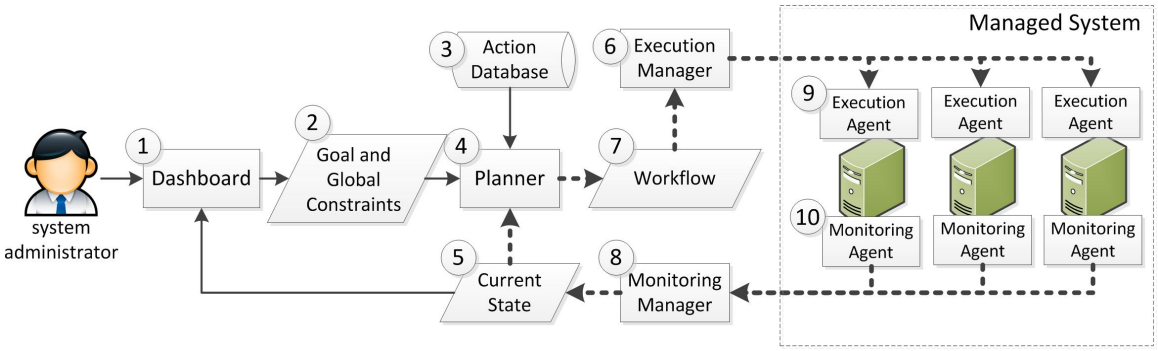


Figure 6: *System Architecture of SFplanner*

(f) the goal state consists of a partial variable assignment as defined in the goal and global constraints of the normalised SFp configuration task, and the variable *satisfied-global* is assigned *true*;

3. **Fast-Downward:** generates the solution of the FDR task by using the output of the previous step as the input of the unmodified Fast-Downward planner. If a solution exists, it will generate a raw workflow which not only achieves the goal state but also satisfies the global constraint at every intermediate step;

4. **Postprocessing:**

(a) removes the abstract operator *verify-global-op* (o_c) from the raw workflow as denoted in figure 4;

(b) generates the partial-order workflow from previously generated total-order workflow using the approach described in (Veloso, Perez, and Carbonell 1990) in order to enable parallel execution;

(c) converts the workflow into JavaScript Object Notation (JSON) format so that it can be easily processed by the execution manager.

Figure 5 shows the generated workflow based on the SFp configuration task shown in figure 2.

The Architecture of SFplanner

As shown in figure 6, SFplanner consists of six main components: dashboard, planner, execution manager, execution agent, monitoring manager and monitoring agent. The relationship between these components is as follows[4]:

- The action database (3) holds a set of actions which would normally be defined by an expert, system engineer, software engineer or other specialist.
- An open-source tool called *facter* is used as the monitoring agent (10) which runs on each managed node. The SFplanner monitoring manager (8) in a central controller periodically pulls and aggregates information from all agents in order to generate the overall current state of the managed system.
- Using the SFplanner dashboard (1), the system administrator specifies and submits the goal and global constraints (2) of the system.
- The planner (4) generates the workflow (7) in order to deploy a new specification on the managed system. This generated workflow is based on the available actions, the current state, and the goal and global constraints.
- The execution manager (6) orchestrates the deployment of the configuration changes by coordinating the execution of actions with all the execution agents (9).

It is possible for a failure to occur during the execution of a workflow, and SFplanner implements a pragmatic approach to handling such failures: each execution agent is responsible for ensuring that each action is executed successfully. Furthermore, it also has to detect and report any failure to the execution manager. If a failure is reported, the execution of the workflow is immediately discontinued. The execution manager then sends a request to the planner to generate an alternative workflow for later execution. A notification is also shown in the dashboard so that the administrator is informed of the failure, and can submit an alternative goal and global constraints, if required.

SFplanner could be used as a fully-automated configuration tool which can correct any drift in the configuration without any human intervention. In figure 6, the dash line arrows show a process loop which could be set to be executed periodically by the administrator in order for SFplanner to verify and correct any drift of the current state from the goal and global constraints.

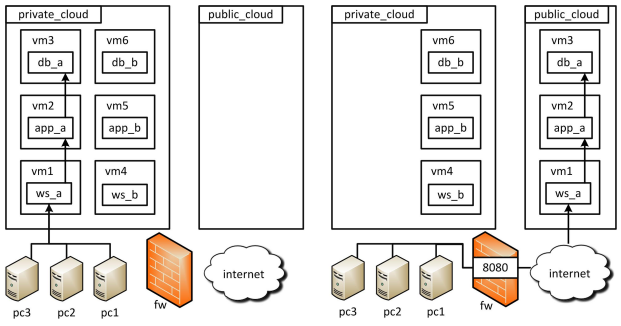
All components of SFplanner are implemented in Java, except the Fast-Downward which is in C++. Some components i.e. SFp-planner and SFp-dashboard have been published as open-source software which is available for download from the following URL:

`http://homepages.inf.ed.ac.uk/s0978621/`

Example

We simulated an example of a "cloud-burst" scenario on a 3-tier web application: an organization wants to dynamically migrate an application from its limited computing infrastructure to a public cloud in order to address a spike in demand which cannot be handled internally.

[4]Each number represents the component in figure 6



(a) Current state (b) Goal state

Figure 7: The states of the cloud-burst problem.

We assume that the company has a private cloud infrastructure which runs two 3-tier web applications: web application A consists of *ws_a*, *app_a* and *db_a* running on virtual machines *vm1*, *vm2* and *vm3* respectively. Web application B consists of *ws_b*, *app_b* and *db_b* installed (but not running) on virtual machines *vm4*, *vm5* and *vm6* respectively. Web application B is a backup of web application A, so that any client (*pc1*, *pc2* or *pc3*) may use either of them.

Due to the limited resources of the physical machines in the company's private cloud, web application A may not be able handle the spike of data processing demanded by the clients which usually happens on the first week of each month. Therefore, the administrator wants to migrate web application A to the public cloud before the start of the spike period. Figure 7 illustrates the initial state and the goal state of the system.

Unfortunately, the workflow for the migration process is not obvious, since the following constraints must not be violated:

1. The service must always available for 24-hours a day without any down-time, including during the migration process.
2. All components of a web application must be in the same infrastructure when it serves the clients. This means that in the final state, *web_a*, *app_a* and *db_a* must be in the same infrastructure (either the private or the public cloud).
3. The firewall must be reconfigured to allow all LAN PCs to be able to connect with *web_a* on the public cloud.

To solve this problem, the administrator would use SFplanner's dashboard to specify the goal and global constraints. These are shown in figure 8. We can see that the goal defines *vm1*, *vm2* and *vm3* to be in the public cloud, while *vm4*, *vm5* and *vm6* are still in private cloud and they are stopped. Port 8080 on firewall *fw* is open so that *pc1*, *pc2* and *pc3* can access *ws_a* which is in the public cloud. Furthermore, the global constraint section states that *pc1*, *pc2* and *pc3* must always refer to a running service.

Based on these facts, SFplanner generates a workflow which consists of 34 actions: web application B is started and all clients are redirected to refer to server B before migrating all components of web application A to the public cloud. After the migration process is finished, and before all clients are redirected back to A, SFplanner opens port 8080 of the firewall *fw* so that all connection requests from clients can be forwarded to *ws_a* which is now in the public cloud. Finally, all clients are redirected back to A and all components (including the virtual machines) of web application B are stopped. A complete generated workflow is illustrated in figure 9.

```
constraint goal {
    $node1.on              $public_cloud
    $node2.on              $public_cloud
    $node3.on              $public_cloud
    $node4.on              $private_cloud
    $node5.on              $private_cloud
    $node6.on              $private_cloud
    $node4.running         false
    $node5.running         false
    $node6.running         false
    $pc1.refer             $ws_a
    $pc2.refer             $ws_a
    $pc3.refer             $ws_a
    $fw.port8080.opened    true
}
constraint global {
    $pc1.refer.running  true
    $pc2.refer.running  true
    $pc3.refer.running  true
}
```

Figure 8: The goal and global constraints defined in SFp.

To solve this configuration task, the planner was run on a computer with a 2.16 GHz dual-core processor and 2 GB of memory, but we only used a single core for computation. We set Fast-Downward to use FF heuristic in the planning process. After 10 tests, the planner was able to solve the problem in average of 1.017 seconds. This included all steps as described in figure 3.

A complete SFp configuration task for this problem which includes the class definitions, the initial state, the goal constraint, and the global constraint is available from the following URL:

`http://homepages.inf.ed.ac.uk/s0978621/cloud.sfp`

and a complete version of the generated workflow can be accessed from the following URL:

`http://homepages.inf.ed.ac.uk/s0978621/cloud.sfw`

Related Work

There has been some previous work on the application of automated planning for configuring computing infrastructure. In CHAMPS (Keller et al. 2004), the requested operators are translated into a set of imperative tasks and organized as a workflow to satisfy the given constraints and maximize the degree of parallelism. The work in (El Maghraoui et al. 2006) proposes translating the facts from Object Oriented Configuration Management Database (OO CMDB) to PDDL and uses a variant of Partial-Order Planner for plan-

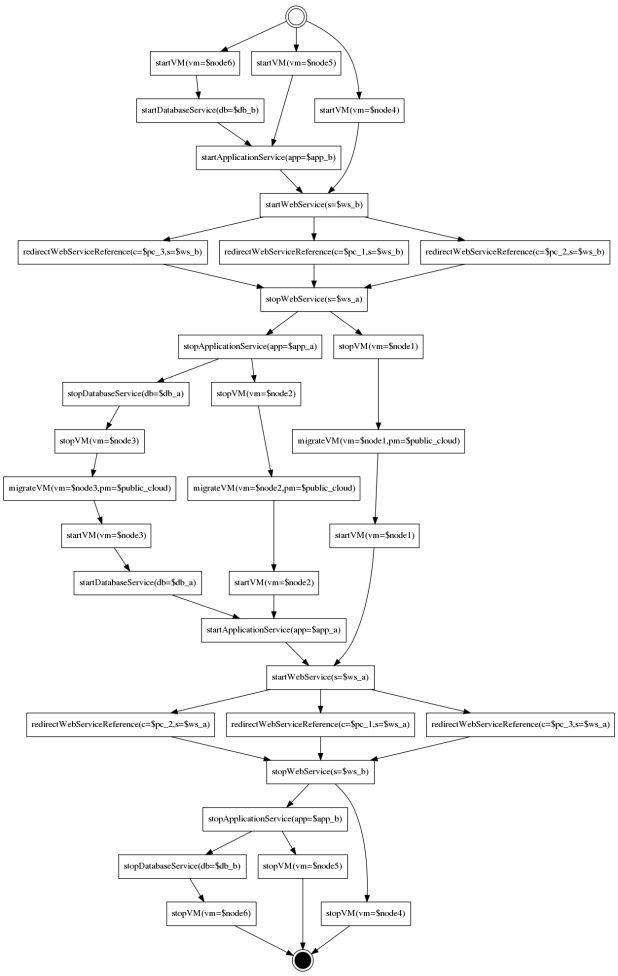


Figure 9: The generated workflow for cloud-burst problem.

ning. Hagen (Hagen et al. 2009) introduces a hybrid planner to generate the workflow from the facts on OO CMDB, and (Hagen and Kemper 2010) uses a variant of state-space planner that directly plans over OO CMDB to generate the workflow. The work in (Levanti and Ranganathan 2009) demonstrates an approach to simplify the interface to the planner by presenting a set of tags as representation states and operations - the user can then select one or more of these tags which are then mapped into a workflow using SPPL planner.

Our previous work (Herry, Anderson, and Wickler 2011) shows that an off-the-shelf planner can be used to automatically generate the workflow for solving a configuration task. It also demonstrates how a planner can be integrated with declarative configuration tools.

The above approaches demonstrate the viability of automated planning for changes to the configuration of a computing infrastructure. However, we are not aware of any other system which supports global constraints in the planning process.

Conclusion and Future Work

This work has presented an experimental configuration tool called SFplanner which can automatically compute a workflow for reconfiguring a computing infrastructure. The execution of the generated workflow brings the managed system into the desired state. Uniquely, the system supports global constraints which can be used to preserve any necessary properties of the system at every stage of the configuration process.

However, in practice, a centralized workflow orchestration is not always ideal - for example, reconfiguration typically occurs as an autonomic reaction to failures. In this case, it is likely that the system failure will also have disrupted the ability of the central controller to communicate effectively with all of the necessary components. To address this, we are currently investigating a more distributed and localised approach to workflow execution which still retains the control of the centralized planning. This should allow more autonomy for individual components and improve the system's resilience, as well as decreasing the response time.

Acknowledgments

We would like to thank Gerhard Wickler and Michael Rovatsos from University of Edinburgh, and Lawrence Wilcock and Eric Deliot from HP Labs Bristol for their valuable feedbacks. This research is fully supported by a grant from 2010 HP Labs Innovation Research Program Award.

References

Anderson, P., and Scobie, A. 2002. LCFG: The next generation. In *UKUUG Winter Conference*.

Desai, N.; Lusk, A.; Bradshaw, R.; and Evard, R. 2003. BCFG: A Configuration Management Tool for Heterogeneous Environments. In *Proceedings of IEEE International Conference on Cluster Computing*. IEEE Computer Society.

DTO Solutions. 2012. ControlTier.

El Maghraoui, K.; Meghranjani, A.; Eilam, T.; Kalantar, M.; and Konstantinou, A. 2006. Model driven provisioning: Bridging the gap between declarative object models and procedural provisioning tools. In *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, 404–423. Springer-Verlag New York, Inc.

Gerevini, A., and Long, D. 2005. Plan constraints and preferences in pddl3. Technical report, University of Brescia.

Goldsack, P.; Guijarro, J.; Loughran, S.; Coles, A.; Farrell, A.; Lain, A.; Murray, P.; and Toft, P. 2009. The smartfrog configuration management framework. *ACM SIGOPS Operating Systems Review* 43(1):16–25.

Hagen, S., and Kemper, A. 2010. Model-Based Planning for State-Related Changes to Infrastructure and Software as a Service Instances in Large Data Centers. In *2010 IEEE 3rd International Conference on Cloud Computing*, 11–18. IEEE.

Hagen, S.; Edwards, N.; Wilcock, L.; Kirschnick, J.; and Rolia, J. 2009. One is not enough: A hybrid approach for it change planning. *Integrated Management of Systems, Services, Processes and People in IT* 56–70.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research* 26(1):191–246.

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence* 173(5-6):503–535.

Herry, H.; Anderson, P.; and Wickler, G. 2011. Automated planning for configuration changes. In *Proceedings of the 2011 LISA Conference*. Usenix Association.

IBM Corp. 2012. Integrated Service Management software, IBM Tivoli.

Keller, A.; Hellerstein, J.; Wolf, J.; Wu, K.; and Krishnan, V. 2004. The CHAMPS system: Change management with planning and scheduling. 1:395–408.

Levanti, K., and Ranganathan, A. 2009. Planning-based configuration and management of distributed systems. In *Integrated Network Management, 2009. IM'09. IFIP/IEEE International Symposium on*, 65–72. IEEE.

Microsoft Corp. 2012. Microsoft Opalis.

Puppet Labs. 2012. Puppet.

Veloso, M. M.; Perez, M. A.; and Carbonell, J. G. 1990. Nonlinear planning with parallel resource allocation. In *In Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control*, 207–212. Morgan Kaufmann.