Automated Planning for Configuration Changes

Herry Herry
University of Edinburgh
Edinburgh, UK
h.herry@sms.ed.ac.uk

Paul Anderson
University of Edinburgh
Edinburgh, UK
dcspaul@ed.ac.uk

Gerhard Wickler
University of Edinburgh
Edinburgh, UK
g.wickler@ed.ac.uk

Abstract

This paper describes a prototype implementation of a configuration system which uses automated planning techniques to compute workflows between declarative states. The resulting workflows are executed using the popular combination of ControlTier and Puppet. This allows the tool to be used in unattended "autonomic" situations where manual workflow specification is not feasible. It also ensures that critical operational constraints are maintained throughout the execution of the workflow. We describe the background to the configuration and planning techniques, the architecture of the prototype, and show how the system deals with several examples of typical reconfiguration problems.

Keywords: configuration management, infrastructure, cloud computing, planning, IaaS

1 Introduction

The growing size and complexity of computing infrastructures has increased awareness of the need for good system configuration tools, and most sites now use some form of tool to manage their configurations. Furthermore, declarative specifications are now widely accepted as the most appropriate approach - the specification describes the "desired" state of the system, and the tool computes the necessary actions to move the system from its current state into this desired state. This has the advantage that the final *state* of the system is explicitly specified, and we can have some confidence that the state of the running system matches our requirements. Previous approaches were more error-prone because they involved specifying the *actions* (for example, using imperative scripts), and the final outcome would not always be obvious. With varying degrees of strictness, most of the currently popular tools take a broadly declarative approach - for example, Puppet [16] , Cfengine [3], BCFG [4] and LCFG [1].

However, none of the above tools make any guarantees about the order of the changes involved in implementing a configuration change. When creating a new service, this is not normally an issue - we specify the requirements and the tool makes all the necessary changes (in some random order). When the tool has finished, we have a running system to our specification. However, if we are making configuration changes to an existing system, we may well care about the state of the configuration during the change; for example, if we want to make a transition from using one server, to using a different one, then we probably want to start the new server, and transfer the clients before shutting down the old one.

Such transitions are often performed manually - the administrator will work out a number of intermediate stages (server B started, clients all using server B, server A stopped), and check that each state has been achieved before presenting the tool with the next state. However, this is both time consuming, error prone, and not suitable for unattended use - for example where we want to make a configuration change "autonomically" in response to some failure or change in load.

One approach to this problem has been the use of manual workflow tools. These allow workflows such as the previous example to be captured and stored for automatic use - a particular workflow can then be invoked and the tool will take care of scheduling the separate stages in the given order. ControlTier [5] and IBM Tivoli Provisioning Manager [12] are examples which provide this kind of capability. However, this still requires that the workflows are computed manually. Even in a small system, a very large number of workflows can be required to cater for every eventuality - for example, moving from every possible failed state into a working state. And choosing an appropriate workflow to suit a particular goal state is not always obvious - indeed such manual workflows are conceptually similar to the imperative scripts which are no longer popular because of their unreliability.

An alternative approach is to make use of automated

planning technology to generate workflows "on the fly". This allows us to specify the current and goal states, together with a set of constraints on the intermediate stages - for example, all clients must always point at a working server. The intermediate states are then computed automatically, and these can then be presented in order to the configuration tool to effect a smooth transition.

This paper describes an experimental system which applies established AI planning tools to automatically generate workflows between declarative system states. The resulting intermediate states are implementable in Puppet and can be scheduled by ControlTier to produce a fully-automated system. We start (section 2) with a full "walk through" of a simple example, based on the server-transition problem described above. Section 3 then covers the background in more detail, including system configuration and automated planning technology. Section 4 describes the prototype system, section 5 presents some more complex examples, and section 6 concludes with a discussion of some of the problems and possible future directions.

2 An Example

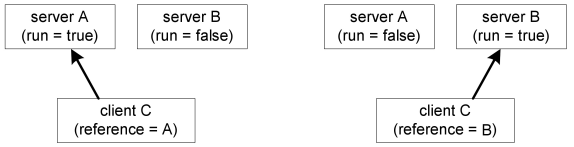
Assume we have a system consisting of two servers A and B, and one client C. Figure 1a shows the *current state*:

1. A.run = *true* (A is running),
2. B.run = *false* (B is stopped),
3. C.server = A (C is using a service of A).

The administrator aims to change the configuration to the *goal state* shown in figure 1b i.e.:

1. A.run = *false* (A is stopped),
2. B.run = *true* (B is running),
3. C.server = B (C is using a service of B).

Since C depends on the server's service, the changes must be implemented under a particular constraint i.e. C must always reference a running server.



(a) Current state (b) Goal state

Figure 1: States of the system.

If we use any declarative tool to implement these changes, then there are six possible sequences of states that could occur i.e.:

1. A.run = *false*, C.server = B, B.run = *true*;
2. C.server = B, A.run = *false*, B.run = *true*;
3. B.run = *true*, A.run = *false*, C.server = B;
4. A.run = *false*, B.run = *true*, C.server = B;
5. C.server = B, B.run = *true*, A.run = *false*;
6. **B.run = *true*, C.server = B, A.run = *false***.

Any of these sequences could appear in practice because the declarative tools implement the changes by executing the actions in an essentially indeterminate order. Unfortunately, only one of these sequences (#6), satisfies the required constraint while others do not. Hence, a declarative tool is highly likely to produce change sequence which leaves the system inoperative for a period of time during the change.

To address the problem, the automated planning technique used in our prototype creates the workflow automatically, based on the given goal states and the available actions. The prototype will generate a workflow which consists of a sequence of actions that satisfies an ordering constraint. Each action has *preconditions* which are constraints that have to be satisfied before executing the action, and *effects* which are states that will be attained after executing the action.

The prototype has the following actions pre-defined in the actions database:

1.
```
start-server
 parameters: <server>
 preconditions: <server>.run = false
 effects: <server>.run = true
```
2.
```
stop-server
 parameters: <server>
 preconditions:
  <server>.run = true
  (forall <client>
    <client>.server != <server>)
 effects: <server>.run = false
```
3.
```
change-reference
 parameters: <server1> <server2> <client>
 preconditions:
  <client>.server = <server1>
  <server2>.run = true
  <client>.server != <server2>
 effects: <client>.server = <server2>
```

To generate the workflow, the administrator (or some autonomic system) simply needs to declare the goal states:

1. C.server = B

2. A.run = *false*

Based on the above goal states and the available actions, our prototype generates the following ControlTier workflow:

```
<command name="config_changes"
 command-type="WorkflowCommand" description=""
 is-static="true" error-handler-type="FAIL">
  <workflow threadcount="1">
    <command name="start-server_B"/>
    <command name="reset-reference_A_B_C"/>
    <command name="stop-server_A"/>
  </workflow>
</command>

<command name="start-server_B" description=""
 command-type="Command" is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>start-server.pp B
   </argument-string>
</command>

<command name="change-reference_A_B_C"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>change-reference.pp A B C
   </argument-string>
</command>

<command name="stop-server_A" description=""
 command-type="Command" is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>stop-server.pp A
   </argument-string>
</command>
```

Submitting this workflow to ControlTier implements the configuration change using the valid sequence of actions (#6).

This example shows that the prototype is able to eliminate the sequencing problem that exists in declarative tools. Moreover, the prototype also simplifies the configuration tasks since it only requires the administrator to declare the goal states, and not the explicit workflow.

3 Background

This section summarises the approaches to system configuration discussed in the introduction, and surveys some background work on automated planning techniques. It concludes with a discussion of some related work in applying planning techniques to the configuration problem.

3.1 System Configuration

As noted in the introduction, approaches to system configuration have mostly evolved via the the following stages:

- **Manual configuration** - the administrator manually computes the actions necessary to change from one configuration to another, and then manually executes the commands necessary to implement this. Clearly, this is error prone, time-consuming, and it is difficult to prove reliably that a given sequence of changes will result in the required configuration under all circumstances.

- **Scripted changes** - similar to the previous approach, except that the sequence of changes is captured in an imperative script, allowing it to be executed multiple times, on different systems. This clearly makes it easier to deal with large numbers of systems, and until comparatively recently, this was probably the most common approach to configuration for many people. However, scripting still suffers from most of the problems of the manual approach. In particular, there is a tendency to blindly apply scripts to situations which do not meet the necessary preconditions, and the outcome can be very unpredictable.

- **Declarative tools** - currently, the most common approach in practice is probably to use a tool which allows a declarative specification of the desired state. The tool will then compute and implement the necessary changes (in an essentially indeterminate order). This guarantees that the resulting configuration matches the required specification, regardless of the starting point. As noted in the introduction, typical tools include Puppet [16], Cfengine [3], BCFG [4] and LCFG [1].

- **Fixed workflow orchestration** - in many cases, it is now essential to be able to perform sequences on configuration changes automatically, and/or unattanded, and use of fixed workflow tools is becoming more common. As noted in the introduction, ControlTier [5] and IBM Tivoli Provisioning Manager [12] are typical examples.

3.2 Automated Planning

Automated planning techniques generate a plan (workflow) automatically by computing a sequence of actions which will transition a system from some *initial state* to some required *goal state*. Each action has *preconditions* which are constraints that have to be satisfied before executing the action, and *effects* which are conditions that will be true after executing the action[1].

[1]Formally, a planning problem can be defined as $P = (\Sigma, s_i, S_g)$, where $\Sigma = (S, A, \gamma)$ is a state transition system, $s_i \in S$ is the initial state, and $S_g \subset S$ is a set of goal states, A is a set of actions, γ is a state transition function, find a sequence of action $\langle a_1, a_2, \ldots, a_k \rangle$ cor-

Practical implementations of automated planners use search algorithms to find an appropriate sequence of actions. There are several approaches to improving the efficiency of a simple brute-force search:

- **State-space planning** uses a subset of the state space as the search space where nodes correspond to the world states, arcs correspond to the state transitions and a path in the search space corresponds to the plan. The algorithms try to find a plan that satisfies the goal from the current state using particular heuristics to minimize the computing time. Metric-FF [10] and SGPlan [11] are examples of planners that use this approach.

- **Plan-space planning** uses the plan space as the search space where nodes are partially specified plans and arcs are the plan refinement operations intended to further complete a partial plan. The algorithms try to eliminate all the flaws in the initial partial plan which is either an unsatisfied sub-goal or a threat. The final plan will bring the system from the initial to the goal state. Planners that use this approach include UCPOP [15] and VHPOP [20].

- **Planning-Graph** uses a graph structure where nodes correspond to world state propositions and actions, and arcs correspond to preconditions and effects of actions. The algorithms expand the graph from the initial state until reaching the last layer that contains all goals which must not be mutually exclusive. The solution (plan) can be found by applying a backward-search algorithm from the last until reaching the first proposition layer. Graphplan planner [2] and LPG [8] are examples of planners that use this approach.

- **Hierarchical Task Network (HTN)** planning uses algorithms that decompose the given tasks using pre-defined methods until it reaches a set of primitive tasks and no non-primitive tasks remain. The tasks are organized as a collection called a task network which consists of a set of tasks and a set of constraints. O-Plan [19] is an example of planner that use this approach.

3.3 Related Works

There has been some previous works on the use of automated planning techniques for sequencing configuration changes in computing infrastructures. For example:

Keller et al. [13] introduced CHAMPS which translates the requested operations into a set of imperative tasks and organizes them as a workflow to satisfy the constraints as well as maximize the high degree of parallelism. However CHAMPS does not reason about the current state of target system as well as the preconditions and effects of each task which could lead to an unsound plan.

In [6], an object modelling language is used to specify the goal states and the operational capabilities of the configured components. The model is mapped in Planning Domain Definition Language (PDDL) [7] and given as the input to a POP (partial-order planning) planner which generates the workflow. Hagen [9] models the component life-cycle using CIM (Common Information Model) objects which are stored in a Configuration Management Database (CMDB). Based on the defined goal states, the state-space planner called LPS (Logical Planning Stat) then directly manipulates the objects in the CMDB to generate the workflow.

Both approaches demonstrate the viability of automated planning techniques for changes to the configuration of a computing infrastructure. They also provide very flexible solutions. However the modelling and the specifications are comparatively complex, and it is not clear how these might be exposed to end-users in a practically useful way. Levanti [14] provides a promising approach to simplifying the interface to the planner – the user is presented with a set of tags which hide much of the configuration details (states and operations). This enables the user to define and refine the goal state by iteratively selecting one or more tags. The workflow is generated by mapping the selected tags in SPPL (Stream Processing Planning LanguageL) [17] as the input for the SPPL planner [18].

We are not aware of any other work which meets our specific aims of using a standard planner to create a system which interfaces easily with common system configuration tools.

4 Prototype

We have developed a prototype implementation which combines an automated planner, together with ControlTier and Puppet to generate and execute plans for configuration changes. This prototype is definitely not (yet) a production-quality tool. However, our main aim has been to prove that the concepts would be applicable to a real environment, so the tool uses production-quality components, and is capable of generating practical workflows from specifications of realistic problems.

As illustrated in figure 2, the prototype consists of four main components i.e. actions database, translator, planner, and mapper. More details of the architectures' component are described as follows (each number represents the component's number in figure 2):

responding to a sequence of state transitions $\langle s_i, s_1, \ldots, s_k \rangle$ such that $s_1 = \gamma(s_i, a_1), s_2 = \gamma(s_1, a_2), \ldots, s_k = \gamma(s_{k-1}, a_k)$, and $s_k \in S_g$.

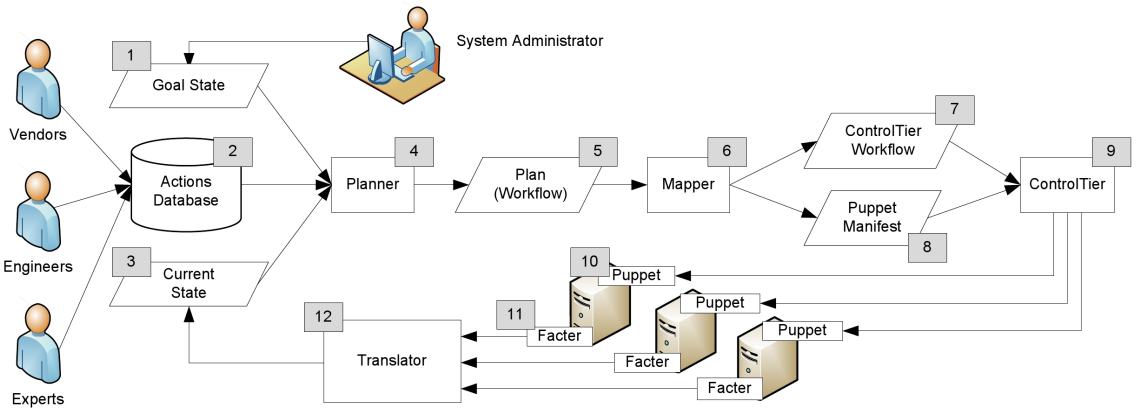


Figure 2: *System Architecture of The Prototype*

- The actions database (2) holds a library of actions, together with their required preconditions and effects. The actions can be written by anyone such as third-party software vendors, in-house software engineers, system administrators, or other specialists.
- A tool called facter (11) is used to acquire the current state (3) of the system. The outputs are aggregated by a translator (12) and then mapped into PDDL.
- The administrator specifies a declarative goal state (1) which is then mapped into PDDL.
- The planner (4) generates a plan (5) that will implement the new specification on the target system. This is based on the available predefined actions (from the actions database), the current state (from facter) and the goal state (supplied by the administrator).
- The mapper (6) uses the plan (5) to generate a ControlTier workflow-command (7) which consists of a set of other workflow-commands or primitive-commands. For each primitive-command, the mapper generates a puppet manifest file (8) to implement the action associated with the plan.
- ControlTier (9) manages the execution of the workflow by sending the puppet manifest file to the appropriate target node and requesting Puppet (10) to implement it.

In the latest prototype, we use LPG [8] as the planner. The main reason is that LPG can generate a plan (workflow) where all actions in each stage are mutually exclusive. This is an advantage because actions of each stage can be invoked in parallel by declarative tools to reduce the execution time. However, since we use the PDDL version 2.1 as the standard input for the planner, the prototype can utilize any available automated planner that supports it.

Currently, besides the translator, all parts have been implemented in Ruby and C. The implementation of the translator is straightforward i.e. translating the facts that are aggregated by the facter to a set of propositions in PDDL. For the actions database, all available actions are currently stored as individual files. If the number of actions were to become large, we could use a more structured database to improve storing and querying performance. For configuring other equipment (e.g. a router) , we could employ a proxy server as a bridge to communicate with the target equipment in order to implement the new specification and acquire its current state.

In the process of generating the workflow, the planner may use generic and domain-specific actions. A generic action, which is called as a "configuration pattern", is a reusable action which is applicable on any configuration problem. Whilst a domain-specific action is an action which is applicable to particular configuration problem. We will show the examples of these types of action in the experiments section.

An error could occur during the implementation of any part of the plan. For example, the "change-reference" action cannot be executed if the target server is broken. To address this problem, the prototype could be set to identify the error from the execution log and perform the re-planning process to compute an alternative plan in order to attain the same goal state. If the alternative plan exists, it will then be implemented on the target system. Otherwise, the prototype could ask the administrator to modify the goal state.

The prototype could also have a self-healing capability simply by evaluating the current and the goal state periodically. It will then generate and execute a plan for correcting any drift in the configuration of the system.

5 Experiments

5.1 Web Services

In the first experiment, we reconfigure a system consisting of two web services WS-A and WS-B, a client PC, and a firewall FW. Currently, PC is using a web service provided by WS-A through port 8080 of FW and WS-B is stopped. As shown in figure 3a, the system's *current state* is:

1. WS-A.run = *true*
2. WS-A.enable_firewall = *true*
3. WS-A.FW.port = 8080
4. WS-B.run = *false*
5. WS-B.enable_firewall = *true*
6. PC.service = WS-A
7. FW.ports(8080).open = *true*
8. FW.ports(9090).open = *false*

The administrator aims to shutdown WS-A for maintenance and redirect PC's reference to WS-B. This will change the configuration to the *goal state* shown in figure 3b which can be specified declaratively as follows:

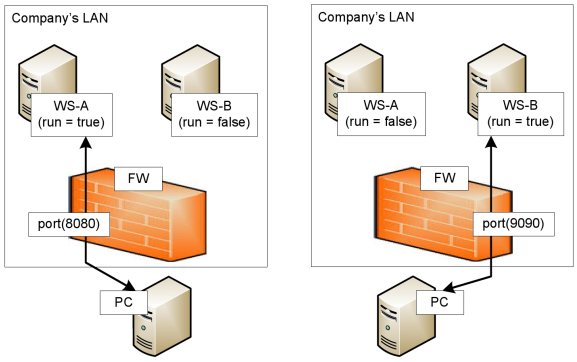
1. WS-A.run = *false*
2. PC.service = WS-B
3. WS-B.FW.port = 9090
4. FW.ports(8080).open = *false*

In addition, the administrator must satisfy the following constraints in the implementation of the changes:

1. The PC depends on the web service, thus it must always reference to a running web service.
2. Any unused port of F must be closed to minimize the vulnerability of the system.

To enable the planner to generate the right workflow, the first constraint is put in the preconditions of action *stop-service*. And the second one is declaratively specified in the goal state (#4). The following applicable actions are available in the actions database:

1.
```
start-service
 parameters: <service> <vm>
 preconditions:
  <service>.run = false
  <vm>.has = <service>
  <vm>.run = true
 effects: <service>.run = true
```



(a) Current state (b) Goal state

Figure 3: The states of the web services system.

2.
```
stop-service
 parameters: <service>
 preconditions:
  <service>.run = true
  (forall (<client>)
    <client>.service != <service>)
 effects: <service>.run = false
```

3.
```
open-fport
 parameters: <firewall> <port>
 preconditions:
  <firewall>.<port>.open = false
 effects:
  <firewall>.<port>.open = true
```

4.
```
close-fport
 parameters: <firewall> <port>
 preconditions:
  <firewall>.<port>.open = true
  (forall (<service>)
    <service>.<firewall>.port = <port>)
 effects:
  <firewall>.<port>.open = false
```

5.
```
assign-fport
 parameters: <service1> <firewall> <port>
 preconditions:
  <firewall>.<port>.open = true
  <service1>.enable_firewall = true
  <service1>.<firewall>.port != <port>
  (forall (<service2>)
    <service2>.<firewall>.port != <port>)
 effects:
  <service1>.<firewall>.port = <port>
```

6.
```
unassign-port
 parameters: <service> <firewall> <port>
 preconditions:
  <service>.<firewall>.port = <port>
  (forall (<client>)
    <client>.service != <service>)
 effects:
  <server>.<firewall>.port != <port>
```

7.
```
change-ref-fport
 parameters: <service1> <service2>
  <client> <firewall> <port>
 preconditions:
```

```
      <client>.service = <service1>
      <client>.service != <service2>
      <service2>.run = true
      <service2>.<firewall>.port = <port>
     effects:
      <client>.service != <service1>
      <client>.service = <service2>
```

By using information of the current and goal state with the application actions in the actions database, the prototype generated the following ControlTier workflow:

```
<command name="config_changes"
 command-type="WorkflowCommand" description=""
 is-static="true" error-handler-type="FAIL">
  <workflow threadcount="1">
   <command name="sub-workflow-1"/>
   <command name="assign-fport_WS-B_FW_P9090"/>
   <command name=
    "change-ref-fport_WS-A_WS-B_PC_FW_P9090"/>
   <command name="sub-workflow-2"/>
   <command name="close-fport_FW_P8080"/>
  </workflow>
</command>

<command name="sub-workflow-1"
 command-type="WorkflowCommand" description=""
 is-static="true" error-handler-type="FAIL">
  <workflow threadcount="2">
   <command name="start-service_WS-B_VM-B"/>
   <command name="open-fport_FW_P9090"/>
  </workflow>
</command>

<command name="sub-workflow-2"
 command-type="WorkflowCommand" description=""
 is-static="true" error-handler-type="FAIL">
  <workflow threadcount="2">
   <command name="stop-service_WS-A"/>
   <command name="unassign-fport_WS-A_FW_P8080"/>
  </workflow>
</command>
```

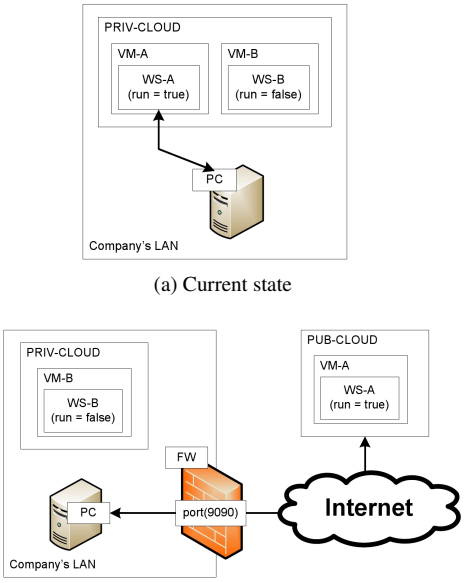
The prototype also generated the primitive ControlTier commands as shown in appendix B.

The generated workflow is a partial-order workflow which consists of one main workflow-command (*config_changes* and two sub-workflow-commands (*sub-workflow-1* and *sub-workflow-2*). *config_changes* is set to be executed by one thread to enforce the ordering constraint, i.e. a command must be invoked after the previous one has finished successfully. On the other hand, each sub-workflow-commands is set to be executed by two threads[2]to enable the parallel execution. This is possible since all commands of the sub-workflow-command are mutually exclusive.

To implement the changes, the workflow was submitted by the mapper to ControlTier which coordinated the execution of each commands. Puppet then used the appropriate manifest file to acheive the desired state.

[2]The number of threads is the same as the number of primitive commands.

5.2 Cloud Burst



(a) Current state

(b) Goal state

Figure 4: The states of the company's system before and after the cloud-burst scenario.

In the second experiment, we simulated the cloud-burst scenario on the computing infrastructure. In this scenario, an organization must dynamically deploy its software application from its limited internal computing resources to the public cloud in order to address a spike in demand.

We assumed a company has a private cloud infrastructure which runs various services to serve its 24-hours operations. One of them, WS-A which is running on virtual machine VM-A, is the most important web service since it processes all financial transaction from company's branch offices. Thus, the administrator has prepared a backup web service, WS-B which is installed on virtual machine VM-B, in case there is a failure on WS-A.

Unfortunately, due to the limited resource of the physical machines, the company's private cloud infrastructure is not capable of serving the spikes in demand which usually happens on the last three days of each month. Therefore, before the spike's period, the administrator plans to migrate WS-A temporarily to the public cloud to minimize its response time.

The migration of WS-A from the private to the public cloud is not an easy task since the administrator must satisfy the following constraints:

1. During the migration process, the service must always available for 24-hours a day without any down-time.

2. The company's firewall must be reconfigured to allow the LAN PCs to have connection with the server on public cloud.

3. The web service application cannot be installed on any other machines due to the limitation of the license.

Based on the above scenario, the current state of the system is illustrated in figure 4a which can be specified declaratively as:

1. VM-A.cloud = PRIV-CLOUD
2. VM-A.run = true
3. WS-A.on = VM-A
4. WS-A.run = true
5. PC.service = WS-A
6. VM-B.cloud = PRIV-CLOUD
7. VM-B.run = false
8. WS-B.on = VM-B
9. WS-B.run = false

Where *PRIV-CLOUD* and *PUB-CLOUD* are the private and public cloud infrastructure respectively.

To enable the cloud-burst, the system needs to achieve the goal state as illustrated in figure 4b. Therefore, the administrator can reconfigure the system using our prototype by declaring the goal state as:

1. VM-A.cloud = PUB-CLOUD
2. WS-A.FW.port = 8080
3. PC.service = WS-A
4. VM-B.cloud = PRIV-CLOUD
5. VM-B.run = false

Where FW is the name of the company's firewall.

Fortunately, to generate the workflow, we only need to add five actions to the actions database since the planner can reuse the actions from the previous examples. The five new actions are:

1. `start-vm`
```
parameters: <vm> <cloud>
preconditions:
 <cloud>.has = <vm>
 <vm>.run = false
effects:
 <vm>.run = true
```

2. `stop-vm`
```
parameters: <vm>
preconditions:
 <vm>.run = true
 (forall <service>
   if <vm>.has = <service>
   then <service>.run = false)
effects:
 <vm>.run = false
```

3. `change-ref`
```
parameters: <service-1> <service-2> <client>
preconditions:
 <client>.service = <service-1>
 <client>.service != <service-2>
 <service-2>.run = true
 <service-2>.enable_firewall = false
effects:
 <client>.service != <service-1>
 <client>.service = <service-2>
```

4. `migrate`
```
parameters: <vm> <cloud-1> <cloud-2>
preconditions:
 <vm>.run = false
 <cloud-1>.has = <vm>
 <cloud-2>.is_public = true
effects:
 !(<cloud-1>.has = <vm>)
 <cloud-2>.has = <vm>
```

5. `set-need-firewall`
```
parameters: <service>
preconditions:
 (forall (<firewall> <port>)
   <service>.<firewall>.port != <port>)
 (forall (<vm> <cloud>)
   if (<vm>.has = <service>
       and <cloud>.has = <vm>)
   then <cloud>.is_public = true)
effects:
 <service>.enable_firewall = true
```

Some of these reusable actions, such as *stop-service* and *start-service*, typically occur in many different situations and form a set of generic *patterns*.

After processing the information, the planner will give its output to the mapper which generated the following ControlTier workflows:

```
<command name="config_changes"
 command-type="WorkflowCommand" description=""
 is-static="true" error-handler-type="FAIL">
  <workflow threadcount="1">
   <command name="sub-workflow-1"/>
   <command name="start-service_WS-B_VM-B"/>
   <command name="change-ref_WS-A_WS-B_PC"/>
   <command name="stop-service_WS-A"/>
   <command name="stop-vm_VM-A"/>
   <command name=
    "migrate_VM-A_PRIV-CLOUD_PUB-CLOUD"/>
   <command name="sub-workflow-2"/>
   <command name="sub-workflow-3"/>
   <command name=
    "change-ref-fport_WS-B_WS-A_PC_FW_P8080"/>
   <command name="stop-service_WS-B"/>
```

```
   <command name="stop-vm_VM-B"/>
  </workflow>
</command>

<command name="sub-workflow-1"
 command-type="WorkflowCommand" description=""
 is-static="true" error-handler-type="FAIL">
  <workflow threadcount="2">
   <command name="open-fport_FW_P8080"/>
   <command name="start-vm_VM-B_PRIV-CLOUD"/>
  </workflow>
</command>

<command name="sub-workflow-2"
 command-type="WorkflowCommand" description=""
 is-static="true" error-handler-type="FAIL">
  <workflow threadcount="2">
   <command name="set-need-firewall_WS-A"/>
   <command name="start-vm_VM-A_PUB-CLOUD"/>
  </workflow>
</command>

<command name="sub-workflow-3"
 command-type="WorkflowCommand" description=""
 is-static="true" error-handler-type="FAIL">
  <workflow threadcount="2">
   <command name="assign-fport_WS-A_FW_P8080"/>
   <command name="start-service_WS-A_VM-A"/>
  </workflow>
</command>
```

The prototype also generated the primitive ControlTier commands as shown in appendix C.

The planner generated the partial-order plan (workflow) which has three sub-workflows. Each sub-workflow has a set of commands that can be run in parallel due to their a mutual exclusive property. Submission the workflows to ControlTier implemented the new configuration specification that enable WS-A servicing more clients than before.

If the administrator would like to stop using the public cloud, WS-A can be migrated back to the private cloud by easily changing the goal state of the system. The prototype will generate and execute automatically the workflow to implement the new specification. In this case, we can also have a full autonomic configuration tool by replacing the administrator with an autonomic agent which will automatically trigger the migration of the system from private to the public cloud or vice versa based on the demands.

6 Conclusions

This work has clearly demonstrated the advantages of automated planning for system reconfiguration – workflows can be automatically generated (providing that a solution exists) between any two declarative states, enabling unattended, autonomic reconfiguration for failure recovery or other reasons. The generated workflows are guaranteed (by design) to achieve the desired target state, at the same time as preserving any necessary properties of the system during the reconfiguration. We have also shown that it is possible to build a practical tool which generates workflows automatically, and uses existing production-quality tools for the deployment (as well as the planning).

However, we suspect that the usability of such systems will be a major challenge – firstly, languages and interfaces are required to enable working administrators to easily translate their requirements and specifications into a form that is usable by the planners. Secondly, administrators need to have confidence that the system will behave in a predictable way – planners are very good at exploiting a lack of precision in the specification to find very "creative" and unexpected solutions! The human interaction aspects of this problem are something which would benefit from future work.

Error recovery is also a very important area. Reconfigurations often occur in precisely those situations where the system itself is unreliable – for example, during network and components failures, or system overload. Plans are likely to fail at some intermediate stage, or a centralised planner may become disconnected and lose track of the current state of an executing plan.

7 Future Work

We are currently interested in investigating more distributed, and localised approaches to automated planning for configuration changes. This will allow more autonomy for individual components (thus improving the resilience) and break the planning problem into a hierarchy of problems which are easier to understand and predict.

We believe that our implementation is much closer than previous work to providing a practical solution for system administrators who are familiar with current configuration tools such as Puppet. However, most administrators would still be unhappy to allow significant changes to their infrastructure by a completely automated system - the chances of unexpected and inappropriate solutions are still too high. We believe that there is considerable scope here for further work on appropriate languages and interfaces - perhaps involving *mixed initiative* solutions which combine automated planning with human guidance, and automated explanations of proposed solutions.

8 Acknowledgments

The authors like to thank Andrew Farrell from HP Labs Bristol for his valuable contributions. This research is fully supported by a grant from 2010 HP Labs Innovation Research Program Award.

References

[1] ANDERSON, P., AND SCOBIE, A. LCFG: The next generation. In *UKUUG Winter Conference* (2002).

[2] BLUM, A., AND FURST, M. Fast Planning through Planning Graph Analysis. *Artificial Intelligence 90* (1997), 281–300.

[3] CFENGINE AS. Cfengine - Automatic Server Lifecycle Management, 2011.

[4] DESAI, N., LUSK, A., BRADSHAW, R., AND EVARD, R. BCFG: A Configuration Management Tool for Heterogeneous Environments. In *Proceedings of IEEE International Conference on Cluster Computing* (2003), IEEE Computer Society.

[5] DTO SOLUTIONS. ControlTier, 2011.

[6] EL MAGHRAOUI, K., MEGHRANJANI, A., EILAM, T., KALANTAR, M., AND KONSTANTINOU, A. 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* (2006), pp. 404–423.

[7] FOX, M., AND LONG, D. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research 20*, 1 (2003), 61–124.

[8] GEREVINI, A., AND SERINA, I. LPG: A planner based on local search for planning graphs with action costs. In *Proceedings of the Sixth Internatinal Conference on AI Planning and Scheduling* (2002), pp. 12–22.

[9] 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 *2010 IEEE 3rd International Conference on Cloud Computing* (2010), pp. 11–18.

[10] HOFFMANN, J. The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *Journal of Artificial Intelligence Research 20*, 20 (2003), 291–341.

[11] HSU, C., WAH, B., HUANG, R., AND CHEN, Y. Constraint partitioning for solving planning problems with trajectory constraints and goal preferences. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07), Hyderabad, India* (2007).

[12] IBM CORP. Integrated Service Management software, IBM Tivoli, 2011.

[13] KELLER, A., HELLERSTEIN, J., WOLF, J., WU, K., AND KRISHNAN, V. The CHAMPS system: Change management with planning and scheduling. In *Network Operations and Management Symposium, 2004. NOMS 2004. IEEE/IFIP* (2004), vol. 1, pp. 395–408.

[14] LEVANTI, K., AND RANGANATHAN, A. Planning-based configuration and management of distributed systems. In *Integrated Network Management, 2009. IM'09. IFIP/IEEE International Symposium on* (2009), pp. 65–72.

[15] PENBERTHY, J., AND WELD, D. UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of the 3rd International Conference on Knowledge Representation and Reasoning* (1992), pp. 103–114.

[16] PUPPET LABS. Puppet, 2011.

[17] RIABOV, A., AND LIU, Z. Planning for stream processing systems. In *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 3* (2005), vol. 20, pp. 1205–1210.

[18] RIABOV, A., AND LIU, Z. Scalable planning for distributed stream processing systems. In *Proceedings of ICAPS* (2006).

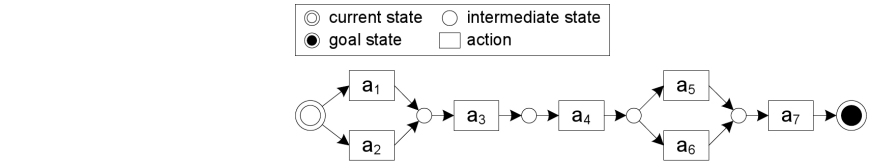
[19] TATE, A., DALTON, J., AND LEVINE, J. O-Plan: A web-based AI planning agent. In *Proceedings of the National Conference on Artificial Intelligence* (2000), pp. 1131–1132.

[20] YOUNES, H., AND SIMMONS, R. VHPOP: Versatile heuristic partial order planner. *Journal of Artificial Intelligence Research 20*, 1 (2003), 405–430.

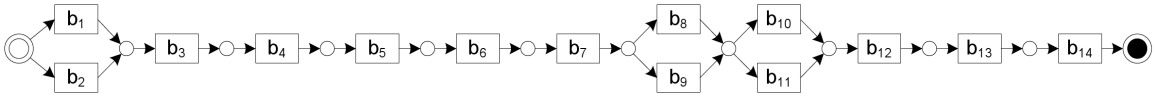
A The Flow-Chart of The Workflows

Figure 5a and 5b illustrate the flow-charts of the generated workflows of web services and cloud burst examples. Each actions are associated with a ControlTier command as follows:

- a_1: `start-service_WS-B_VM-B`
- a_2: `open-fport_FW_P9090`
- a_3: `assign-fport_WS-B_FW_P9090`
- a_4: `change-ref-fport_WS-A_WS-B_PC_FW_P9090`
- a_5: `stop-service_WS-A`
- a_6: `unassign-fport_WS-A_FW_P8080`
- a_7: `close-fport_FW_P8080`
- b_1: `open-fport_FW_P8080`
- b_2: `start-vm_VM-B_PRIV-CLOUD`
- b_3: `start-service_WS-B_VM-B`
- b_4: `change-ref_WS-A_WS-B_PC`
- b_5: `stop-service_WS-A`
- b_6: `stop-vm_VM-A`
- b_7: `migrate_VM-A_PRIV-CLOUD_PUB-CLOUD`
- b_8: `set-need-firewall_WS-A`
- b_9: `start-vm_VM-A_PUB-CLOUD`
- b_{10}: `assign-fport_WS-A_FW_P8080`
- b_{11}: `start-service_WS-A_VM-A`
- b_{12}: `change-ref-fport_WS-B_WS-A_PC_FW_P8080`
- b_{13}: `stop-service_WS-B`
- b_{14}: `stop-vm_VM-B`



(a) The Workflow of Web Services



(b) The Workflow of Cloud Burst

Figure 5: The flow-chart of the workflows.

B Primitive ControlTier Commands of Web Services

```
<command
 name="start-service_WS-B_VM-B"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>stop-service.pp WS-B VM-B
   </argument-string>
</command>

<command name="open-fport_FW_P9090"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>stop-open-fport.pp FW 9090
   </argument-string>
</command>

<command name="assign-fport_WS-B_FW_P9090"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>assign-fport.pp WS-B FW 9090
   </argument-string>
</command>

<command
 name="change-ref-fport_WS-A_WS-B_PC_FW_P9090"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>change-ref-fport.pp WS-A WS-B
   PC FW 9090</argument-string>
</command>

<command name="stop-service_WS-A"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>stop-service.pp WS-A
   </argument-string>
</command>

<command name="assign-fport_WS-A_FW_P8080"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>unassign-fport.pp WS-A FW 8080
   </argument-string>
</command>

<command name="close-fport_FW_P8080"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>close-fport.pp FW 8080
   </argument-string>
</command>
```

C Primitive ControlTier Commands of Cloud-Burst

```
<command name="open-fport_FW"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>open-fport.pp FW 8080
   </argument-string>
</command>

<command name="start-vm_VM-B_PRIV-CLOUD"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>start-vm.pp B PRIV-CLOUD
   </argument-string>
</command>

<command name="start-service_WS-B_VM-B"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>start-service.pp WS-B VM-B
   </argument-string>
</command>

<command name="change-ref_WS-A_WS-B_PC"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>change-ref.pp WS-A WS-B PC
   </argument-string>
</command>
```

```
<command name="stop-service_WS-A"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>stop-service.pp WS-A
   </argument-string>
</command>

<command name="stop-vm_VM-A"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>stop-vm.pp VM-A
   </argument-string>
</command>

<command
 name="migrate_VM-A_PRIV-CLOUD_PUB-CLOUD"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>migrate.pp VM-A PRIV-CLOUD
   PUB-CLOUD</argument-string>
</command>

<command name="set-need-firewall_WS-A"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>set-need-firewall.pp
   WS-A</argument-string>
</command>

<command name="start-vm_VM-A_PUB-CLOUD"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>start-vm.pp VM-A PUB-CLOUD
   </argument-string>
</command>

<command name="assign-fport_WS-A_FW_P8080"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>assign-fport.pp WS-A FW 8080
   </argument-string>
</command>

<command name="start-service_WS-A_VM-A"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>start-service.pp WS-A VM-A
   </argument-string>
</command>

<command
 name="change-ref-fport_WS-B_WS-A_PC_FW_P8080"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>change-ref-fport.pp WS-A WS-A
   PC FW 8080</argument-string>
</command>

<command name="stop-service_WS-B"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>stop-service.pp WS-B
   </argument-string>
</command>

<command name="stop-vm_VM-B"
 description="" command-type="Command"
 is-static="true">
  <execution-string>exec.rb</execution-string>
  <argument-string>stop-vm.pp VM-B
   </argument-string>
</command>
```