Choreographing Configuration Changes

Herry Herry*, Paul Anderson†, and Michael Rovatsos‡
School of Informatics, University of 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. However, these workflows are *orchestrated* by a central controller which creates a potential bottleneck, and may also be susceptible to communication failures, which are particularly likely since reconfiguration often occurs as an autonomic response to system failures.

Fully distributed planning, on the other hand, is not a good solution to this problem – avoiding deadlock/livelock may require agents to have considerable global knowledge, and achieving this is likely to result in even more costly inter-agent communication. Predicting the behaviour of such systems is also more difficult and hence a fully decentralised design is likely to be 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 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. Our evaluation results show that the Nuri agents can achieve the goal state without any central control, while maintaining pre-specified global constraints throughout the changes.

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 [2] and Chef [3] are now ubiquitous. Many of these tools 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 [4]. An alternative approach is to specify the workflow manually [5], [6]. 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.

Most practical configuration tools are also 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. There has been some previous work on distributed workflow execution using multi-agent systems (e.g. [7], [8], [9], [10]) and the *Behavioural Signatures* model proposed for SmartFrog [11] is particularly relevant. However, in all these cases, the dependencies must be computed manually which is error-prone and time-consuming. The resulting models must also be validated for deadlock and livelock conditions before they can be deployed.

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. The following global constraints must be satisfied in deployment:

- 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.

To model the system, we use an object-oriented configuration language called SFP which is introduced in [1]. For our example, we define five schemata as follows (For brevity, we omit the some procedures.):

```
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 none of the software components are (yet) installed.

To bring the system to the desired state and preserve the global constraints, 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 [12]. We then use an off-the-shelf planner to generate the workflow.

A configuration task consists of an initial state as represented by the above model. 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
}
```

From this, the planner can generate a workflow which is used to automatically construct and deploy a set of distributed components. These components implement a BSig model capable of choreographing the changes autonomously. The details of this translation process are described in section IV.

In using this model, each agent applies an execution algorithm called *cooperative 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.

In executing the model, each agent is communicating purely in a *peer-to-peer* fashion with other agents. Communication is initiated when an agent needs to invoke an operator which has preconditions that 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 is a workflow generated by the planner, that should be executed by all agents during configuration changes (without any single point of control). 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*.

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.

Definition 2: $v_i = d_i$ is a local of agent α iff $v_i = d_i \in$ *postcondition*(o_i) and $o_i \in$ *operators*(α).

A local operator determines *what* local changes can be made *when*.

Definition 3: 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, each 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.

Local operators of each agent can be obtained from the workflow by considering the namespace associated with every procedure. Preconditions and postconditions of each operator can be obtained from: the grounded procedure specification and the procedure orderings of the workflow. In order to maintain the global constraint during execution, we inject postconditions of precedence operators into a particular operator's preconditions. This ensures that the model is free of global conflicts, even though its execution is distributed.

During execution, instead of selecting an arbitrary operator, the agent should select one of the operators that has the lowest priority index, which is calculated using the following equation:

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

where $Succ_j$ is the set of successor operators of o_j.

Priority index values reflect the ordering constraints between operators as specified by the workflow. Our execution algorithm will use these values to ensure that there is no deadlock or livelock situation during execution.

Local and system BSig model are defined as follows:

Definition 4: 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.

Definition 5: A Behavioural Signature model of a system is a tuple $\mathcal{M} = \langle A, M \rangle$, where A is a set of agents,

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
```

$M = \bigcup_{\forall ag \in A} \{m_{ag}\}$, and $\forall m_i, m_j \in M \,.\, \mu_i = \mu_j$.

V. Executing the Model

To execute the model, we use a novel algorithm called *cooperative reactive regression*. It is *cooperative* since it prioritizes goals requested by other components over the local goal. It is *reactive* since it continuously tries to find and repair 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. This execution algorithm is shown in algorithms 1, 2, 3, and 4.

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.

VI. Implementation

We have integrated these choreography and cooperative reactive-regression algorithms into the Nuri configuration tool[1]. Currently, we use an implementation of FastDownward [13] as the planner. The choreographer constructs a system BSig model based on the workflow generated by this planner. Each local model is deployed to the target agent using *push-based* mechanism.

Each managed 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. Each component is an instance of a Nuri module, and is responsible for managing a software package or a resource.

[1]See http://edin.ac/18TxUO6 for a diagram of the Nuri architecture (Nuri source code is available at https://github.com/herry13/nuri)

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
```

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
```

The module includes an SFP file specifying the schema, and implementation code (in Ruby).

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

In our first evaluation, we used Nuri to simultaneously deploy two instances of the 3-tier web application system to two public cloud infrastructures i.e. HPCloud and Amazon Web Service (AWS). We varied the number of VMs in the application layer to measure the effect of system's size on Nuri's performance. These systems are deployed from scratch, which means that there is no existing VM on any public cloud

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
```

in the initial state.

Figure 1 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. This is due to very low network latency time ($< \sim$0.15s) between the central controller with HPCloud and AWS. We believe that the result will be different if the network latency time is higher. However, if there is a network outage on our infrastructure, or on the public cloud that hosts the central 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 a solution plan is a PSPACE-complete problem in general case [14]. In practice, the performance of the planner varies according to the number of modules and the dependencies of their procedures, as well as the complexity of the goal/global constraints formulae.

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. In another, we manually deleted some random VMs on the public cloud. For these cases, since the agent of each VM continuously executed the cooperative reactive regression algorithm, it detected such errors as goal flaws, selected and invoked some operators to fix them. This shows that any drift from the desired state could be fixed distributively without any re-choreographing. As shown in figure 2, this is reflected in the faster recovery times compared to a central re-planning solution.

VIII. Conclusions and Future Work

This paper has described a technique to compile a workflow into a Behavioural Signature model for autonomous execution. The evaluations show that the execution of this model

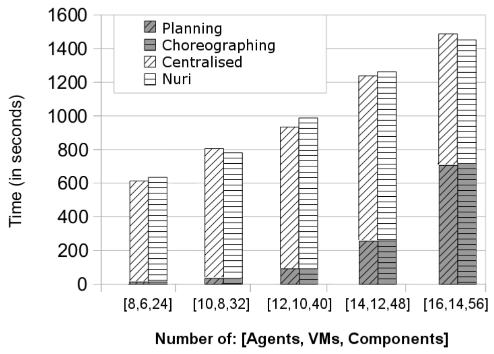

Fig. 1: The deployment times using centralised and Nuri.

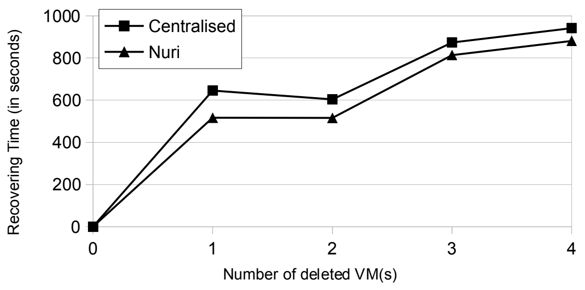

Fig. 2: The recovering times using centralised and Nuri.

using a cooperative reactive regression algorithm: 1) enables autonomous distributed execution and preserves the global constraints, and 2) synchronises the agents and maintains ordering constraints without the need for re-planning to fix any drift from the desired state. This eliminates any single point of failure and hence increases the system's resilience.

Acknowledgment

The authors would like to thank to Lawrence Wilcock and Eric Delliot from HP Labs Bristol, Andrew Farrell, and anonymous reviewers. 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] Puppet Labs, "Puppet," 2013. [Online]. Available: http://www.puppetlabs.com/puppet
[3] Opscode Inc., "Chef," 2013. [Online]. Available: http://www.opscode.com/chef
[4] 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.
[5] IBM Corp., "Integrated Service Management software, IBM Tivoli," 2013. [Online]. Available: http://www.ibm.com/software/tivoli
[6] Microsoft Corp., "Microsoft System Center," 2013. [Online]. Available: http://www.microsoft.com/en-us/server-cloud/system-center
[7] A. Barker, C. D. Walton, and D. Robertson, "Choreographing web services," *IEEE Transactions on Services Computing*, vol. 2, no. 2, pp. 152–166, 2009.
[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] M. Helmert, “Concise finite-domain representations for PDDL planning tasks,” *Artificial Intelligence*, vol. 173, no. 5-6, pp. 503–535, 2009.

[13] ——, “The fast downward planning system,” *Journal of Artificial Intelligence Research*, vol. 26, no. 1, pp. 191–246, 2006.

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