QoS and Energy Management Coordination using Discrete Controller Synthesis

Noël De Palma
Grenoble University, France
noel.de_palma@inrialpes.fr

Gwenaël Delaval
Grenoble University, France
gwenael.delaval@inria.fr

Eric Rutten
INRIA, France
eric.rutten@inria.fr

Abstract

Green computing is nowadays a major challenge for most IT organizations. Administrators have to manage the trade-off between system performances and energy saving goals. Autonomic computing is a promising approach to control the QoS and the energy consumed by a system. This paper precisely investigates the use of synchronous programming and discrete controller synthesis to automate the generation of a controller that enforces the required coordination between QoS and energy managers. We illustrate our approach by describing the coordination between a simple admission controller and an energy controller.

Categories and Subject Descriptors I.2.2 [*Automatic Programming*]: Program synthesis; D.2.9 [*Software Engineering*]: Management; I.2.2 [*Distributed Systems*]

Keywords Green Computing, Autonomic Computing, Synchronous Programming, Discrete Controller Synthesis

1. Introduction

Green computing is nowadays a major challenge for most IT organizations that involve medium and large scale distributed infrastructures like Grids, Clouds and Clusters. Distributed systems and Internet services usually require a variety of software systems that are organized in complex architectures based on replication and multi-tier organization. The administration of such systems must take into account the impact of green computing on the traditional distributed system issues like dependability or scalabilitity. In particular, administrators have to manage the trade-off between system performances and energy saving goals. Autonomic computing [8], i.e. self-management in the face of evolving load conditions or failures is a promising approach that aims at providing a high-level support for self-* aspects. To control the QoS of a system, autonomic managers can act for instance on :

- The work submitted to the system (e.g admission control [13])
- The resource used by the system (e.g static or dynamic resource provisioning [3])
- The work achieved by the system (e.g service degradation [11])

GCM'10, 29-NOV-2010, Bangalore, India.

On the other side, autonomic managers can control the energy consumed by a system by using these following actuators :

- The frequency or voltage of server CPUs (e.g dynamic voltage scaling [2])
- The state of hardware devices (e.g ACPI state management [7])
- The server consolidation (e.g virtualisation [9])

Control techniques have been used to ease the design of autonomic computing systems [1][9]. However few of these works have explored the coordination between QoS and energy managers. This paper precisely investigates this problem by using discrete controller synthesis to automate the generation of a controller that enforces the required coordination between QoS and energy managers.

Discrete controller synthesis (DCS) [12] allows to design programs in a mixed imperative/declarative way. From a program with some freedom degrees left by the programmer (e.g., free controllable variables), and a temporal property to enforce which is not a priori verified by the initial program, DCS tools compute off-line automatically a *controller* which will constrain the program (by e.g., giving values to controllable variables) such that, whatever the values of inputs from the environment, the *controlled program* verifies the temporal property.

The advantages are (i) to generate automatically the controller required to enable the cooperation of multiple Autonomic Managers from high-level policy, (ii) to ease the evolution of the coordination strategy and (iii) the generated controller is correct by construction.

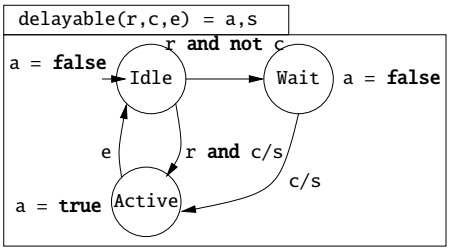
We illustrate our approach by describing the coordination between a simple admission controller and an energy controller. This paper is organized as follow : section 2 is about synchronous programming and discrete controller synthesis, section 3 gives an example of an admission and energy controller and their possible coordination, section 4 describes the design of the controllers using discrete controller synthesis and section 6 concludes this work and gives an insight of our future works.

2. Synchronous Programming and Discrete Controller Synthesis

For our contribution, we use the DCS tool SIGALI [10] and the language BZR [6]. This language allows to describe reactive systems by means of generalized Moore machines, i.e., mixed synchronous dataflow equations and automata [5], with parallel and hierarchical composition. The basic behavior is that at each reaction step, values in the input flows are used in order to compute the values in the output flows for that step. Inside the nodes, this is expressed as a set of declarations, which takes the form of equations defining, for each output and local, the values that the flow takes, in terms of

an expression on other flows, possibly using local flows and values computed in preceding steps (also known as state values).

Figure 1 shows a small program in this language. It features a task, which can either be idle or active. When it is idle, i.e., in the initial Idle state, then the occurrence of the input `r` *requests* the launch of the task. Another input `c` (which will be controlled further by the synthesized controller) can either allow the activation, or temporary block the request and make the automaton go to a waiting state. When active, the task can be ended with the input `e`. This `delayable` node has two outputs, `a` featuring the instantaneous activity of the task, and `s` being emitted on the instant when it becomes active.




```
node delayable(r,c,e:bool) returns (a,s:bool)
  let
    automaton
      state Idle
        do a = false ; s = r and c
        until r and c then Active
            | r and not c then Wait
      state Wait
        do a = false ; s = c
        until c then Active
      state Active
        do a = true ; s=false
        until e then Idle
    end
   tel
```

Figure 1. Delayable task in graphical and textual syntax.

The main feature of this language is that its compilation involves *discrete controller synthesis* (DCS). DCS allows to compute automatically a controller, i.e., a function which will act on the initial program so as to enforce a given temporal property. Concretely, the BZR language allows the declaration of *controllable variables*, the value of which being not defined by the programmer. These free variables can be used in the program so as to let some choices undecided (e.g., choice between several transitions). These variables are then defined, in the final executable program, by the controller computed by DCS.

The Figure 2 shows an example of use of these controllable variables. This example consists in two instances of the `delayable` node, as defined in Figure 1. These instances run in parallel, defined by synchronous composition: one global step corresponds to one local step for every equation, i.e., here, for every instance of the automaton in the `delayable` node. Then, the `twotasks` node so defined is given a *contract* composed of two parts: the **`with`** part allowing the declaration of controllable variables (c_1 and c_2), and the **`enforce`** part allowing the programmer to assert the property to be enforced by DCS, using the controllable variables. Here, we want to ensure that the two tasks running in parallel won't be both active at the same time. Thus, c_1 and c_2 will be used by the computed controller to block some requests, leading automata of tasks to the wating state whenever the other task is active.

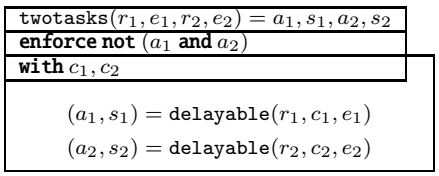


Figure 2. Mutual exclusion enforced by DCS in BZR.

3. QoS and Energy management

To illustrate our work, we choose a standard pattern for scalability and availability of Internet server. Modern Internet Application Servers are composed of different tiers such as web, application and database servers. Upon HTTP client requests, either the requests target a static web document, in this case the web server directly process the requests; or it refers to a dynamically generated document, in that case the web server forwards the requests to the application server, and a fraction of those also require processing by database server. In this pattern, a given server S is statically replicated at deployment time and a load balancer distributes incoming requests among the replicated servers. The distribution algorithm is usually Round-robin (equally distributing the load between the servers).

Adapting dynamically the number of requests accepted in the system is a means to maintain a given QoS and to avoid server trashing despite varying load. An example of a simple admission controller is described section 3.1.

On the other hand, adapting dynamically the number of replicas is a means to dynamically allocate or free machines, i.e. to dynamically turn cluster nodes on - to be able to efficiently handle the load imposed on the system - and off - to save power under lighter load. An example of such an energy controller is described section 3.2.

3.1 Admission control

For the purpose of this paper we use a load balancing scheme in a cluster of Web servers that includes an admission control which accepts/rejects new client requests to maintain high system throughput (new requests will be rejected if the load is too high). The admission control algorithm is based on the CPU utilization metric, that is retrieved from the Web servers at fixed intervals. The admission controller computes a moving average of the collected data in order to remove artifacts characterizing the CPU consumption. It finally computes an average CPU load across all nodes, so as to observe a general load indication of the whole replicated server (CPU_AVG). The acceptance rate of client requests is based on the average CPU load and on a cost associated to each request types. For simplicity, we do not take into account session affinity (requests belonging to existing session should be immediately forwarded to the right web server). Among the new accepted requests, the load balancer computes the fraction of requests to be assigned to each server according to a simple round-robin policy. The important parameters here are the maximal CPU usage above which the admission controller must reject a request (called MaxCPU_AC) and the cost estimation for each type of requests.

3.2 Energy control

In this section, we describe an energy controller for replicated cluster-based systems. Energy optimization aims at dynamically adapting the degree of replication according to the load the system receives. This adaptation is used to dynamically turn cluster nodes on and off to save power under lighter load.

The controller is mainly based on performance thresholds that describe an optimal performance region where the system must be

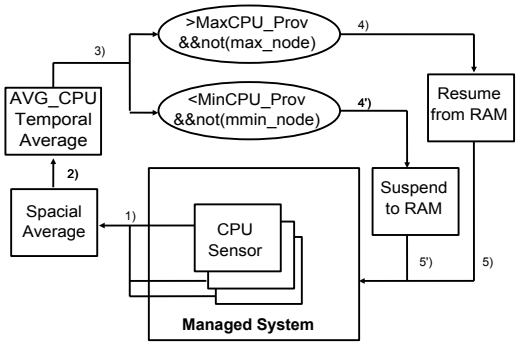


Figure 3. Optimization loop

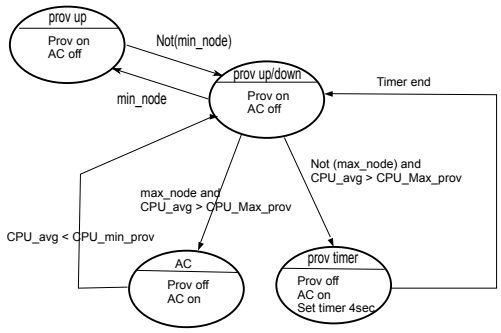


Figure 4. Super Controller

depending on the workload. The expected benefits are (i) improving energy consumption; and (ii) preserving user-perceived performance in the face of wide variations of the load. The sensors used are the same than the admission controller sensors. The actuators are used to increase/decrease the number of server replicas and to update the loadbalancer to take into account the new replicated server configuration.

The autonomic manager makes use of an analysis/decision part based on a maximal and a minimal CPU threshold (MaxCPU_Prov and MinCPU_Prov) that regulates node allocation. It is also constrained by a minimal and a maximal node threshold (MinNodes and MaxNodes) that represents the maximal (resp. minimal) number of node that can be allocated to the clustered server. The behavior of this autonomic manager is shown in Figure 3. When it receives notifications from sensors, if a reconfiguration is required, it modifies the number of allocated resources (i.e the nodes) by contacting a node allocator to allocate a new node (if $\mathrm{AVG\_CPU} > \mathrm{MaxCPU\_Prov}$) or to de-allocate it (if $\mathrm{AVG\_CPU} < \mathrm{MinCPU\_Prov}$). Turning machines off and (especially) on is quite costly. We measured that such an operation costs about 45 seconds in the average. Instead, we rely on suspension to RAM, which allows to suspend and resume the activity of a machine at a low cost (about 4 seconds in the average for resuming a machine) while saving as much energy as if it was turned off. Suspend-to-RAM stores information on system configuration, open applications, and active files in main memory (RAM), while most of the system's other components are turned off. When a machine is suspended, only the RAM and the network device are power on.

3.3 Controller coordination

In this section, we analyze the control of a system composed of both the admission controller and the energy controller described above. The policy we described here aims to ensure a given QoS by consolidating current workload onto the minimum number of machines sufficient to serve it while minimizing the number of rejected requests.

It is important to understand that both controllers are developed independently. Without coordination, the admission controller may prevent the energy controller to add a new node in the system and may result in a higher number of rejected requests. Remember the parameters that regulate the behavior of each controller are MaxCpu_AC for the admission control, MaxCPU_Prov, MinCPU_Prov and Min/Max_Node for the energy controller.

Whatever the settings are ($\mathrm{MaxCpu\_AC} \leq \mathrm{MaxCPU\_Prov}$ or $\mathrm{MaxCpu\_AC} > \mathrm{MaxCPU\_Prov}$) there is always a chance that the admission controller prevents the energy manager to add more nodes in the system. For instance, imagine a setting with one node allocated to the clustered server and where MaxCpu_AC = 80%, MaxCpu_Prov = 90%. If the AVG_CPU = 75% at the time of sampling and if we have 3 incoming requests with 5% cost each. The energy controller will not add a new node, the QoS will be guaranteed but at the cost of 2 request rejections.

To implement the policy described before, we need the controllers to provide a on/off switch to activate/deactivate their operations. These switches will be used by a super controller to implement our policy. The state followed by the super controller are depicted Figure4 :

- Prov up/down : the energy controller can freely scale up or down the clustered server to minimize the energy consumed while maintaining the QoS.
- Prov up : the clustered server is in its minimal configuration, the energy controller can only resume a server from RAM.
- AC : the clustered server has reached its maximal allowed configuration. The energy controller is disabled whereas the admission controller is on (to maintain the QoS until necessary).
- Prov timer : The energy controller has requested a new server to be resumed from RAM. The admission controller is on during the required time to enable the new server (4 sec in our setting). This prevents the possible trashing of the clustered server during the delay required to insert the new server in the cluster.

As we will see in the next section, this super controller will be automatically generated using discrete controller synthesis.

4. Synchronous controller design

We can first notice that, if we program directly the automaton showed in Figure 4, this can lead, at first to some unefficiencies or error while identifying when the different controllers has to be activated or de-activated, and then, to a design which can be tricky to make evolve. This problem appears in this automaton by the several "AC on/off" placed on several identified states, jeopardizing the modularity of the design.

We want to show how the use of DCS can help to design controllers first independently, and only then coordinating them by adding policies to be enforced.

We first design the controller for the provisioning policy. Figure 5 gives the automaton of this designed. The outputs of this automaton are four Booleans, `add_machine` and `remove_machine` being signals allowing the controller to request the resuming or suspension of a machine, `act_prov_up` being true whenever the provisioning "up" policy is active (i.e., it watches upon the average

CPU and can request for a new machine), and act_prov_down being true whenever the "down" policy is active.

The initial state is UpDown, where both "up" and "down" policies are active. When the CPU average reach a maximum level, the controller requests a new machine, and goes to the Adding state, where it awaits for the new machine to be actually available (notified by the timer_end input). In this Adding state, machines can neither be added nor removed, so both policies are idled. When timer_end occurs, the controller can either go back to UpDown, or if there is no more machines able to be resumed, go to the Down state where the "up" policy is idled. This Down state is left once one machine is suspended. The Up state is used when no machine can be removed (typically, there is only one active left): the "down" policy is then idled. We can notice here that this automaton does not comprise any controllability (for DCS).

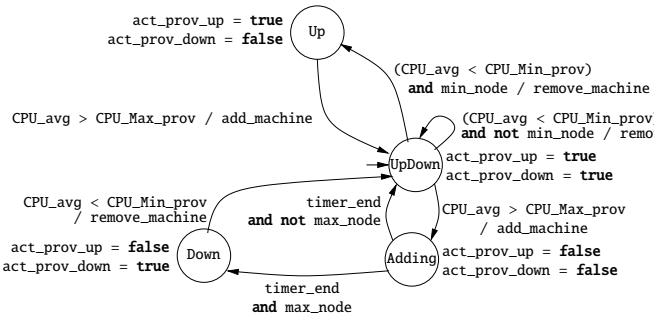


Figure 5. Automaton for the provisioning policy.

Then, we can design independently the automaton for the admission controller. We will state that this controller can be activated or idled at any time. Figure 6 gives this automaton. The input c activates or de-activates the controller, and the output act_ac is true whenever it is active.

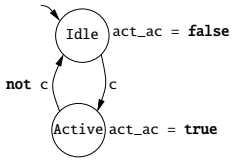


Figure 6. Automaton for the admission controller.

Typically, c is meant to be declared as a controllable variable: the choice of activation of the admission controller is left to the synthesized controller, whose aim is to enforce the main coordination policy. The main coordination program is shown in Figure 7. The two automata presented before are composed in parallel, and a contract is added to enforce the coordination policy. Here, we want that (i) the admission controller will never be active when the "up" provisioning is active, and (ii) if the "up" provisioning is idle, then the admission controller must be active. The first part of our coordination policy is stated by "**not** act_ac **and** act_prov_up", and the second part by "act_ac **and not** act_prov_up". c is declared as a controllable variable.

5. Controller Simulation

The designed and synthesized controller can then be simulated, before its system integration. Figure 8 shows a simulation example.

This figure shows a scenario illustrating our controller in action. The Boolean input provisioning_up represents the condition CPU_avg > CPU_Max_prov and provisioning_down the condition CPU_avg < CPU_Min_prov. At the beginning, the policies "provisioning up" and "provisioning down" are active (outputs act_prov_up and act_prov_down). At step 4, the input provisioning_up becomes true, triggering the addition of a machine. Until the step 7, where the input trigger_end reports that the new machine is available, the controller deactivates the provisioning policies, and activates the controller admission.

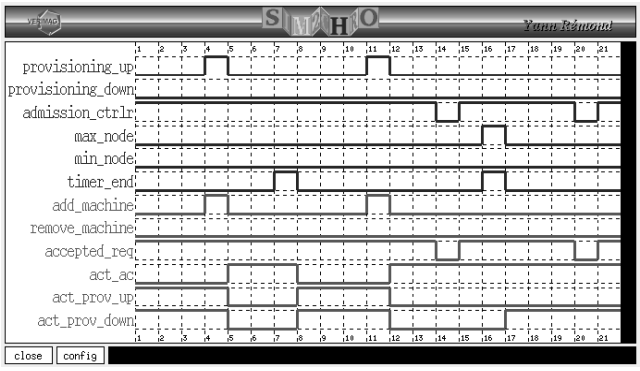


Figure 8. Controller simulation

At step 11, once again, provisioning_up triggers the addition of a new machine. This time, both timer_end and max_node are emitted, reporting that no new machine can be added. Thus, the controller activates only the "provisioning down" policy, and the admission controller remains activated.

6. Conclusion

Green computing impacts the traditional distributed system issues like dependability or scalabilitity. Managing the trade-off between system performances and energy saving goals is a complex issue. Autonomic computing is a promising approach to control this trade-off. A major challenge is to allow the coexistence between QoS and energy Managers in the same system in a consistent, efficient and flexible way.

Our approach is based on synchronous programming and Discrete Controller Synthesis. The advantages are (i) to generate automatically the controller required to enable the cooperation of multiple Autonomic Managers from high-level policy, (ii) to ease the evolution of the coordination strategy and (iii) the generate controller is correct by construction.

As a future work, we plan to integrate (i) a new consolidation manager based on virtual machines such as [9] and another QoS manager based on service degradation such as [11].

References

[1] Tarek F. Abdelzaher,John A. Stankovic, Chenyang Lu, Ronghua Zhang and Ying Lu. *Feedback Performance Control in Software Services.* IEEE Control Systems Magazine, 23, 2003.

[2] Ricardo Bianchini and Ram Rajamony. *Power and Energy Management for Server Systems.* Computer Journal, 37, 68-74, 2004.

[3] Sara Bouchenak, Noel De Palma, Daniel Hagimont, Christophe Taton. *Autonomic Management of Clustered Applications.* In proc. of CLUSTER, 2006

[4] Xiangping Chen, Huamin Chen, and Prasant Mohapatra. *Aces: An efficient admission control scheme for qos-aware web servers.* Computer Communications, 26:1581–1593,2003.

[5] J.-L. Colaço, B. Pagano, and M. Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. In *Embedded Software (EMSOFT)*, Jersey city, New Jersey, USA, September 2005.

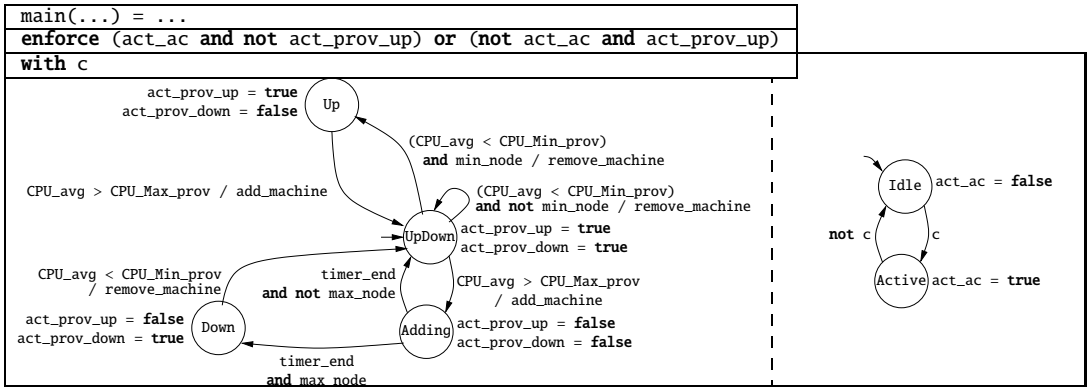


Figure 7. BZR program for coordination policy between provisioning and admission controller.

[6] G. Delaval, H. Marchand, and E. Rutten. Contracts for modular discrete controller synthesis. In *Languages, Compilers and Tools for Embedded Systems, LCTES, Stockholm, Sweden, April*, 2010.

[7] Hewlett-Packard, Intel, Microsoft, Phoenix, and Toshiba. *Advanced Configuration and Power Interface* http://www.acpi.info/

[8] Kephart J. *An architectural blueprint for autonomic computing.* IBM White Paper, 2003.

[9] Dara Kusic, Jeffrey O. Kephart, James E. Hanson2 Contact Information, Nagarajan Kandasamy and Guofei Jiang. *Power and performance management of virtualized computing environments via lookahead control.* Cluster Computing, 12(1), 2008.

[10] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic. Synthesis of discrete-event controllers based on the signal environment. *j. Discrete Event Dynamic System*, 10(4), October 2000.

[11] Jeremy Philippe, Noel De Palma, Fabienne Boyer, Olivier Gruber. *Self-adapting Service Level in Java Enterprise Edition.* In proc. of Middleware, 143-162, 2009:

[12] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25(1), 1987.

[13] Bhuvan Urgaonkar and Prashant Shenoy. *Cataclysm: Scalable overload policing for internet applications.* Journal of Network and Computer Applications, 31:891–920, 2008.