View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Edinburgh Research Explorer

THE UNIVERSITY *of* EDINBURGH

Edinburgh Research Explorer

Choreographing Configuration Changes

Citation for published version:
Herry, H, Anderson, P & Rovatsos, M 2013, Choreographing Configuration Changes. in Proceedings of the 9th International Conference on Network & Service Management (CNSM). IEEE Computer Society, pp. 156-160. DOI: 10.1109/CNSM.2013.6727828

Digital Object Identifier (DOI):
10.1109/CNSM.2013.6727828

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Proceedings of the 9th International Conference on Network &amp; Service Management (CNSM)

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

OPEN ACCESS

Download date: 05. Apr. 2019

Herry, H., Anderson, P., & Rovatsos, M. (2013). Choreographing Configuration Changes. In Proceedings of the 9th International Conference on Network & Service Management (CNSM). IEEE Computer Society. 10.1109/CNSM.2013.6727828

Choreographing configuration changes

Herry Herry*, Paul Anderson†, and Michael Rovatsos‡
School of Informatics
University of Edinburgh
Edinburgh, UK
*h.herry@sms.ed.ac.uk, †dcspaul@ed.ac.uk, ‡mrovatso@inf.ed.ac.uk

Abstract—**This paper describes the automatic generation of a set of reactive agents capable of autonomously reconfiguring a computing infrastructure into a specified goal state. The agent interactions are guaranteed to be deadlock/live-lock free, can preserve pre-specified global constraints during their execution, and autonomically maintain the goal state once it has been achieved. We describe novel algorithms for the generation and execution of the agent model, and evaluate the results on some realistic problems, using a prototype implementation.**

I. Introduction

We have previously shown [1] that automated planning techniques can be used to generate the workflows necessary to reconfigure large computing installations. These workflows ensure the achievement of the required, declarative goal state, and guarantee the maintenance of any necessary global constraints throughout the reconfiguration. In our previous work, the workflows were *orchestrated* by a central controller. However, this creates a potential bottleneck in large systems, and can also be unreliable if the communication with the controller is disrupted, which is particularly relevant since reconfiguration frequently occurs as an autonomic response to system failures.

Fully distributed planning, on the other hand, is not a good solution to this problem either – avoiding deadlock/livelock may require agents to have considerable global knowledge and achieving such global knowledge is likely to result in even more costly inter-agent communication. Predicting the behaviour of such systems is also more difficult and hence they are less acceptable to system administrators in real situations.

In this paper, we present a novel solution which aims to avoid the drawbacks of both these extremes: the workflow generated by the planner is used to automatically construct a set of purely reactive agents which *choreograph* the execution of the workflow without the need for a central controller. This combination of centralised planning with distributed execution provides robust, autonomous execution while retaining the advantages of a predictable, deadlock-free workflow. Additionally, the agents form a self-healing system by continuously attempting to maintain the goal state. We have implemented this process by modifying the Nuri workflow planner [1] to generate and deploy a reactive agent model based on *Behavioural Signatures* [2], [3]. Our evaluation results show that the Nuri agents can achieve the goal state without any central control, while maintaining the global constraints throughout the changes. Moreover, each agent is capable of repairing drifts from the goal state, autonomously, and without any re-planning process.

Section II describes the background to the problem in more detail, and section III illustrates the proposed solution by working through an example. Sections IV and V describe the algorithms for the model generation and execution respectively. Sections VI and VII describe the implementation and its evaluation on some concrete examples.

II. Background & Related Work

The scale and complexity of modern computing infrastructures demand an automated approach to the management of their configuration, and tools such as Puppet [4] and Chef [5] are now ubiquitous. Many of these tools (Puppet, LCFG [6], BCFG [7]) adopt a *declarative* approach which allows the explicit specification of the desired end-state of the system – the tool then computes the necessary workflow to achieve that state. One disadvantage of this approach is that the user has no control over the generated workflow which may contain intermediate states that violate essential constraints [8]. An alternative approach is to specify the workflow manually (an approach used, e.g., by IBM Tivoli Provisioning Manager [9], Microsoft System Center [10], Juju [11] and RunDeck [12]) – however, this requires a separate workflow for each initial/final state pair, and the resulting configuration needs to be verified against the requirements for the final state. We have previously shown [1] that a declarative approach, combined with an automated planner, can generate workflows which achieve the desired state while maintaining the global constraints throughout the configuration process.

Most practical configuration tools are highly centralised – a central controller gathers information about the state of the system and orchestrates the workflow by communicating directly with the systems involved at each step. SmartFrog [2] is an interesting exception, where the components which manage the various aspects of the system can be augmented with *Behavioural Signatures* [3]. These allow the administrator to specify a model which defines state-dependencies between components so that a change of state in one component may depend on changes of state in other components. This can produce a cascade of distributed state changes with a particular ordering constraint. Figure 1 illustrates an example which manages a 3-tier web application deployed on several virtual machines. Unfortunately, the dependencies must be computed manually which is error-prone and time-consuming.

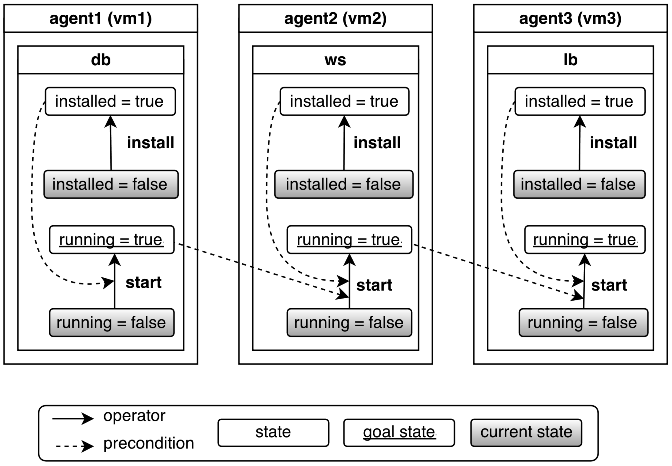

Fig. 1: The Behavioural Signature model for deploying the example 3-tier web application system.

The resulting models must also be validated for deadlock and livelock conditions before they can be deployed.

There has been some previous work on distributed workflow execution using multi-agent systems: [13], [14], [15] describe a fully distributed execution framework where the workflow is represented as interaction models of a multi-agent system, each consisting of roles and constraints. But as with [3], the model has to be computed manually. [16] describes a control loop technique to monitor the execution of a workflow in a multi-agent system. In [17], the workflow is compiled as a compact representation based on Simple and Disjunctive Temporal Constraints Networks. These works have a common where the workflow is executed in progressive.

III. Example

Assume we wanted to deploy a 3-tier web application consisting of a load balancer, a web service, and a database service, onto three virtual machines (VMs) on a public cloud. Each VM has an agent that controls some components, and manages the configuration of the VM including the installed software. In this case, there are three agents: $agent_1, agent_2$, and $agent_3$, and there are six components: vm_1, vm_2, vm_3, lb, ws, and db that manage the configuration of VM-1, VM-2, VM-3, a load balancer, a web service, and a database service respectively.

Although the software will be deployed independently on different VMs, the deployment process must satisfy some global constraints due to service dependencies:

- The web service depends on the database service: whenever the web service is running then the database service must be running as well;
- the load balancer depends on the web service: whenever the load balancer is running then the web service must be running as well.

The following sections describe the steps required to automatically deploy the example system with Nuri.

A. Modelling the System

To model the system, we use an object-oriented configuration language called SFP which is introduced in [1]. SFP is a prototype-based language that allows us to model a resource as an object. An object has a set of attributes, each of which is assigned a value. The state of an object is represented by the collection of attribute-value pairs, and the union of all objects' states represents the state of the system. An object may also have one or more procedures, each representing the capability of an object to change its state or another object's state by modifying the attribute values.

SFP also allows us to define a "loose" desired state as a logical formula. This not only allows the administrator to work at a higher level by defining a set of possible desired states as a constraint, but it also affords the planner more flexibility in searching for the best solution. In addition, SFP allows us to define global constraints as logical formulae – these are constraints which must be satisfied at every stage during a sequence of reconfiguration steps.

SFP provides a notation to define a *schema*, which is a contractual structure of the objects that implement it. This ensures the consistency of attributes (name and type) and procedures across objects conforming to the same schema. For our example, we define five abstract schemata as follows[1]:

```
schema VM { created = false }
schema Service {
 installed = false; running = false
 procedure install {
  cost = 10
  conditions { this.installed = false }
  effects { this.installed = true }}
 procedure start {
  cost = 5
  conditions { this.installed = true;
               this.running = false }
  effects { this.running = true }}
 ...
}
schema Database extends Service
schema WebService extends Service
schema LoadBalancer extends Service
```

The current state of the example system can be modelled as follows:

```
vm1 isa VM {
 created = true
 db isa Database { installed = false;
                  running = false }}
vm2 isa VM {
 created = true
 ws isa WebService { installed = false;
                    running = false }}
vm3 isa VM {
 created = true
 lb isa LoadBalancer { installed = false;
                      running = false }}
```

The above model shows that there are three VMs: vm_1, vm_2, and vm_3, and each has software component db, ws, and lb respectively. All of the VMs exist, but only one of the software components is (yet) installed.

[1] For brevity, we omit the procedures *uninstall* and *stop* of the *Service* schema.

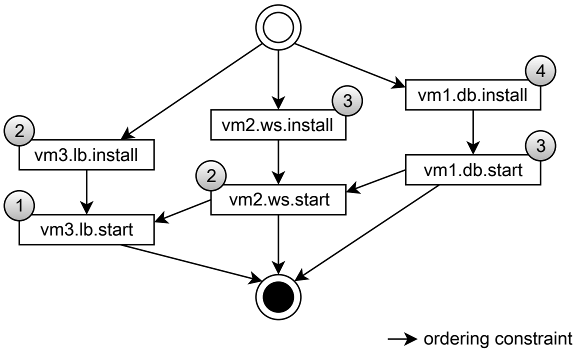

Fig. 2: The workflow for deploying the example 3-tier web application. Numbered circles contain priority index value of a particular sub-procedure (see section IV).

B. Planning the Configuration Changes

The deployment of the example system involves a sequence of configuration steps that select and execute procedures to change attribute values to achieve certain objectives, moving towards the desired state, and preserving the pre-defined global constraints while the changes are made. To achieve this, we employ a technique described in [1] which compiles the above model together with the specification of goal and global constraints into a classical planning problem. We then use an off-the-shelf planner to generate the workflow.

Before employing this technique, we must define the configuration task which consists of an initial state, a goal state, and a global constraint. The initial state is represented by the above model, while the goal and global constraints can be defined in SFP as follows:

```
goal {
 vm1.db.running = true
 vm2.ws.running = true
 vm3.lb.running = true
}
global {
 if vm3.lb.running = true then vm2.ws.running = true
 if vm2.ws.running = true then vm1.db.running = true
}
```

By compiling this configuration task to a classical planning problem, and submitting the compilation result to the planner, we obtain the workflow depicted in figure 2.

C. Implementing the Configuration Changes

To implement the configuration changes, the workflow is used to automatically construct and deploy a set of distributed components which implement a Behavioural Signature (BSig) model. These components choreograph the changes autonomously, achieving the desired state and preserving the global constraints while avoiding reliance on any central controller. The details of this translation process are described in section IV.

Figure 1 shows the generated BSig model for the example system, based on the workflow generated in the planning step (figure 2). In using this model, each agent applies an execution algorithm called *reactive-regression* that always selects and invokes an operator that can be used to perform a transition toward the goal state. Before invoking an operator, though, the agent must satisfy any preconditions by selecting and invoking appropriate local operators, and/or sending a request to another agent to achieve particular goals described in the preconditions. This algorithm is described in section V

The execution of the example BSig model can be summarised as follows:

1) Based on the model, $agent_3$ detects a goal-flaw i.e. `vm3.lb.running = true`. It selects operator `vm3.lb.start` to repair the flaw. Since this operator has a local precondition `vm3.lb.installed = true`, then $agent_3$ selects operator `vm3.lb.install`. $agent_3$ directly invokes this operator since its precondition has been satisfied. Afterwards, $agent_3$ sends a goal request to $agent_2$ to satisfy a remote precondition `vm2.ws.running = true`, and waits for the reply.
2) Concurrently, $agent_2$ detects a goal-flaw i.e. `vm2.ws.running = true`. It selects operator `vm2.ws.start` to repair the flaw. Since this operator has a local precondition `vm2.ws.installed = true`, then $agent_2$ selects operator `vm2.ws.install`. $agent_2$ directly invokes this operator since its precondition has been satisfied. Afterwards, $agent_2$ sends a goal request to $agent_1$ to satisfy a remote precondition `vm1.db.running = true`, and waits for the reply. In this step, it postpones $agent_3$'s request since it has not yet achieved state `vm2.ws.running = true`.
3) Concurrently, $agent_1$ detects a goal-flaw i.e. `vm1.db.running = true`. It selects operator `vm1.db.start` to repair the flaw. Since this operator has a local precondition `vm1.db.installed = true`, then $agent_1$ selects operator `vm1.db.install`. $agent_1$ directly invokes this operator since its precondition has been satisfied. Afterwards, $agent_1$ invokes operator `vm1.db.start` since its preconditions have been satisfied. Since it receives a request from $agent_2$, it sends an acknowledgement.
4) $agent_2$ recevies a *success* reply from $agent_3$ and will continue to invoke operator `vm2.ws.start` since its precondition has been satisfied. Afterwards, it sends an acknowledgement to $agent_1$.
5) Since $agent_1$ receives a *success* reply from $agent_2$, it continues to invoke operator `vm3.lb.start` as its precondition has been satisfied.
6) The execution of operator ends here since all agents have achieved their goals.

The sequence diagram of this execution is available on: http://homepages.inf.ed.ac.uk/s0978621/cnsm13/sequence.pdf.

In executing the model, each agent is communicating purely *peer-to-peer* with other agents. Communication is initiated when an agent needs to invoke an operator which has preconditions which can only be satisfied by other agents. Based on the replies, an agent can decide whether the selected operator may be invoked or not.

IV. Choreographing the Model

The choreography aims to define a “global scenario” which should be executed by all agents during configuration changes without any single point of control. This global scenario is a workflow generated by the planning step. If only one agent is involved in this scenario, then the execution can be performed sequentially in a straightforward way. If, however, the scenario involves more than one agent, then it must be split up into local scenarios for each agent. We refer to each of these as a *local BSig model* which defines the agent’s local goal and specifies *which* local changes can be made *under what circumstances*. We will refer to the goal of a single agent as a *local goal*, and to the goal of the whole system as the *system goal*.

A. Local Goal

Each local model may have a goal state, which can be defined as follows:

Definition 1: A local goal g of an agent is defined as a set of variable assignments, each of the form $v = d$, where v is a combination of variable name and namespace and d is a value.

For example, the workflow in figure 2 involved three procedures: `vm1.db.start`, `vm2.ws.start`, and `vm3.lb.start`. They support the system goal states `vm1.db.running = true`, `vm2.ws.running = true`, and `vm3.lb.running = true`, respectively. From the namespace of those procedures and the goals they support, we can determine that

- $g_{agent_1} = \{$`vm1.db.running = true`$\}$,
- $g_{agent_2} = \{$`vm2.ws.running = true`$\}$, and
- $g_{agent_3} = \{$`vm3.lb.running = true`$\}$.

In this case, we could have achieved the same result by only considering the namespace of each variables of the system goal state, however there are some situations where this is insufficient. For example, assume that we have two agents $agent_x$ and $agent_y$, and the first agent controls the creation of a new VM on a public cloud infrastructure through procedure `cloud.create_vm`. The second agent controls a virtual machine vm_y, but does not have the VM creation capability. This implies that a goal of `vmy.created = true` should be assigned to $agent_x$ instead of $agent_y$.

B. Local Operators

Each local BSig model may also have a set of local operators that determine *what* local changes can be made *when*. The definition of such a local operator is as follows:

Definition 2: A local operator o_j of a Behavioural Signature model is defined as a 4-tuple $o_j = \langle name, pre, post, p\rangle$ where:

- $name$ is a combination of namespace with operator name,
- pre and $post$ are the precondition and postcondition of the operator, each of which is a set of pairs $v = d$ assigning value d to variable v,
- p is an integer value that represents the priority index of operator o_j compared to other operators.

The local operators of each agent can be obtained from the workflow by considering the namespace associated with every procedure. For example, procedure `vm2.ws.start` of the workflow in figure 2 has namespace `vm2.ws`. By considering this namespace, we can assign the procedure to agent α_2 as one of its local operators.

Each local operator should have a precondition, which is a partial state description that must be satisfied before executing the operator, and a postcondition (a partial description of the state after execution). This information can be obtained from two sources: the grounded procedure specification and the ordering constraints of the workflow. In our example, based on procedure `start` of schema `Service` inherited from component `ws` of vm_2, the precondition and postcondition of operator `vm2.ws.start` are[2]:

- precondition:
  $\{$`vm2.ws.running = false`, `vm2.ws.installed = true`$\}$
- postcondition:
  $\{$`vm2.ws.running = true`$\}$

Based on the workflow’s ordering constraints (shown as arrows in figure 2), procedure `vm2.ws.start` must be executed after `vm2.ws.install` and `vm1.db.start`. Both predecessor procedures have the following postconditions:

- postcondition of `vm2.ws.install`:
  $\{$`vm2.ws.installed = true`$\}$
- postcondition of `vm1.db.start`:
  $\{$`vm1.db.running = true`$\}$

To ensure that the ordering constraints are satisfied, these postconditions are added to the preconditions of `vm2.ws.start`. Thus, its final precondition and postcondition are:

- precondition:
  $\{$`vm2.ws.running = false`, `vm2.ws.installed = true`, `vm1.db.running = true`$\}$
- postcondition:
  $\{$`vm2.ws.running = true`$\}$

As shown above, this technique injects some additional constraints into a particular operator’s precondition in order to maintain the global constraint during execution. This ensures that the model is free of global conflicts, even though its execution is distributed.

During execution, an agent may have a choice of more than one operator that can be selected to repair some flaws, i.e. violated goal conditions. Instead of selecting an arbitrary operator, the agent should select one of the operators that has the lowest priority index. The index of every operator is obtained by taking the maximum of its successor operator indices and incrementing this value by one (operators without successors are assigned an index of 1). i.e:

$$pi(o_j) = \begin{cases} 1 & \text{if } Succ_j = \emptyset \\ max(pi(Succ_j)) + 1 & \text{if } Succ_j \neq \emptyset \end{cases} \qquad (1)$$

where $Succ_j$ is the set of successor operators of o_j. For the workflow in our example system, the values in the circles in figure 2 represent the priority index of the operators involved.

Priority index values are very important because they reflect the ordering constraints between operators as specified by the

[2] The keyword `this` in a SFP procedure refers to the parent object of the procedure.

workflow generated by the planner. Our execution algorithm will use these priority index values to ensure that there is no deadlock or livelock situation during execution.

C. System BSig Model

To complete our definition of the overall configuration model, we can now define the BSig model of the whole system $\mathcal{M}$ is the union of all local models involved:

Definition 3: A local model is a tuple $m_i = \langle \mu_i, g_i, \mathcal{O}_i \rangle$, where g_i is the local goal, $\mathcal{O}_i$ is a set of local operators of $agent_i$, and μ_i is the model's serial number[3]

Definition 4: A Behavioural Signature model of a system is a tuple $\mathcal{M} = \langle A, M \rangle$, where A is a set of agents, $M = \bigcup_{\forall ag \in A} \{m_{ag}\}$, and $\forall m_i, m_j \in M \,.\, \mu_i = \mu_j$.

To create this model manually would involve defining the precondition of each operator in order to preserve global constraints, and then proving that 1) the model would achieve the desired state, 2) the model would preserve the global constraints, and 3) there is no potential deadlock or livelock. These properties could be verified using model checking techniques, but any issues would need to be diagnosed and corrected manually. Since we generate the model automatically from the workflow, these properties are satisfied by definition.

V. Executing the Model

A model $\mathcal{M} = \langle A, M \rangle$ is deployed by sending each local model $m_i \in M$ to $agent_i \in A$. Each agent that receives a local model will stop execution of any current model, and start to excite the received one.

To execute the model, we uses a novel algorithm called *reactive regression*. It is *reactive* since it continuously tries to find and repair the flaws in the local model by comparing the local goal with the current state. It involves *regression* because it always tries to select and invoke the nearest operator to the goal state in order to repair existing flaws, and recursively invokes other local operators and/or sends a new goal to another agent in order to satisfy the precondition of the selected operator. In other words, the algorithm tries to execute the workflow using a distributed backward-chaining method, which may produce distributed, cascading effects of configuration changes if the execution involves multiple agents.

Each agent performs this algorithm using one main thread and a set of satisfier threads. The main thread is responsible for continuously finding and repairing any goal flaws. The satisfier threads are responsible for receiving and achieving a goal from another agent, and sending back information about their local status.

After receiving a local model $m = \langle \mu, g, \mathcal{O} \rangle$, the agent will start a main thread by invoking method *ExecuteModel* as specified in algorithm 1. First, it loads the serial number μ, the local goal g and local operators $\mathcal{O}$ from available local model (line 3). Lines 4-9 show a loop where the main thread continuously tries to achieve the local goal, unless there is a goal request sent by another agent being processed by a satisfier thread (line 5). This will allow the agent to achieve an intermediate (requested) goal first, and then continue to achieve its own local goal.

Algorithm 1 ExecuteModel

```
1: // main thread
2: global $stopped \leftarrow false$
3: $\mu, g, \mathcal{O} \leftarrow$ GetLocalModel()
4: while $stopped = false$ do
5:     if GetActiveSatisfierThread() $= \emptyset$ and
       AchieveLocalGoal($\mu, g, \mathcal{O}, 1$) = failure then
6:         $stopped \leftarrow true$
7:         return failure
8:     end if
9: end while
```

Algorithm 2 AchieveLocalGoal($\mu, goal, \mathcal{O}, \pi$)

```
1: $current \leftarrow$ GetLocalCurrentState()
2: $flaws \leftarrow$ ComputeFlaws($goal, current$)
3: if $flaws = \emptyset$ then return no-flaw end if
4: $operator \leftarrow$ SelectOperator($flaws, \mathcal{O}, \pi$)
5: if $operator = None$ then return failure end if
6: // at this step: $operator.priorityIndex >= \pi$
7: if $operator.selected = true$ then return ongoing end if
8: $operator.selected \leftarrow true$
9: $\pi' \leftarrow operator.priorityIndex + 1$
10: $pre_{local}, pre_{remote} \leftarrow$ SplitPreconditions($operator$)
11: repeat
12:     status = AchieveLocalGoal($\mu, pre_{local}, \mathcal{O}, \pi'$)
13: until $status =$ no-flaw or $status =$ failure
14: if $status =$ failure or
    AchieveRemoteGoal($\mu, pre_{remote}, \pi'$) = failure or
    Invoke($operator$) = failure then
15:     $operator.selected \leftarrow false$
16:     return failure
17: end if
18: $operator.selected \leftarrow false$
19: return flaw-repaired
```

Method *AchieveLocalGoal* in algorithm 2 is called by the main thread to achieve the local goal, or a satisfier thread to achieve a requested goal. This method is *reactive* because it directly selects and invokes an operator to repair any goal flaws. First, it compares the given goal with the current state to find goal flaws (lines 1-2). If such flaws exist, it searches all applicable operators from $\mathcal{O}$ with priority index higher or equals than a given value (π), and then selects an operator with the lowest priority index (line 4). This selection based on priority index aims to maintain the ordering constraints as specified in the workflow. If there is no applicable operator, then the goal cannot be achieved and status *failure* is returned (lines 5). If the selected operator is being executed by another

[3] The serial number is used by the agents to ensure that they only communicate with other agents who are running the same version of the model.

Algorithm 3 AchieveRemoteGoal($\mu, goals, \pi'$)

```
1: $goals' \leftarrow$ SplitGoalsByAgent($goals$)
2: for each $\langle agent, goal \rangle$ in $goals'$ do
3:     $response \leftarrow$ SendGoalToAgent($agent, \mu, goal, \pi'$)
4:     if $response \neq success$ then return failure end if
5: end for
6: return success
```

Algorithm 4 ReceiveGoalFromAgent($agent_i, \mu_i, goal_i, \pi_i$)

```
1: // satisfier thread
2: global stopped
3: $\mu, g, \mathcal{O} \leftarrow$ GetLocalModel()
4: if $\mu_i < \mu$ then
5:     SendResponseTo($agent_i$, denied)
6: else
7:     repeat
8:         $status \leftarrow$ AchieveLocalGoal($\mu, goal_i, \mathcal{O}, \pi_i$)
9:     until $status$ = no-flaw or $status$ = failure
10:    if $status$ = no-flaw then
11:        SendResponseTo($agent_i$, success)
12:    else if $status$ = failure then
13:        SendResponseTo($agent_i$, failure)
14:    end if
15: end if
```

thread, then it returns an *ongoing* status value (lines 7), so that the thread can wait until the execution has finished. Otherwise, the selected operator will be invoked after its local and remote precondition has been satisfied. A local precondition is achieved by *recursively* calling the same method where the operator's local precondition and priority index are passed as arguments (lines 11-13). A remote precondition is achieved by calling method *AchieveRemoteGoal* (line 14).

Method *AchieveRemoteGoal* in algorithm 3 separates the remote preconditions based on the target agents (line 1). It sends the new goal to each agent (line 3), and then wait for its response. A *success* response means that the goal has been achieved, other it cannot be achieved (lines 4). This response determines whether the selected operator can be invoked or not.

Any requested goal sent by another agent is received by a satisfier thread that directly invokes method *ReceiveGoalFromAgent* in algorithm 4. This method has three parameters i.e. a remote agent that sends the request ($agent_i$), remote agent's model serial number (μ_i), the requested goal ($goal_i$), and the minimum priority index of the next selected operator (π_i). In lines 4-5, it checks whether the remote agent uses an expired model. If it does then a *denied* message is sent to the requester agent. Otherwise, it repetitively calls method *AchieveLocalGoal* to achieve the requested goal. If the goal has been achieved (*no-flaw*) then it sends back status *success* (lines 10-11), or if there is a failure then it sends back status *failure* (lines 12-13).

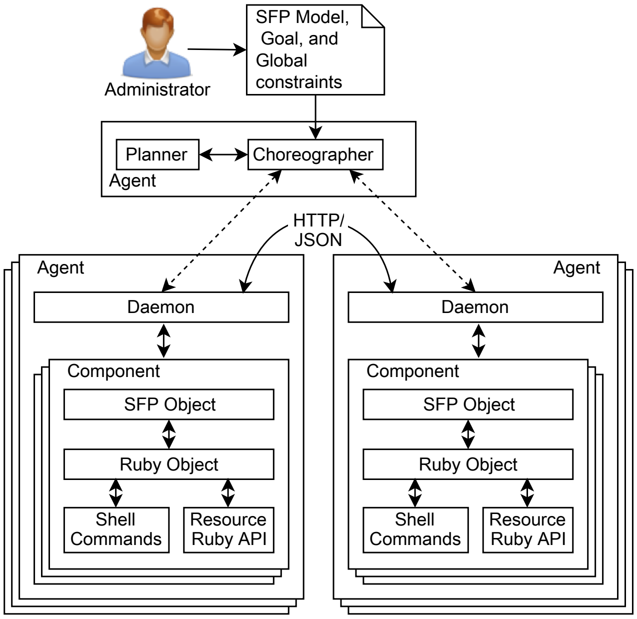

Fig. 3: The architecture of Nuri configuration tool.

VI. Implementation

We have integrated these choreography and reactive-regression algorithms into the Nuri configuration tool[4]. Nuri uses SFP as its configuration language. The system is implemented in Ruby except for the planner component, which is written in Ruby and C++. The Nuri architecture is shown in figure 3.

In Nuri, each node is managed by an agent which consists of a daemon and a set of components. The daemon is responsible for managing the local BSig model: accepting a new model from the choreographer, instantiating and constructing required components based on the model, and executing the model using the reactive regression algorithm. Each component is an instance of a Nuri module, and is responsible for managing a software package or a resource. A Nuri module is an abstract component which mainly consists of two files: 1) an SFP file that specifies a schema as an abstract description of the resource, and 2) a Ruby file that specifies a Ruby class as the implementation of the SFP schema. There is a clear separation between the declarative description in SFP and its Ruby implementation. The mapping from SFP to Ruby and vice versa is done by a daemon that uses the Ruby-SFP library. This clear separation allows us to have different implementations, for example one in Ruby and another in Java, communicating transparently using the same configuration language.

In the choreography stage, an administrator defines and submits an SFP model with the goal and global constraints of the managed system to a choreographer agent. Based on the SFP model, the choreographer aggregates the current state of the system by sending a request to each agent's daemon (dashed lines in the diagram). It then sends the current state

[4] Source code repository: https://github.com/herry13/nuri.

together with the goal and global constraints to the planner to automatically generate a workflow. Currently, we use an implementation of FastDownward [18] as the planner using FF and LM-Cut heuristics in searching. With this workflow, the choreographer constructs a system BSig model. Each local model of the system model is deployed to the target agent (dashed lines). The current implementation uses *push-based* architecture, but it is also possible to use *pull-based*.

Whenever an agent's daemon receives a local BSig model, it stops all threads and then restarts the execution using this new model. In execution, it calls the *getState* Ruby method of each component to get the current state of the resource. This state is translated into SFP to be compared with the local goal of the local BSig model to find any flaw. If any such flaw exists, the daemon will search for a local operator of the local model that can repair the flaw and satisfy the priority index constraint. If an operator is found and it requires some precondition provided by other agents, this daemon will send the goal request to other agent's daemon through HTTP/JSON protocol. Whenever all operator's preconditions have been satisfied, the daemon will invoke a Ruby method that implements the selected operator. Afterwards, the execution result is verified by the daemon by comparing post-invocation state with postcondition of the selected operator.

VII. Evaluation

We have used Nuri to deploy and reconfigure several example systems to a public cloud. One of these is a three-tier web application that consists of a VM with an Apache Load Balancer, a set of VMs where each has an Apache Web Server with Tiki Wiki Content Management, and a VM with a MySQL Database Server. A ganglia monitoring service is also installed and running on each VM.

In our first evaluation, we used Nuri to simultaneously deploy two instances of the system to two public cloud infrastructures i.e. HPCloud and Amazon Web Service (AWS), one as the main system and the other as the backup. We varied the number of VMs in the application layer to measure the effect of system's size on Nuri's performance. To manage the public clouds, we run two agents on our internal servers that act as cloud proxies for each public cloud. Each agent has a component with the capability to create and delete a VM on particular public cloud.

These systems are deployed from scratch, which means that there is no existing VM on any public cloud in the initial state. For the case where each system has 5 VMs in the application layer, the choreographer agent automatically constructed the model in about 11 minutes and 40 seconds (dominated by the planner). This model consists of 16 local models: 2 models for the cloud proxy agents, 7 models for the agents of the main system, and 7 models for the agents of the backup system. The first 2 models were sent to each cloud proxy agent. While the 7 models of the main system was sent to HPCloud proxy agent, and the other 7 models of the backup system was sent to AWS proxy agent. These 14 local models were kept temporarily until the VMs have been created and the agent's daemon has run on the VM. Whenever the agent daemon of a VM is available, the appropriate local model is sent by the cloud proxy agent to this daemon. Afterwards, the daemon executes the model to achieve its local goal. We checked the state of the system periodically after the model had been deployed by requesting the current state of all agents. The outputs showed that the number of goal flaws decreases as time passes. In about 24 minutes and 12 seconds after we submitted the SFP model and goal and global constraints, the output showed that the system had reached the goal state, i.e. there are two systems available on the HPCloud and on AWS. We also checked the execution logs of each agent and compared the invocation timestamp of all operators. The comparison results showed that the global constraints were not violated during execution.

Figure 4 illustrates the comparison of deployment times of the above system using Nuri and a centralised execution framework in [1] that uses partial-order workflow execution engine. It shows that there is no significant difference between these two frameworks. This is due to very low network latency time ($< \sim 0.15$s) between the central controller with HPCloud and AWS. We believe that the result will be different if the network latency time is higher. However, the centralised framework requires a controller which may be on our private infrastructure, or on any public cloud. Obviously, if there is a network outage on our infrastructure, or on the public cloud that hosts the controller, then the whole execution will be stopped. But in Nuri, the execution of the system on a healthy public cloud will continue even if there is a problem on the controller's infrastructure. The grey bars show that there is no significant difference between the planning time for the centralised framework, and the choreographing time in Nuri. Since our choreographing process consists of planning and translation steps, it shows that the translation requires insignificant time.

Finding an optimal plan is a PSPACE-complete problem [19]. In practice, the performance of the FastDownward planner varies according to the number of modules and the dependencies of their procedures, as well as the complexity of goal/global constraints formulae. The planner requires a grounded representation of all procedures, where on most cases the grounding process time is larger than the searching time. As part of future work we will develop a more efficient grounding technique.

In a second evaluation experiment, we tested the self-healing capability of Nuri on previously deployed systems. We manually stopped or uninstalled some services randomly and checked the state of the system several minutes later. Since the agent of each VM continuously executed the reactive regression algorithm, it detected such errors as goal flaws, selected and invoked some operators to fix them. In another, we manually deleted some random VMs on the public cloud. In this case, since the cloud proxy was continuously executing its local model, it could detect that those VMs did not exist on the public clouds. Based on the model, it then automatically created some VMs to replace the deleted ones. After installing and starting a Nuri daemon on each new VM, the cloud proxy

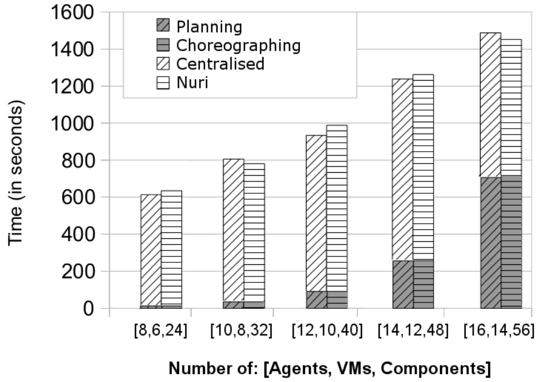

Fig. 4: The deployment times using centralised framework and Nuri.

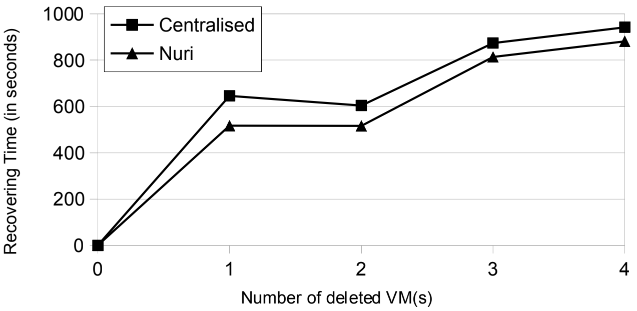

Fig. 5: The recovering times using centralised framework and Nuri.

agent sent the appropriate local model to be executed on the new VM. After several minutes, all deleted VMs had been replaced with the new ones and the system was again stable. These results show that Nuri can fix a drift from the desired state by using the existing BSig model, distributively, and without any re-choreographing process. Figure 5 shows the comparison of recovering times using centralised framework and Nuri. It shows that the centralised framework requires some extra time for re-planning to generate a workflow that fixes the drift, which is not required by Nuri.

VIII. Conclusions and Future Work

This paper has described a technique to compile a workflow to a Behavioural Signature model. The evaluations show that the execution of this model using a reactive regression algorithm enables autonomous distributed execution, while preserving the global constraints. This eliminates a single point of failure and hence increases the system's resilience. The evaluations also show that the reactive regression algorithm is able to synchronise the agents and maintain the ordering constraints without the need for re-planning to fix some drifts from the desired state.

In the current implementation, the generated model is capable of autonomic recovery from an initial state (or some intermediate state) of the generated plan. Future work will attempt to generalise this by merging multiple workflows to support autonomic recovery from multiple initial states.

Acknowledgment

The authors would like to thank to Lawrence Wilcock and Eric Delliot from HP Labs Bristol for their valuable feedback and use cases. We also thank to Andrew Farrell for valuable discussions on Behavioural Signature model. This research is fully supported by a grant from HP Labs Innovation Research Program Award.

References

[1] H. Herry and P. Anderson, "Planning with global constraints for computing infrastructure reconfiguration," in *AAAI-12 Workshop on Problem Solving using Classical Planners (CP4PS'12)*. AAAI Press, 2012.

[2] P. Goldsack, J. Guijarro, S. Loughran, A. Coles, A. Farrell, A. Lain, P. Murray, and P. Toft, "The smartfrog configuration management framework," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 1, pp. 16–25, 2009.

[3] A. Farrell, S. Prakash, and J. Rolia, "Behavioral Signatures for Business Service Management in the Cloud," HP Labs, Tech. Rep., 2010.

[4] Puppet Labs, "Puppet," 2013. [Online]. Available: http://www.puppetlabs.com/puppet

[5] Opscode Inc., "Chef," 2013. [Online]. Available: http://www.opscode.com/chef

[6] P. Anderson and A. Scobie, "LCFG: The next generation," in *UKUUG Winter Conference*, 2002.

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

[8] H. Herry, P. Anderson, and G. Wickler, "Automated planning for configuration changes," in *Proceedings of the 25th Large Installation System Administration Conference (LISA '11)*. Usenix Association, 2011.

[9] IBM Corp., "Integrated Service Management software, IBM Tivoli," 2013. [Online]. Available: http://www.ibm.com/software/tivoli

[10] Microsoft Corp., "Microsoft System Center," 2013. [Online]. Available: http://www.microsoft.com/en-us/server-cloud/system-center

[11] Canonical, "Juju," 2013. [Online]. Available: {http://juju.ubuntu.com}

[12] "RunDeck," 2013. [Online]. Available: http://rundeck.org

[13] A. Barker, C. D. Walton, and D. Robertson, "Choreographing web services," *IEEE Transactions on Services Computing*, vol. 2, no. 2, pp. 152–166, 2009.

[14] P. Besana, V. Patkar, A. Barker, D. Robertson, and D. Glasspool, "Sharing choreographies in openknowledge: A novel approach to interoperability," *Journal of Software*, vol. 4, no. 8, pp. 833–842, 2009.

[15] P. Anderson, S. Bijani, and A. Vichos, "Multi-agent negotiation of virtual machine migration using the lightweight coordination calculus," in *Proceedings of the 6th International KES Conference on Agents and Multi-agent Systems – Technologies and Applications*, 2012.

[16] R. Micalizio, "A distributed control loop for autonomous recovery in a multi-agent plan," in *Proceedings of the 21st international jont conference on Artifical intelligence*. Morgan Kaufmann Publishers Inc., 2009, pp. 1760–1765.

[17] J. A. Shah, P. R. Conrad, and B. C. Williams, "Fast distributed multi-agent plan execution with dynamic task assignment and scheduling," in *Proc. of ICAPS*, vol. 9, 2009, pp. 289–296.

[18] M. Helmert, "The fast downward planning system," *Journal of Artificial Intelligence Research*, vol. 26, no. 1, pp. 191–246, 2006.

[19] T. Bylander, "The computational complexity of propositional strips planning," *Artificial Intelligence*, vol. 69, no. 1, pp. 165–204, 1994.