Component-Based Autonomic Managers for Coordination Control[⋆]

Soguy Mak Karé Gueye[1], Noël de Palma[1], and Eric Rutten[2]

[1] LIG / UJF, Grenoble, France
{soguy-makkare.gueye,noel.depalma}@imag.fr
[2] LIG / INRIA, Grenoble, France
eric.rutten@inria.fr

Abstract. The increasing complexity of computing systems has motivated the automation of their administration functions in the form of autonomic managers. The state of the art is that many autonomic managers have been designed to address specific concerns, but the problem remains of coordinating them for a proper and effective global administration. In this paper, we define controllable autonomic managers encapsulated into components, and we approach coordination as their synchronization and logical control. We show that the component-based approach supports building such systems with introspection, adaptivity and reconfiguration. We investigate the use of reactive models and discrete control techniques, and build a hierarchical controller, enforcing coherency properties on the autonomic managers at runtime. One specificity and novelty of our approach is that discrete controller synthesis performs the automatic generation of the control logic, from the specification of an objective, and automata-based descriptions of possible behaviors. Experimental validation is given by a case-study where we coordinate two self-optimization autonomic managers and self-repair in a replicated web-server system.

Keywords: Adaptive and autonomic systems, Coordination models and paradigms, Software management and engineering, Discrete control.

1 Coordinating Autonomic Loops

1.1 The Need for Coordination Control

The administration of distributed systems is automated in order to avoid human manual management because of cost, duration and slowliness, and error-proneness. Autonomic administration loops provide for management of dynamically reconfigurable and adaptive systems. The architecture of an autonomic system is a loop defining basic notions of Managed Element (ME) and Autonomic Manager (AM). The ME, system or resource is monitored through sensors. An analysis of this information is used, in combination with knowledge about the system, to plan and decide upon actions. These reconfiguration operations

[⋆] This work is supported by the ANR INFRA project Ctrl-Green.

are executed, using as actuators the administration functions offered by the system API. Self-management issues include self-configuration, self-optimization, self-healing (fault tolerance and repair), and self-protection. In this work we consider AMs for self-optimization and self-repair.

Complex and complete autonomic systems feature numerous different loops for different purposes, managing different dimensions. This is causing a problem of co-existence, because it is hard to avoid inconsistencies and possible interferences. Therefore, coordination is recognized as an important challenge, not completely solved by present research, and requiring special attention [11]. Coordinating AMs can be seen as the problem of synchronization and logical control of administration operations that can be applied by AMs on the MEs in response to observed events. Such an additional layer, above the individual administration loops, constitutes a coordination controller. AMs are considered as being MEs themselves, with an upper-level AM for their coordination. The state of the art in designing these hierarchical coordination controllers is to develop them, e.g., using metrics dedicated to the aspect around which coordination is defined, like energy and performance [6]. As a hand-made methodology, this remains complex and error-prone, and hard to re-use.

1.2 Coordination as Discrete Control of Components

Our contribution is an automated methodology for the coordination control of autonomic managers. A novelty is that it provides for concrete design assistance, in the form of the automatic generation, from a Domain-Specific Language (DSL), of the control logic, for a class of coordination problems expressed as invariance properties on the states of the AMs. The benefit is manifold: the designer's work is eased by the automatic generation, the latter is done based on formal techniques insuring correctness of the result, and the re-use of designs is facilitated by the automated generation of the control logic.

We identify the needs for the coordination of AMs, which have to be instrumented in order to be observable (e.g., current state or execution mode) and controllable (e.g., suspend activity), so that they can be coordinated. For this, component-based approaches provide us with a well-structured framework. Each component is defined separately, independently of the way it can be used in assemblies. They also have to be equipped with a model of their behavior, for which we use Finite state Machines (FSM), also called automata.

We specify the coordination policy as a property between the states of the AMs, like for example mutual exclusion, which must be kept invariant by control. Given this control objective, and the behavioral model FSMs, we use techniques stemming from Control Theory to obtain the controller. Such techniques have recently been applied to computing systems, especially using classical continuous control models, typically for quantitative aspects [10]. The advantage is that they ensure important properties on the resulting behavior of the controlled system e.g., stability, convergence, reachability or avoidance of some evolutions. We use discrete control techniques, especially Discrete Controller Synthesis (DCS) [5], well adapted for logical or synchronization purposes. In this work, we focus on software engineering and methodology; formal aspects available elsewhere [8] are not in the scope of this paper. Our work in this paper builds upon previous preliminary results [9] where we had considered a specific experiment in coordinating two administration loops (self-sizing and DVFS), implemented and tested as a simple prototype. Here, our new contributions are:

1. a generalized method, leveraged to the level of a component-based approach, to design controllable AMs, with a DSL for describing administration behavior as automata, with controllable points at the interfaces;
2. a method for the specification and generation of coordination controllers enforcing invariance properties, based on the formal DCS technique;
3. a fully implemented experimental validation of a case-study coordinating self-optimization autonomic managers (self-sizing and DVFS), completed with a new one for self-repair; we thus demonstrate reusability by combining the same sizing loop in a new context, with a new coordination policy.

In the remainder of the paper we will make recalls in Section 2 on component-based approaches, and the reactive language BZR. The kernel of our contribution is first in Section 3 where we describe the method for instrumenting a basic AM component in order to become an ME itself; and then in Section 4 where we propose construction methods on these bases for composite components, where coordination of the AMs is managed. An additional contribution is the experimental validation by the case study in Section 5, with self-optimization and self-repair autonomic managers in a replicated web-server system.

2 Background: Components and Reactive Control

2.1 Component Model

In classical component models, a component is a run-time entity that is encapsulated, and that has a distinct identity. A component has one or more interfaces. An interface is an access point to a component, that supports a finite set of service. Interfaces can be of two kinds: (i) server interfaces, which correspond to access points accepting incoming service calls, and (ii) client interfaces, which correspond to access points supporting outgoing service calls. Communication between components is only possible if their interfaces are connected through an explicit binding. A component can be composite, i.e. defined as an assembly of several subcomponents, or primitive, i.e. encapsulating an executable program. The above features (hierarchical components, bindings between components, strict separation between component interfaces and component implementation) are representative of a classical component model.

Reflexive component models extend these features to allow well scoped introspection and dynamic reconfiguration capabilities over component's structure. They perfectly match our needs since they are endowed with controllers, which provide access to component internals, allowing for component introspection and control of their behavior. These general reflexive component-based concepts

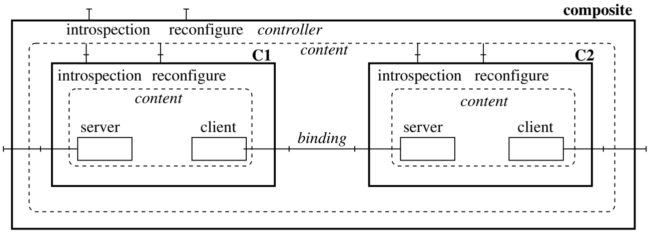
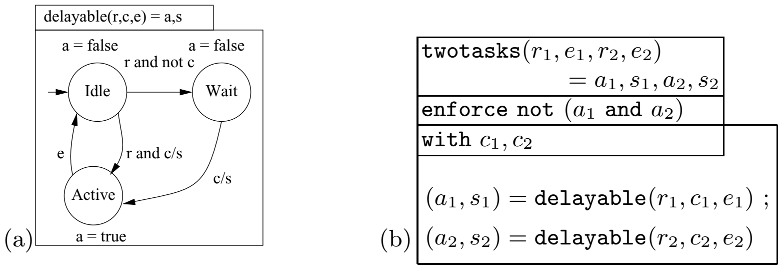


Fig. 1. A composite component



```
twotasks(r1, e1, r2, e2)
                    = a1, s1, a2, s2
enforce not (a1 and a2)
with c1, c2

(a1, s1) = delayable(r1, c1, e1) ;
(a2, s2) = delayable(r2, c2, e2)
```

Fig. 2. Delayable task: (a) graphical syntax ; (b) exclusion contract

have a number of particular instances, amongst which Fractal [4]. This reflexive component model comes with several useful forms of controllers, which can be combined and extended to yield components with different control interfaces. This includes controllers for: (i) attributes, which are configurable through getter and setter methods, (ii) bindings, to (dis)connect client interfaces to server interfaces, (iii) contents, to list, add and remove subcomponents and (iv) lifecycle, to give an external control over component execution, including starting and stopping execution.

Figure 1 shows an example of a composite component with: functional and control interfaces, a content built from two subcomponents (C1 and C2) that implement the functional interfaces and, a controller that implements the control interfaces. The controller in the composite can react e.g., to a reconfiguration access by removing subcomponent C2: i.e., stopping it, unbinding it from C1 and from the composite's client interface, re-binding C1's client to the composite, and uninstalling C2's code. The experimental validation in Section 5 relies on the Java-based reference implementation of Fractal (Julia): in this framework, a component is represented by a set of Java objects.

2.2 Reactive Languages and BZR

Automata and Data-Flow Nodes. We briefly introduce, with examples, the basics of the Heptagon language; due to space limitations, formal definitions are not recalled [8]. It supports programming of nodes, with mixed synchronous data-flow equations and automata, 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. 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. Figure 2(a) 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` can either allow the activation, or temporarily block the request and make the automaton go to a `waiting` state. Input `e` notifies termination. The outputs represent, resp., `a`: activity of the task, and `s`: triggering starting operation in the system's API.

Such automata and data-flow reactive nodes can be reused by instantiation, and composed in parallel (noted `";"`) and in a hierarchical way, as illustrated in the body of the node in Figure 2(b), with two instances of the `delayable` node. They run in parallel: one global step corresponds to one local step for every node, with possible communication through share flows. 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.

Contracts and Control. BZR (`http://bzr.inria.fr`) extends Heptagon with a new behavioral contract [8]. Its compilation involves *discrete controller synthesis* (DCS), a formal operation [5] on a FSM representing possible behaviors of a system, its variables 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 inhibiting the minimum possible behaviors, therefore it is called *maximally permissive.* Algorithms are related to model checking techniques for state space exploration; they are exponential in the states, but can manage our models built at coarse grain abstraction level.

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 during compilation, according to the expression given in the `enforce` statement. 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(b) shows an example of contract coordinating two instances of the `delayable` node of Figure 2. 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 Designing Controllable AMs

3.1 Design of an AM

Controllable AM components have to share features of an AM, as well as of an ME, as we recall here. Basic features required for a system to be an ME managed in an autonomic fashion have been identified in previous work e.g., in the context of component-based autonomic management [13]. It must be observable and controllable, and we need to know its possible behaviors, for which we use automata models. *Observability* goes through a capacity of introspection, exhibiting internals to the outside world, particularly to an AM in charge of management, in a context with other MEs. In the autonomic framework, it corresponds to the sensors. In components as in Figure 1, it is in introspection interfaces that can be called from outside. At implementation level, it can be for example `get` functions accessing internal variables. *Controllability* is defined by the capacity to change features inside the ME from the outside. In the autonomic framework, it corresponds to the actuators. In components, it corresponds to reconfiguration actions e.g., mode switching, or attribute updating, as mentioned in Section 2.1. At implementation level, it can be done in different ways, from interceptors on the interfaces in Java, to actions modifying the structure as with Fscript [7]. In short, the system is equipped with a library of sensors/monitors and actions, with an API given by the system, and programmed by the designer.

The AM is a reactive node, transforming flows of sensor observations into flows of reconfiguration actions. Internally, it features decision-making mechanisms, which can range from simple threshold triggers to elaborate MAPE-K. They can involve quantitative measures and continuous control, or logical aspects modeled as FSMs. Therefore a reactive language such as BZR can be used as a DSL for the decision part on the AMs: it offers high-level programming, as well as formal tools to bring safe design and guaranteed behaviors. In components, the AM is the controller in Figure 1, or it can be an additional component in the assembly of MEs, bound with the appropriate control interface of other subcomponents. At implementation level, it reacts to notifications by treating them, depending on applications, in FIFO order or considering most recent values. It calls the action functions of the controlled MEs as above, or modifies the composite by ME additions or removals.

We aim at combining the two, to have AMs which can be manipulated as MEs, which involves the same features: making AMs observable and controllable.

3.2 Controllable AMs

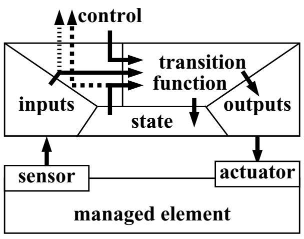
To make an AM *observable*, relevant states of the component are exhibited to the outside. They convey information necessary for coordination decisions. Automata represent states of components, w.r.t their activity and/or ability to be coordinated, but independently of the coordination policy. The most basic case, for an AM as for any ME, is to distinguish only idle state (where the AM is not performed) and activity (with actual effect on the ME). Beyond this, one can have a more refined, *grey box* model, distinguishing states corresponding to phases in a sequence, where the AM can be stopped or suspended without loss of consistency. States support monitoring e.g., to explicitly represent that some bound is reached, such as minimum number of a resource, or maximum capacity.

To make an AM *controllable*, we use the transitions between states, which are guarded by conditions, and can fire reconfiguration actions. They represent possible choices offered by the AM between different reactions, in order for an external coordinator to choose between them. They offer control points to be used according to a given policy. Hence they give the controllability of the AM.

Figure 3(a) shows how an FSM can be an instantiation of the general autonomic loop, with knowledge on possible behaviors represented as states, and analysis and planning as the automaton transition function. In the autonomic framework observability comes through additional outputs, as shown by dashed arrows in Figure 3(a) for an FSM AM, exhibiting (some) of the knowledge and sensor information. Controllability corresponds to having the AM accept additional input for control. Its values can be used in the guards to guide choices between different transitions. Such automata are specified in BZR, benefiting from its modular structures and compiler. In BZR, a transition with a condition `e and c`, where `e` is a Boolean expression, will be taken only if `c` is `true`: it can inhibit or enable the transition.

In components, automata are associated with the component controller in the membrane as in Figure 3(b), and are updated at runtime to reflect the current state of component. Automata outputs trigger actions in reconfiguration control interfaces, such as `"stop AM"`, or convey information to upper layers.

At implementation level, the modular compilation of BZR generates for each node two methods (in C or Java): *reset* and *step. reset* initializes internal variables and is called only once. *step* makes the internal variable reflect continuously

(a) 

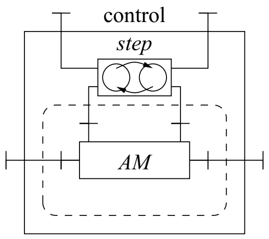
(b) 

Fig. 3. Controllable AM: (a) case of a FSM manager; (b) wrapped component

the state of the component. Each call to *step* takes as inputs relevant information from inside, through the bindings to sub-components, or from outside, through control interfaces of the composite, and control interfaces feature functions accessing returned values. Returned values trigger reconfiguration actions.

4 Coordinated Assembly of Controllable AMs

4.1 Coordination Behaviors, Objective, and Controller

We now present how such controllable AMs can be assembled in composites, where the coordination is performed in a hierarchical framework. The AMs and other components involved in the coordination (which can be composites themselves, recursively) are grouped into a composite. It orchestrates their execution, using the possibilities offered by each of them, through control interfaces, in order to enforce a coordination policy or strategy. We base our approach on the hierarchical structure in Figure 4: the top-level AM coordinates lower-level ones. We describe the problem with one level of hierarchy, but it can be recursive.

The first thing is to construct a model of the global behavior of the assembly of components. This is done using the local automata from each of the concerned subcomponents: they are composed in parallel in a new BZR node: we then have a model of all the possible behaviors in the absence of control. Controllable variables must be identified and designated i.e., the possible choice points, which will be the actuators offered to the discrete controller to enforce the coordination.

Specification of the coordination policy is done by associating a BZR contract to this global behavior. It can make reference to the information explicitly exhibited by subcomponents, and to inputs of the coordinator. The control objective is to restrict possible behaviors to states where a Boolean expression will remain invariantly true, whatever the sequences of uncontrollables.

Once we have a global automaton model of the behavior, a list of controllables, and a control objective, we manually encode the control problem into the BZR language as a contract. The BZR language compiler, and its associated

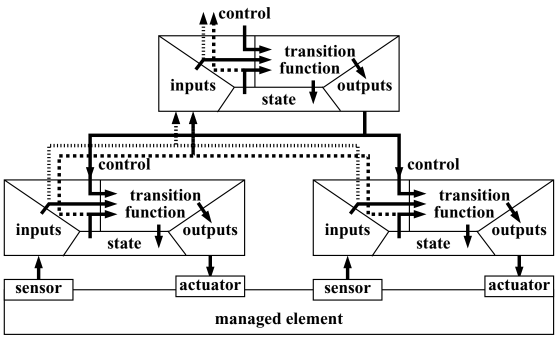


Fig. 4. Autonomic coordination for multiple administration loops

DCS tool, solves the control problem by automatically computing the controller ensuring, by automated formal computation, a correct behavior respecting the contract. It must be noted that as in any control or constraint problem, it may happen that the system does not offer enough controllability for a solution to exist. Either it lacks actuators, or sensors, or explicit states to base its decisions on. In these cases, the DCS fails, which constitutes a diagnostic of non-existence of a solution for the current coordination problem. Then the system has to be redesigned, either by relaxing the contract, or by augmenting controllability. Another meaningful point is that the BZR language features hierarchical contracts [8], where the synthesis of a controller can use knowledge of contracts enforced by sub-nodes, without needing to go into the details of these nodes. This favors scalability by decomposing the synthesis computation. It also corresponds very closely to the hierarchical structure proposed here, and can be exploited fully.

4.2 Hierarchical Architecture

In the autonomic framework our approach defines a hierarchical structure as shown in Figure 4. Given that AMs have additional outputs exhibiting their internals, and additional inputs representing their controllability, an upper-level AM can perform their coordination using their additional control input to enforce a policy. Considering the case of FSM managers makes it possible to encode the coordination problem as a DCS problem. The transition function of this upper-level AM is the controller synthesized by DCS.

In components, this translates to a hierarchical structure where a composite features a controller, bound with the outside and with subcomponents, through control interfaces. This way it can use knowledge on life-cycles of subcomponents in order to decide on reconfiguration actions to execute for their coordination.

At implementation level, there is an off-line phase, where the BZR compilation is used. It involves extracting the relevant automata from AMs descriptions : to each of them is associated a BZR node. These nodes are used to compose a new BZR program, with a contract, declaring controllable variables in the `with` statement. The policy has to be specified in the `enforce` statement. The complete BZR program is compiled, and code is generated as previously described. The resulting *step* function is integrated at the coordinator component level, calling steps from sub-components. At run-time, in our implementation, the controller in the composite component is responsible of executing the *step* method call: it gathers the necessary information from the sub-components, then executes the *step*, which interacts with the local *step* functions in subcomponents. It then triggers and propagates reconfiguration commands to sub-components based on the returned result, as discussed before.

4.3 Change Coordination Policy

This separation of concerns, between description of possible behaviors, and coordination policy, favors the possibility to change policy without changing the individual components. For example, if we wanted to change priorities between

AMs, this would impact the contract but not the individual components. On the other hand, if for the same overall policy we wanted to use other components with a different implementation of the same management, we can switch an AM by another one and recompute the coordination controller. The fact that our approach is modular, and that it uses automated DCS techniques, facilitates reuse of components, and also the modification of systems, by re-compilation.

5 Case Study: Coordinating Administration Loops

5.1 Description of the Case Study

We consider the coordination of two administration loops for resource optimization (self-sizing) and server recovery (self-repair) for the management of a replicated servers system based on the load balancing scheme.

Self-sizing. It addresses the resource optimization of a replication-based system. It dynamically adapts the degree of replication depending on the CPU load of the machines hosting the active servers. Its management decisions rely on thresholds, `Minimum` and `Maximum`, delimiting the optimal CPU load range. It computes a moving average of the collected CPU load (`Avg_CPU`) and performs operations if the average load is out of the optimal range: Adding operations (`Avg_CPU` $>=$ `Maximum`), removal operations (`Avg_CPU` $<=$ `Minimum`).

Self-repair. It addresses fail-stop failure of a machine hosting a single or replicated server. Its management decisions rely on Heartbeat. When the machine hosting the server becomes non-responsive, it redeploys the server towards another machine selected from an unused machine pool. Then it updates the new server configuration from the configuration of the failed server and starts it.

Co-existence and Coordination Problem. Failures can trigger incoherent management decisions by self-sizing. The failure of the load balancer can cause an underload of the replicated servers since the latter do not receive requests until the load balancer is repaired. The failure of a replicated server can cause an overload of the remaining servers because they receive more requests due to the load balancing. A strategy to achieve an efficient resource optimization could be to (1) *avoid removing a replicated server when the load balancer fails*, and (2) *avoid adding a replicated server when one fails.*

5.2 Modelling and Control for Coordinating the AMs

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

Modelling AMs. Figure 5 shows the automata modelling both the behaviors and the control of the self-sizing manager. The two external ones model the control of the adding (resp. removal) operations. This is done with the local flows `disU` (resp. `disD`), which, when `true`, prevent transitions where output `add`

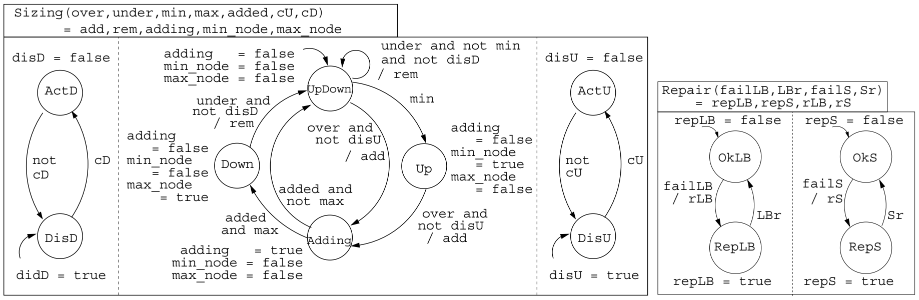


Fig. 5. Managers models : self-sizing (left) and self-repair (right)

(resp. `rem`) triggers operations. The center one models the behaviors. Initially in the `UpDown` state, when an Overload occurs and adding operations are allowed, the manager requests a new server, and goes to the `Adding` state and can no longer perform operations. It awaits until the server becomes active (`node_added` is `true`), and returns back to the `UpDown` state or goes to the `Down` state if the degree of replication is maximal. The `Down` state is left once one server is removed upon an `Underload` event. The `Up` state is the state in which the degree of replication is minimal. The manager cannot remove server but can add server upon `Overload` event if allowed.

Figure 5 shows the automata modelling the behaviors of the self-repair managers for the load balancer (LB) and the replicated servers (S). The right automaton concerns servers, and is initially in `OkS`. When `failS` is `true`, it emits repair order `rS` and goes to the `RepS` state, where `repS` is `true`. It returns back to `OkS` after repair termination (`Sr` is `true`). Repair of the LB is similar.

Coordination Controllers. The automata are composed in order to have the global behavior model, and a contract specifies the coordination policy. Automata in Figure 5 are composed. The policies (1) and (2) in Section 5.1 give a contract as follows:

```
enforce ((repLB implies disD) and (repS implies disU)) with Cu,Cd
```

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

Compilation. The BZR compiler generates the corresponding Java code of the BZR program. Two main methods allow to interact with the program: `reset` for initializing it and `step`. The latter takes as parameters all the automata inputs and returns all the automata outputs.

5.3 Experiments

We apply our approach for coordinating one instance of self-sizing and two instances of self-repair managers for the management of a replicated-based system.

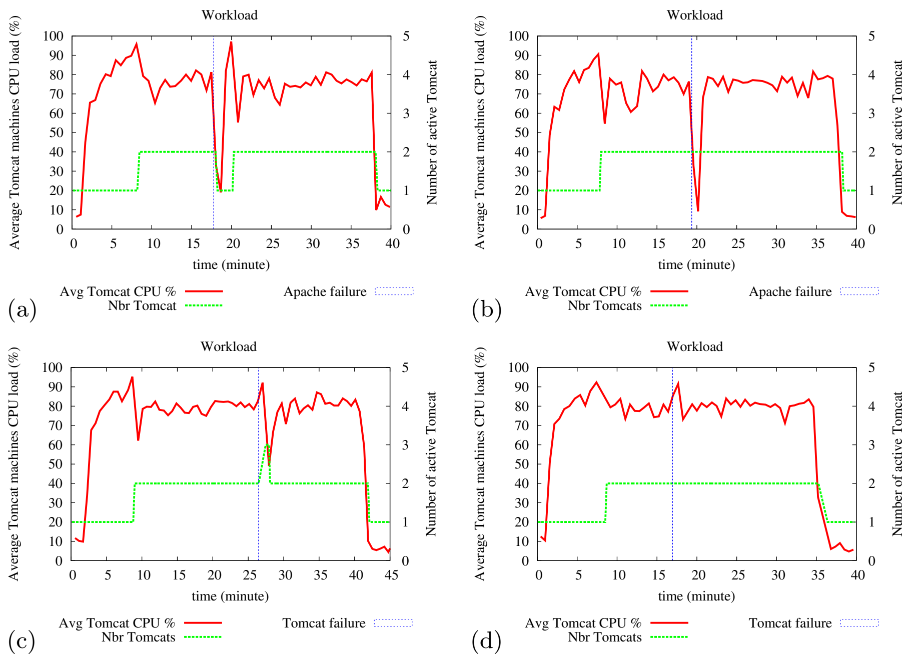


Fig. 6. Apache (a,b) and Tomcat (c,d) server failure ; uncoordinated and coordinated

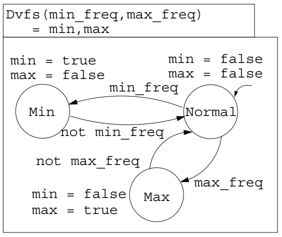
The system is composed of one Apache server (self-repair) and replicated Tomcat servers (self-sizing and self-repair). Homogeneous machines are used to host the Tomcat servers. Initially during each execution, the system is started with one Apache server and one Tomcat server. We inject a workload in two phases, a ramp-up workload followed by a constant workload. We wait until there are two active Tomcat servers before injecting a failure.

Figure 6(a,b) shows executions in which the Apache server fails, which causes a decrease of the CPU utilization of the machines hosting the Tomcat servers. In the uncoordinated execution (Figure 6(a)), this leads to the removal of a Tomcat but once the Apache is repaired another Tomcat is added. In the coordinated execution (Figure 6(b)) the decrease does not lead to the removal of a Tomcat since the coordination controller inhibits removal operations because of the failure. Figure 6(c,d) shows executions in which a Tomcat server fails, which causes an increase of the CPU utilization of the remaining Tomcat servers. While in the uncoordinated execution (Figure 6(c)) this increase leads to adding a new tomcat (which is removed after the repair), in the coordinated execution (Figure 6(d)) no adding operation is performed. The adding operations are inhibitted by the coordination controller which is aware of the Tomcat failure.

5.4 Reusability of Models: Integrating the Dvfs Manager

We consider the management of the CPU frequency for more resource optimization. Each machine hosting a replicated server is equipped with a Dvfs manager.

(a)



(b)

```
main(over, ...) = add, ...
enforce (Obj1 and Obj2 and Obj3)
with cU, cD
    (add, ...) = Sizing(over, ..., cU, cD) ;
    (min, max) = Dvfs(min_freq, max_freq) ;
    (repLB, ...) = Repair(failLB, ...)
```

Fig. 7. DVFS model, and its Coordination with sizing and repair

This latter dynamically increases or decreases the CPU-frequency of the machine depending on the CPU load. Its management decisions, like self-sizing, rely on thresholds, `Minimum` and `Maximum`, delimiting the optimal CPU load range. In a context where Dvfs are activated, a strategy to improve resource optimization could be to delay as long as possible adding a new server when the machines hosting active servers are not in their maximum CPU frequency: (3)*avoid adding unless all machines are at maximum speed.*

We model the global states of the set of Dvfs in Figure 7(a). Initially in the `Normal` state, when the input `min_freq` (resp. `max_freq`) is `true`, it goes to the `Min` (resp. `Max`) state when the CPU frequency of all machines hosting replicated servers is minimal (resp. maximal) and the output `min` (resp. `max`) is `true`. It returns back the `Normal` state if the CPU frequency of at least, one of the machines is neither maximal nor minimal. Coordinating the three AMs consists in adding the automaton for the Dvfs in the composition with the following contract to be enforced: (*not max* ***implies*** *disU*) representing the policy (3), as shown in Figure 7(b), with Obj_i representing policy (i) ($i = 1, 2, 3$).

6 Related Work and Discussion

The general question of coordinating autonomic managers remains an important challenge in Autonomic Computing [11] although it is made necessary in complete systems with multiple loops, combining dimensions and criteria. Some works propose extensions of the MAPE-K framework in order to allow for synchronization [15], which can be e.g., through the access to a common knowledge [2]. A distinctive aspect of our approach is to rely on FSM-based behavioral models, amenable to formal techniques like verification or DCS. Coordination of multiple energy management loops is done in various ways, e.g., by defining power vs. performance tradeoffs based on a multi-criteria utility function in a non-virtualized environment [6]. These approaches seem to require modifying AMs for their interaction, and to define the resulting behavior by quantitative integration of the measure and utilities, which relies on intuitive tuning values, not handling logical synchronization aspects. We coordinate AMs by controling their logical state, rather than modifying them.

Component-based frameworks are classically associated with implementations offering APIs for sensing and actuating. For example FScript [7] is a middleware layer offering relatively high level support for programming complex reconfiguration actions. However, there is no support in expressing explicitly and directly the set of configurations or modes, and the switches between them: they have to be managed by tedious manual programming with side effects. Our work proposes higher level programming of control aspects, with first-class language constructs explicitly representing control states and events. The formally based semantics of our language support enables concrete use of verification and synthesis techniques. Relating component-based frameworks and formal models is widely done, e.g. relying on process calculi for abstract reasoning on composition constructs. Closer to the concrete modeling of behavioral aspects, some modeling approaches offer access to verification as model checking [3]. We specifically concentrate on reconfiguration control, not on the general component approach, and it relates this aspect of Fractal with the synchronous approach to reactive systems, and its support for correct program design, particularly DCS. [1] proposes a framework allowing multiple autonomic managers for the management of a Behavioral Skeleton. The coordination of the managers is ensured through consensus which seems to be manually implemented, which can become complex. [12] proposes a component-based programming framework to build an autonomic application as the composition of autonomic components, with agents based on rules with priority level for conflicting decisions resolution.

Concerning decision and control, some approaches rely upon Artificial Intelligence and planning [14] which has the advantage of managing situation where configurations are not all known in advance, but the corresponding drawback of costly run-time exploration of possible behaviors, and lack of insured safety of resulting behaviors. Our work adheres to the methodology of control theory, and in particular DES, applied to computing systems [10]. Compared to traditional error-prone programming followed by verification and debugging, such methods bring correctness by design of the control. Particularly, DCS offers automated generation of the coordination controller, facilitating design effort compared to hand-writing, and modification and re-use (see Section 4.3). Also, maximal permissivity of synthesized controllers is an advantage compared to over-constrained manual control, impairing performance even if correct. Applications of DCS to computing systems have been rare until now e.g. to address deadlock avoidance [16]. Compared to this, we consider more user-defined objectives.

7 Conclusion

We address the problem of coordinating multiple AMs using a component-based approach. We define a method for the componentization of AMs as MEs, equipped with automata-based behavioral models. The control of coordinated AMs is based on a formal control technique : DCS, which performs automatic generation. It support addition of new AMs, without re-designing the whole system, and is encapsulated in a design process for non-expert users. A fully implemented case-study, for sizing and repair AMs, gives experimental validation.

In perspective, we are busy evaluating our approach for coordination of a variety of autonomic administration loops in the context of an industrial data-center. We have ongoing work on managing coordination in a multi-tier system, involving intra-tier and inter-tier coordination, hierarchically. We consider using more elaborate control techniques, with DCS for optimal control, or the distribution of the controllers. Finally, we have ongoing work on programming language support for our methodology, integrating the whole approach in the compilation process of a component-based language.

References

1. Aldinucci, M., Danelutto, M., Kilpatrick, P., Xhagjika, V.: LIBERO: A framework for autonomic management of multiple non-functional concerns. In: Guarracino, M.R., Vivien, F., Träff, J.L., Cannatoro, M., Danelutto, M., Hast, A., Perla, F., Knüpfer, A., Di Martino, B., Alexander, M. (eds.) Euro-Par-Workshop 2010. LNCS, vol. 6586, pp. 237–245. Springer, Heidelberg (2011)
2. Alvares de Oliveira Jr., F., Sharrock, R., Ledoux, T.: Synchronization of multiple autonomic control loops: Application to cloud computing. In: Sirjani, M. (ed.) COORDINATION 2012. LNCS, vol. 7274, pp. 29–43. Springer, Heidelberg (2012)
3. Barros, T., Boulifa, R., Cansado, A., Henrio, L., Madelaine, E.: Behavioural Models for Distributed Fractal Components. Res. Report RR-6491, INRIA (2008)
4. Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.-B.: The fractal component model and its support in java. Software – Practice and Experience (SP&E) 36(11-12) (September 2006)
5. Cassandras, C., Lafortune, S.: Introduction to Discrete Event Systems. Springer-Verlag New York, Inc., Secaucus (2006)
6. Das, R., Kephart, J.O., Lefurgy, C., Tesauro, G., Levine, D.W., Chan, H.: Autonomic multi-agent management of power and performance in data centers. In: Proc. Conf. AAMAS (2008)
7. David, P.-C., Ledoux, T., Léger, M., Coupaye, T.: FPath and FScript: Language support for navigation and reliable reconfiguration of fractal architectures. Annals of Telecommunications 64(1), 45–63 (2009)
8. Delaval, G., Marchand, H., Rutten, E.: Contracts for modular discrete controller synthesis. In: Proc. Conf. LCTES (2010)
9. Gueye, S.M.K., de Palma, N., Rutten, E.: Coordinating energy-aware administration loops using discrete control. In: Proc. Conf. ICAS (2012)
10. Hellerstein, J., Diao, Y., Parekh, S., Tilbury, D.: Feedback Control of Computing Systems. Wiley-IEEE (2004)
11. Kephart, J.: Autonomic computing: The first decade. In: Proc. Conf. ICAC (2011)
12. Liu, H., Parashar, M., Hariri, S.: A component based programming framework for autonomic applications. In: Proc. ICAC (2004)
13. Sicard, S., Boyer, F., De Palma, N.: Using components for architecture-based management: the self-repair case. In: Proc. Conf. ICSE (2008)
14. Sykes, D., Heaven, W., Magee, J., Kramer, J.: Plan-directed architectural change for autonomous systems. In: Proc. WS SAVCBS (2007)
15. Vromant, P., Weyns, D., Malek, S., Andersson, J.: On interacting control loops in self-adaptive systems. In: Proc. Conf. SEAMS (2011)
16. Wang, Y., Lafortune, S., Kelly, T., Kudlur, M., Mahlke, S.: The theory of deadlock avoidance via discrete control. In: Proc. ACM POPL (2009)