Discrete control for the coordination of administration loops * (extended abstract)

Soguy Mak-Karé Gueye
LIG / UJF
Grenoble, France
soguy-mak-kare.gueye@inria.fr

Noël De Palma
LIG / UJF
Grenoble, France
noel.de_palma@inria.fr

Eric Rutten
LIG / INRIA
Grenoble, France
eric.rutten@inria.fr

ABSTRACT

The increasing complexity of computer systems has led to the automation of administration functions, in the form of autonomic managers. One important aspect requiring such management is the issue of energy consumption of computing systems, in the perspective of green computing. As these managers address each a specific aspect, there is a need for using several managers to cover all the domains of administration. However, coordinating them is necessary for proper and effective global administration. Such coordination is a problem of synchronization and logical control of administration operations that can be applied by autonomous managers on the managed system at a given time in response to events observed on the state of this system. We therefore propose to investigate the use of reactive models with events and states, and discrete control techniques to solve this problem. In this paper, we illustrate this approach by integrating a controller obtained by synchronous programming, based on Discrete Controller Synthesis, in an autonomic system administration infrastructure. The role of this controller is to orchestrate the execution of reconfiguration operations of all administration policies to satisfy properties of logical consistency. We apply this approach to coordinate three managers : two energy-aware ones, which control server provisioning and processor frequency, and a repair manager.

Categories and Subject Descriptors

K.6 [**Management of Computing and Information Systems**]: System Management; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Computer-aided software engineering, State diagrams*; D.2.11 [**Software Architectures Domain-specific architectures**]:

General Terms

Languages, Performance, Verification

Keywords

Autonomic managers, administration loops, coordination, reconfiguration, components, model-based approach, reactive programming, discrete controller synthesis

*This research is supported by ANR INFRA (ANR-11-INFR 012 11) under a grant for the project ctrl-Green

1. COORDINATING LOOPS

1.1 Administration loops

One important challenge in autonomic computing is to get multiple autonomic managers to work together coherently with respect of their administration objectives. They are built to address different administration concerns. They react to events that possibly lead the system to a state in which their objectives are not met. Their reaction consists in applying reconfiguration operations to restore the system to a state satisfying their objectives. They target different objectives which can lead to conflicting decisions during their coexistence that possibly results to system inconsistency. It is necessary to coordinate their execution. Coordinating autonomic managers execution consists in controlling their behaviours in order to let them perform administration operations only if it is appropriate.

1.2 Our approach

We address the design of a coordination controller for the coexistence of multiple administration loops within a computing system using discrete control techniques. Our work proposes a solution for implementing a coordination infrastructure for autonomic managers, seen themselves as manageable elements.

Most of the proposed solutions for coordinating managers are based on hierarchical and hand-made manager structures which are in charge of synchronizing managers operations. Indeed, coordination is a problem of synchronization and logical control of administration operations that can be applied by autonomic managers on the managed system. Our approach is based on models of reactive systems, which are automata-based, and on *Discrete Controller Synthesis* (DCS) to generate automatically a hierarchical manager with the correct coordination constraint, so that logical consistency invariants are enforced. Discrete control techniques provide tools for modelling and automatic computation of controller that is able to ensure coordination policy at

runtime. We address this with the BZR programming language [5], which is a synchronous language integrating DCS into its compilation, making user-friendly, and generating C or Java executable code. This paper shows complete control models and objectives, for the coordination of the three components.

1.3 Related work

Few works have investigated administration loops coordination. Kumar [8] proposes vManage, a coordination approach that loosely couples platform and virtualization management to improve energy savings and QoS while reducing VM migrations. Kephart [4] addresses the coordination of multiple autonomic managers for power/performance tradeoffs based on a utility function in a non-virtualized environment. Nathuji [10] proposes VirtualPower to control the coordination among virtual machines to reduce the power consumption. These works involve coordination between control loops, but these loops are applied to the managed applications. However, these works propose adhoc specific solutions that have to be implemented by hand. If new managers have to be added in the system the whole coordination manager need to be redesigned. The design of the coordination infrastructure becomes complex for the co-existence of a large number of autonomic managers.

We consider coordinating administration loops as a problem of synchronization and logical control of administration operations, for which one design methodology is to apply techniques from discrete control theory. Indeed, control theory is the classical discipline for the design of automatic controllers of devices, with the advantage of offering interesting properties on the resulting behaviour of the controlled system i.e., on its possible evolutions and on those which will be insured to be avoided. Control theory and techniques have recently began to be used for computing systems, which are quite different from the usual electro-mechanical systems. In most of the cases, continuous models are used, typically for quantitative aspects [6]. Some of these works concern power management and coordination, with quantitative models [11], or considering performance composability [7]. More recently, there has been some work, although less current, relying on models of the family of Discrete Event Systems (DES) [2], and using the notions of supervisory control [12], typically for logical or synchronization purposes [14]. This discrete control is based on models in the form of transition systems, like Petri nets or automata. For such reactive systems, there are specification and programming languages, such as synchronous languages [1] as well as formal analysis tools, such as verification and Discrete Controller Synthesis tools [9]. In this work we will use these reactive control techniques for coordinating autonomic administration loops.

1.4 Outline of the paper

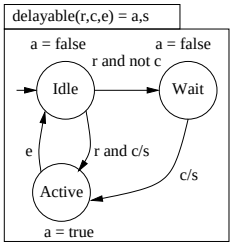
The paper is organized as follows. Section 2 presents tools we use for modelling and for declaring the coordination policy in order for DCS to automatically generate the corresponding coordination manager. Section 3 presents an application of our approach for coordinating two energy-aware managers, which controls server provisioning (Self-Sizing) and processor frequency (Dvfs), and a third autonomic manager which control server availability (Self-Repair). In this section we illustrate the design process of the coordination manager with our approach from modelling to generating the corresponding coordination manager. Section 4 present the integration and evaluation of a coordination manager to coordinate Self-Sizing and Dvfs managers execution within a replicated web-server system. Finally, in section 5 , we conclude the paper and outline directions for future work.

2. SYNCHRONOUS PROGRAMMING AND DISCRETE CONTROLLER SYNTHESIS

In this section we first briefly introduce the basics of the Heptagon language.We then describe the main feature of the BZRlanguage, which extends Heptagon with a new behavioural contract construct [5].

2.1 Automata and data-flow nodes

We first briefly introduce basics of the Heptagon language, for programming reactive systems by means of mixed synchronous data-flow equations and automata [3], with parallel and hierarchical composition. The basic behavior is that at each reaction step, values in the input flows are used, as well as local and memory values, in order to compute the values in the output flows for that step [1]. Inside the nodes, this is expressed as a set of equations defining, for each output and local, the value of the flow, in terms of an expression on other flows, possibly using local flows and state values from past steps.




```
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: graphical / textual syntax.

Figure 1 shows a small program in this language. It programs the control of a `delayable` task, which can either be idle, waiting or active. When it is in the initial Idle state, the occurrence of the `true` value on input `r` *requests* the starting of the task. Another input `c` (which will be controlled by an external controller) can either allow the activation, or temporarily block the request and make the automaton go to a waiting state. When active, the task can end and go back to the idle state, upon the notification input `e`. This `delayable` node has two outputs, `a` representing activity of the task, and `s` being emitted on the instant when it becomes active : this latter triggers the concrete starting operation in the system's API. Such automata and data-flow reactive nodes can be composed in parallel and in a hierarchical way. They can be defined and re-used by instanciations of the nodes, as illustrated in Figure 2, with two instances of Figure 1's node. They run in parallel, defined by synchronous composition (noted ";"): one global step corresponds to one local step for every equation, i.e., here, for every instance of the `delayable` node.

The compilation produces executable code in target languages such as C or Java, in the form of an initialisation

function *reset*, and a *step* function implementing the transition function of the resulting automaton, which takes incoming values of input flows gathered in the environment, computes the next state on internal variables, and returns values for the output flows. This function is called at relevant instants from the infrastructure where the controller is used.

2.2 Contracts and control in BZR

The BZR language (available at `http://bzr.inria.fr`) extends Heptagon with a new behavioral contract [5]. Its compilation involves *discrete controller synthesis* (DCS) [9]. DCS is a formal operation on automata [12]: given a FSM representing possible behaviors of a system, its variables are partitioned into controllable ones and uncontrollable ones. For a given control objective (e.g., staying invariantly inside a subset of states, considered "good"), the DCS algorithm automatically computes, by exploration of the state graph, the constraint on controllable variables, depending on current state, for any value of the uncontrollables, so that remaining behaviors satisfy the objective. This constraint is the least necessary, inhibiting the minimum possible behaviors, therefore it is called *maximally permissive*. Formalisms and algorithms are related to model checking techniques for state space exploration, and described elsewhere [9, 2].

Concretely, the BZR language allows for the declaration, using the `with` statement, of controllable variables, the value of which are not defined by the programmer. These free variables can be used in the program to describe choices between several transitions. They are then defined, in the final executable program, by the controller computed by DCS, according to the expression given in the `enforce` statement. A possibility exists, not used here, to take into account knowledge about the environment in an `assume` statement; observes can be used to have objective like always having a task $t1$ between $t2$ and $t3$. BZR compilation invokes a DCS tool, and inserts the synthesized controller in the generated executable code, which has the same structure as above: *reset* and *step* functions.

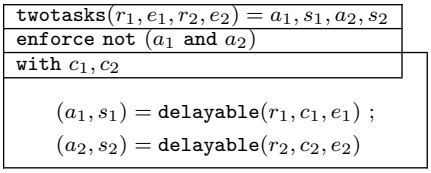


Figure 2: Exclusion enforced by DCS in BZR.

Figure 2 shows an example of contract coordinating two instances of the `delayable` node of Figure 1. The `twotasks` node has a `with` part declaring controllable variables c_1 and c_2, and the `enforce` part asserts the property to be enforced by DCS. Here, we want to ensure that the two tasks running in parallel will not be both active at the same time: `not` (A1 `and` A2). Thus, c_1 and c_2 will be used by the computed controller to block some requests, leading automata of tasks to the waiting state whenever the other task is active. The constraint produced by DCS can have several solutions: the BZR compiler generates deterministic executable code by favoring, for each controllable variable, value `true` over `false`, in the order of declaration in the `with` statement.

3. COORDINATING MANAGERS

3.1 Coordination of administration loops

Nowadays one important aspect in managing computing systems is energy-efficiency, because their deployment involves an important number of computational resources to guarantee quality of service requirement (QoS), which is energy consuming. As case-study, we propose to coordinate two pre-existing energy-aware administration loops, in order to ensure an efficient management of the energy consumption of a computing system with our approach, as well as as a self-repair loop. These administration loops are legacy code, in the sense that we do not design or re-design them.

3.1.1 Energy-aware and self-repair loops

DVFS. This controller targets single node management. It dynamically increases or decreases the CPU-frequency of a node according to the load received, compared to given thresholds for minimum and maximum values, provided minimal or maximal levels of frequency have not yet been reached. It is local to the node it manages and is implemented either in hardware or software. Ours is a user-space software and follows the on-demand policy.

Self-Sizing. This controller is for replicated servers based on a load balancer scheme. Its role is to dynamically adapt the degree of replication according to the system load. It analyzes the CPU usage of nodes to detect if the system load is in the optimal performance region. It computes a moving average of collected load monitored by sensors. When the controller receives a notification that the average exceeds the maximum threshold, and the maximum number of replication is not reached, it increases the degree of replication by selecting one of the unused nodes. If the average is under the minimum threshold and the minimum number of replication is not reached, it decreases replication by turning a node off.

Self-Repair manager. This AM deals with server availability [13]. It continuously monitors the servers, and when a failure occurs it selects an unused node and restores the server in it. This AM addresses fail-stop failure.

3.1.2 Coexistence and coordination problem

In case of the management of the energy consumption of a replicated server system, one can use both Self-Sizing and Dvfs AMs. However, the cost of increasing the CPU-frequency is less energy-consuming than increasing the number of replicated servers, and decreasing the degree of replication is more energy-saving than decreasing the CPU-frequency, which leads to the need for the Self-Sizing and Dvfs AMs to cooperate. One better usage of these managers to optimize efficiently the energy consumption relies on acting on CPU-frequency of active nodes before planning to add a new node, in case of overload of the system: this can not happen without coordination, because the AMs are independent and execute without knowledge about each-other. Also, when a server failure occurs, Self-Repair is in charge, but the other

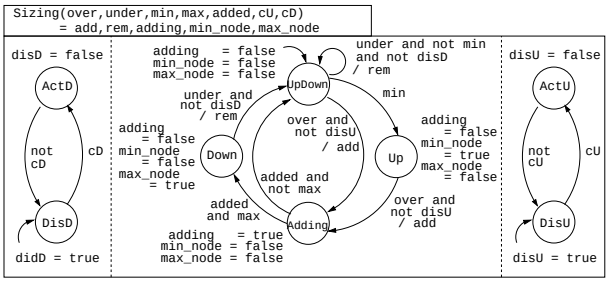


Figure 3: Self-Sizing AM behavior

AMs can be fired abusively because of transitory overload of remaining servers, or when a Load-Balancer failure occurs, during repair an apparent underload can mislead into downsizing.

We want to coordinate the Ams in order to avoid the interferences described above. We address CPU-bound application; memory-bound applications can be regulated differently. This can be specified informally by the following policy or strategy:

1. avoid upsizing unless all nodes are at maximum speed.
2. avoid downsizing when the load balancer fails.
3. avoid upsizing when a server fails.

3.2 Discrete control for coordinating the AMs

We pose the coordination problem as a control problem, first modeling the behaviors of AMs with automata, then defining controllables and a control objective in order to apply DCS.

3.2.1 Automata modeling the AMs behavior

Self-Sizing Figure 3 shows the model for the Self-Sizing AM. There are three parallel sub-automata; the two external ones manage control of the AM, and the center one is the AM, with four states, initially in `UpDown`. When an underload is notified by input `under` being `true`, and input `min` is `false`, meaning minimum number of servers is not reached, and `disD` is `false`, meaning downsizing is not disabled, then output `rem` triggers the removal of a server, and it goes back to `UpDown`. If `min` is `true`, then it goes to `Up`, where only overloads are managed upon notification by `over`: if `disU` is `false` (upsizing is not disabled) then `add` triggers the addition of a server, going to the `Adding` state, where neither adding or removal of a node can be requested until input `added` notifies termination. Then if input `max` is `false`, meaning maximum number of servers is not reached, control goes back to `UpDown`. If `max` is `true`, then it goes to `Down`, where only underloads are managed.

In this control automaton, the manager can be suspended: this is done with the local flows (resp.) `disU` and `disD` mentioned above, which, when `true`, prevent transitions where output (resp.) `add` or `rem` triggers operations. Automata on the sides of Figure 3 define the current status : disabled or not. In the case of upsizing (the downsizing is similar), initially, the automaton is in `DisU` where flow `disU` is inhibiting upsizing operations. When control input `cU` is `true`, it goes to the state `ActU` where operations are allowed, until `cU` is `false`. Hence, `cU` and `cD` define choice points, and are control interfaces made available for the coordination controller.

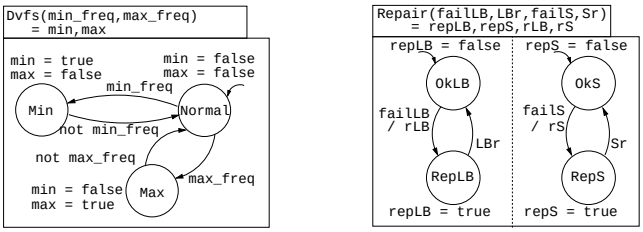


Figure 4: Dvfs (left) and Repair (right) monitoring AMs behavior

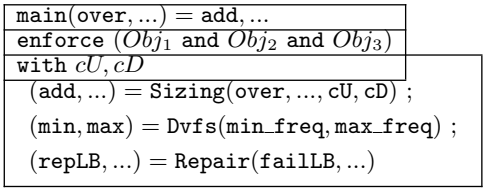


Figure 5: Coordination node: control policy on the composed models.

DVFS The DVFS managers are local to the CPUs, and they offer no control point for an external coordinator. However they are an interesting case where it is important and useful to instrument them with *observability*, because some information on their state is necessary to take appropriate coordination decision. Figure 4 shows the corresponding automaton. Intitial state is `Normal`: at least one of the set of DVFS managers can apply both CPU-frequency increase and decrease operations. When all nodes are in their maximum CPU-frequency, input `max_freq` notifies this with value `true`, and the observer goes to `Max`, where output `max` is `true`. Symmetrically, we have `min_freq` leading to `Min`, where output `min` is `true`.

Self-Repair as shown in Figure 4 (right) there are two parallel, similar automata observing the AM for the load balancer (LB) and servers (S). The right automaton concerns servers, and is initially in `OkS`, until `failS` notifies a server failure, emitting repair order `rS` and leading to state `RepS`, where `repS` is `true`, until `Sr` notifies repair termination, leading back to `OkS`. Repair of the LB is similar.

3.2.2 Coordination controller

Automaton modeling the global behavior. Figure 5 shows how it is defined, in the body of the `main` more, by the parallel composition of automata from Figures 3 and 4, thus defining all possible behaviors, in the absence of control, of the three AMs. The interface of `main` is the union of interfaces of sub-nodes, except for `cU, cD` locals used for coordination.

Contract. It is given by the declaration, in the `with` statement, of the controllables: `cU, cD`, and the Boolean ex-

pression for the control objective in the `enforce` statement: w.r.t. the policy of Section 3.1.2, the conjunction of:

1. `Obj1 = not max implies disU`
2. `Obj2 = repLB implies disD`
3. `Obj3 = repS implies disU`

With implications, DCS keeps solutions where `disU, disD` are always `true`, correct but not progressing ; but as said Section 2.2, BZR favors `true` over `false` for `cU, cD`, hence enabling the Sizing AM when possible.

Compilation. The composition of automata and the coordination policy constitute the coordination controller in the composite component. The BZR programming language is compiled, generating the corresponding Java code which, given notifications of overloads and underloads, state of the local DVFS Ams, and failures of either LB or servers, will perform reconfiguration actions, enforcing the rules of Section 3.1.2, and executing additions and removals of servers, and repairs of LB or servers. The control problem in our case study is simple, but illustrates the approach completely, and is implemented for the two energy-aware AMs.

We use DCS which computes a controller able to ensure the respect of this coordination policy by acting on `cU` and `cD` (Figure 5). The assembly of the set of models with the generated controller will constitute the coordination controller. BZR language allows to generate the composition in C or Java programming language, which allows to directly integrate it into a system.

4. EXPERIMENTAL EVALUATION

This section presents experimentations of our approach for coordinating autonomic managers. We evaluate the efficacy of the above coordination controller designed for the Self-Sizing and Dfvs managers compared to the uncoordinated coexistence of latters. The managed system is a CPU-bound system composed of replicated Tomcat servers. An Apache server is used as front to balance the workload between the active Tomcat servers. Each node that hosts a Tomcat server is equipped with a Dvfs. The objective of the coordination controller is to prevent Self-Sizing from increasing the number of active Tomcat servers when it is possible to increase the CPU frequency of current active Tomcat server nodes in case of overload ,i.e., the CPU load exceeds the maximum threshold. The maximum threshold for The CPU load is fixed at 90% for both managers.

Initially, during each execution, one Tomcat server is launched and its node is at its minimum CPU frequency. Figures 6 and 7 present executions. Curve avg_load correspond to the average CPU load the Probe for self-Sizing computes, the curves starting with load_node correspond to the CPU load computed by probe for Dvfs in each Tomcat node. The curves starting with CPUFreq_node correspond to the CPU frequency level and the curve degree_replication is the number of current Tomcat servers launched. The decrease of the load observed in the curve load_node1 (near 40%) after an overload is due to the restarting of the Apache server to update its list of Tomcat server and node1 does not receives any request from Apache.

Figure 6 shows executions in which we inject a workload supportable by one Tomcat server at higher CPU-frequency. As shown in Figure 6(b), the coordination controller prevents Self-Sizing from adding a new Tomcat server when it is possible to increase CPU-frequency compared to Figure 6(a) in which there is no coordination and both CPU-frequency and adding operations are executed.

Figure 7 shows executions with more important workload which should result to adding new tomcat server. As shown in Figure 7(b), the coordination controller does not prevent Self-Sizing from adding new replicated Tomcat server when it is necessary. The coordination controller is able to ensure the respect of coordination policy.

5. CONCLUSIONS

One major challenge in system administration is the coordination of multiple autonomic managers for coherence. In this paper we presented an approach for coordinating multiple self-management modules in a consistent manner to manage a system. This approach, based on synchronous programming and Discrete Controller Synthesis, has the advantage of generating the required controller to enable the correct by construction coordination of multiple autonomic managers. The advantages of this approach are following: *1)* High-level of programming, *2)* Correctness of the controller, *3)* Automated generation/synthesis of the controller and *4)* The latter is maximally permissive.

We have tested this approach for coordinating two energy-based self-management modules: Sizing, which manages the degree of replication for a system based on a load balancer scheme, and Dvfs, which manages the level of CPU frequency for a single node. In this case, the coordination policy was to allow Sizing to add new node only when all Dvfs modules cannot apply increase operations at all in response to the increasing load the system receives. Additionally, we control the coordination with the third loop handling repair.

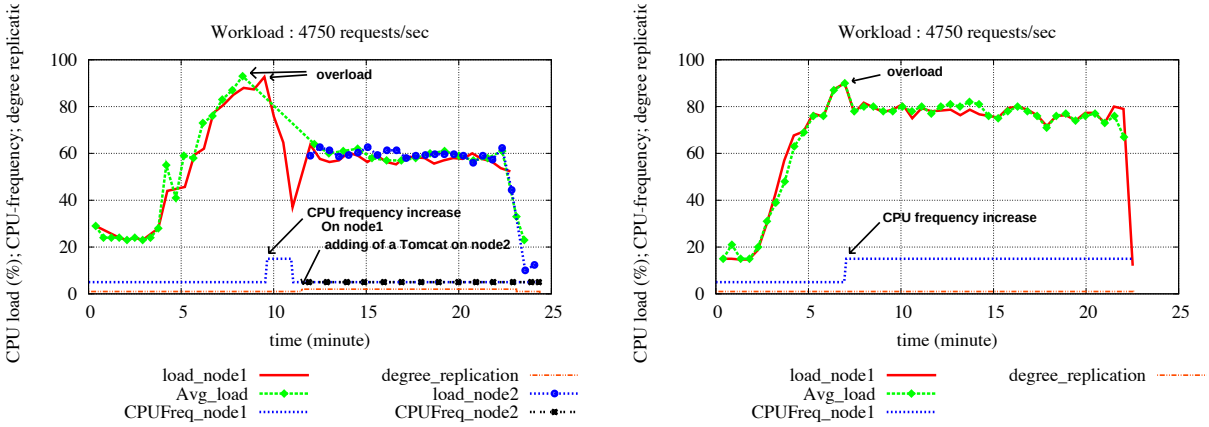
For future work, we plan to evaluate this approach for large scale coordination with more complex coordination policies and several managers, combining both self-optimization and self-regulation frequency managers with self-repair manager that heal fail-stop clustered multi-tiers system. We plan to address control scenarios more elaborate than just exclusion, and involving sequences of events and paths in the global automaton, or also weights representing cos functions associated to alternative states and paths.

6. REFERENCES

[1] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous languages twelve years later. *Proc. IEEE*, 91(1), 2003.

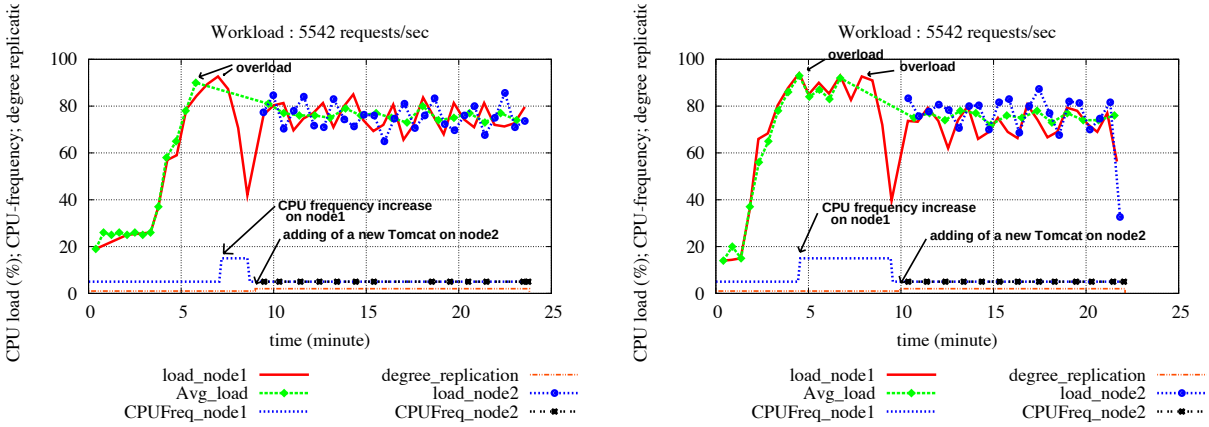
[2] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems.* 2006.

[3] J.-L. Colaço, B. Pagano, and M. Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. In *ACM Int. Conf. on Embedded Software (EMSOFT'05)*, Sept. 2005.



(a) uncoordinated coexistence (b) coordinated coexistence

Figure 6: Workload supportable by one Tomcat Server at higher CPU-frequency



(a) uncoordinated coexistence (b) coordinated coexistence

Figure 7: Workload necessitating two Tomcat servers

[4] R. Das, J. Kephart, C. Lefurgy, G. Tesauro, D. Levine, and H. Chan. Autonomic multi-agent management of power and performance in data centers. In *Proc. 7th int. joint conf. on Autonomous agents and multiagent systems: industrial track*, AAMAS, 2008.

[5] G. Delaval, H. Marchand, and E. Rutten. Contracts for modular discrete controller synthesis. In *Conf. on Languages, Compilers and Tools for Embedded Systems, LCTES*, 2010.

[6] J. Hellerstein, Y. Diao, S. Parekh, and D. Tilbury. *Feedback Control of Computing Systems*. Wiley, 2004.

[7] J. Heo, P. Jayachandran, I. Shin, D. Wang, T. Abdelzaher, and X. Liu. Optituner: On performance composition and server farm energy minimization application. *IEEE Trans. Parallel Distrib. Syst.*, 22(11), 2011.

[8] S. Kumar, V. Talwar, V. Kumar, P. Ranganathan, and K. Schwan. vManage: loosely coupled platform and virtualization management in data centers. In *Proc. 6th int. conf. Autonomic computing*, ICAC, 2009.

[9] H. Marchand, P. Bournai, M. L. Borgne, and P. L. Guernic. Synthesis of discrete-event controllers based on the Signal environment. *Discrete Event Dynamic System: Theory and Applications*, 10(4), Oct. 2000.

[10] R. Nathuji and K. Schwan. Virtualpower: coordinated power management in virtualized enterprise systems. In *21st Operating Systems Principles*, SOSP, 2007.

[11] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu. No power struggles: Coordinated multi-level power management for the data center. In *ASPLOS*, 2008.

[12] P. Ramadge and W. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. on Control and Optimization*, 25(1):206–230, Jan. 1987.

[13] S. Sicard, F. Boyer, and N. D. Palma. Using components for architecture-based management: the self-repair case. In *30th Int. Conf. on Software Engineering (ICSE)*, 2008.

[14] Y. Wang, T. Kelly, and S. Lafortune. Discrete control for safe execution of it automation workflows. In *2nd Eu. Conf. on Computer Systems*, EuroSys, 2007.