Modular Coordination of Multiple Autonomic Managers [*]

Gwenaël Delaval, Soguy Mak-Karé Gueye, Noël De Palma
Univ. Grenoble Alpes, LIG
Grenoble, France
gwenael.delaval@inria.fr, soguy-mak-kare.gueye@inria.fr, noel.depalma@imag.fr

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

ABSTRACT

Complex computing systems are increasingly self-adaptive, with an autonomic computing approach for their administration. Real systems require the co-existence of multiple autonomic management loops, each complex to design. However their uncoordinated co-existence leads to performance degradation and possibly to inconsistency. There is a need for methodological supports facilitating the coordination of multiple autonomic managers. In this paper we propose a method focusing on the discrete control of the interactions of managers. We follow a component-based approach and explore modular discrete control, allowing to break down the combinatorial complexity inherent to the state-space exploration technique. This improves scalability of the approach and allows constructing a hierarchical control. It also allows re-using complex managers in different contexts without modifying their control specifications. We build a component-based coordination of managers, with introspection, adaptivity and reconfiguration. We validate our method on a multiple-loop multi-tier system.

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 Engineering**]: Software Architectures—*Domain-specific architectures, languages, patterns*

Keywords

Autonomic computing; Component dynamic adaptation; Automated management; Control loops; Formal methods; Self-adaptive systems; Software reuse

[*]This research is partly supported by the FSN Datalyse project and ANR INFRA (ANR-11-INFR 012 11) under a grant for the project Gtrl-Green.

CBSE'14, June 30–July 4, 2014, Marcq-en-Baroeul, France.
Copyright 2014 ACM 978-1-4503-2577-6/14/06 $15.00.
http://dx.doi.org/10.1145/2602458.2602465 .

1. INTRODUCTION

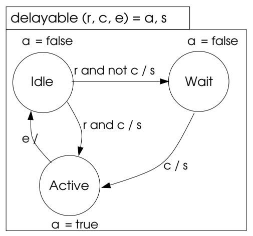
1.1 Context

Complex computing systems are increasingly designed to be self-adaptive, and therefore adopt the autonomic computing approach for the management of their administration [18]. Computing infrastructures are equipped with Autonomic Managers (AM), where monitors or sensors gather relevant information on the state and events of the Managed Elements (ME). Execution of administration actions, offered by the system API, implements regulation of the ME's activities. In between, the loop is closed by a decision component. An AM is a component that continuously reacts to flows of input information by flows of output actions, it can therefore be considered as a reactive system [11]. Self-management issues include self-configuration, self-optimization, self-healing (fault tolerance and repair), and self-protection. Typical examples are found in data-centers, with managers for resources, dependability, and energetic efficiency, as we consider in the Ctrl-Green project[1]. Usually, the automation of such administration issues is approached by building efficient and robust AMs, such as self-sizing, self-repair [21], robust reconfiguration [4] or consolidation [16].

1.2 Coordination of managers

Real systems have multiple dimensions to be managed. They do require the co-existence of multiple autonomic managers. However their uncoordinated execution can lead to interferences that could cause performance degradation or even inconsistency [1]. This is still an open problem in autonomic computing [17]. One solution consists in re-designing a new global loop taking into account combined effects, but this is even more complex than for individual loops, and is contrary to the benefits of modularity and re-usability of the AMs. Therefore, there is a deep need for methodological supports facilitating the coordination of multiple autonomic managers. Many approaches have been proposed for coordinating managers. For instance in the presence of quantitative metrics, like energy and performance, it is possible to define composition functions [6], for example involving notions of utility. Here, we consider the case of event-based coordination focusing on qualitative aspects. The coordination strategy can be ensured by an upper-level AM. This latter AM, above the individual AMs considered as MEs themselves, constitutes a coordination controller.
Some component-based frameworks, e.g., Fractal [5], provide

[1]`http://www.en.ctrlgreen.org/`

(a)


(b)
```
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
```

(c)

twotasks(r_1, e_1, r_2, e_2)
$= a_1, s_1, a_2, s_2$

enforce not (a_1 **and** a_2)

with c_1, c_2

$(a_1, s_1) =$ delayable(r_1, c_1, e_1) ;
$(a_2, s_2) =$ delayable(r_2, c_2, e_2)

Figure 1: Heptagon/BZR example: delayable task control(a) graphical / (b) textual ; (c) exclusion contract.

a structural hierarchical framework, associating a control behavior locally to a component, where the problem of coordination can be addressed. AM components are equipped with notions of observability and controllability. However, hand-made methodologies remain complex and error-prone, and hard to re-use. The difficulty in designing coordinators is in the combinatorial complexity of cases of interferences, for which there is a need for support models and tools. Another way to look at it is to consider actions invocations and their firing conditions as events, and to enforce a control logic that prevents malfunction, based on states of the AMs. Coordinating AMs can then be seen as the problem of synchronization and logical control of administration operations which can be applied by AMs on the MEs in response to observed events. The combinatorial complexity of formal techniques at compilation time calls for methods explicitly considering scalability issues, e.g., through modularity.

1.3 Our approach and contribution

In previous work [7, 9], we have defined the notion of controllable autonomic manager components, and proposed the reactive control of their coordinated assemblies in a hierarchical, systematic structure. Our approach involves formal models and techniques originally designed for reactive embedded systems. We adopt the so-called "synchronous" languages which are specially well-fit for the specification, validation and implementation of reactive kernels [11], which makes them relevant for the problem domain of autonomic loops. Additionally, we benefit from the Discrete Control Synthesis (DCS) technique, stemming from Control Theory [20] and integrated in the synchronous languages and tools [19] : it enforces coordination logic between concurrent activities, in terms of events and states, with automated algorithms used off-line, at compilation time. However, in [9], our hierarchical proposal remained monolithic.

In this paper, our contribution is to leverage the approach with a method stressing modularity, with benefits for the design of multiple-loop managers: (i) re-use of complex managers and their control specifications without modification, in different contexts, (ii) scalability for the state-space-based control technique, by breaking down its combinatorial complexity. Another contribution is the validation of our method in the coordination of a multi-loop autonomic multi-tier system, supporting multiple applications.

The principle of our approach is to identify design constraints on AMs : observability and controllability, and to construct a component-based structure where they are explicit, in a way not involving modifying the AMs. Section 2 introduces background ; Section 3 defines the modular specification and formalization of behaviors and coordination control objectives ; Section 4 validates our approach on a class of multi-tier autonomic systems ; Section 5 discusses related work ; Section 6 concludes and draws perspectives.

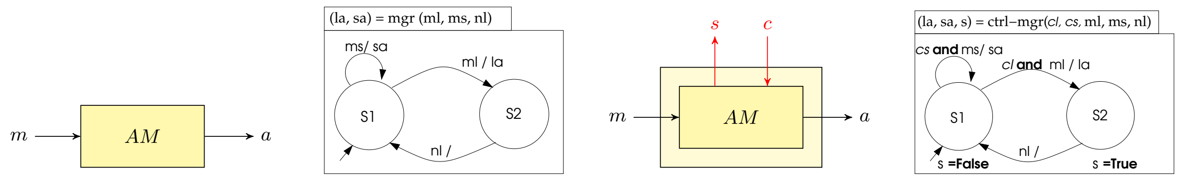
2. BACKGROUND: REACTIVE CONTROL

2.1 Reactive languages and Mode Automata

Reactive systems are characterized by their continuous interaction with their environment, reacting to flows of inputs by producing flows of outputs. They are classically modeled as transition systems or automata, with languages like StateCharts [13]. We adopt the approach of synchronous languages [11], because we then have access to the control tools used further. The synchronous paradigm refer to the automata parallel composition that we use in these languages, allowing for clear formal semantics, while supporting modelling asynchronous computations [12]: actions can be asynchronously started, and their completion is waited for, without blocking other activity continuing in parallel. The Heptagon/BZR language [8] supports programming of mixed synchronous data-flow equations and automata, called Mode 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 next state and the values of 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 1(a,b) shows a small Heptagon/BZR program. The node `delayable` programs the control of a 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 the concrete task start 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 1(c), with two instances of the `delayable` node. They run in parallel, in a synchronous way: one global step corresponds to one local step for every node.

The compiler 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. It takes incoming values of



(a) manager (b) manager model (c) controllable manager (d) controllability model

Figure 2: Modelling managers control

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 Discrete control and Heptagon/BZR

Using a reactive language gives all the support of the the classical formal framework of Labelled Transition Systems (LTS), not formally described here due to space limitations. In this work, we focus on software engineering and methodology; formal techniques are not in the scope of this paper [8]. Particularly, we benefit from state-space exploration techniques, like Model-Checking or, more originally, Discrete Controller Synthesis (DCS). Initially defined in the framework of language theory [20], DCS has been adapted to symbolic LTS and implemented in tools within the synchronous technology [19]. It is applied on an FSM representing possible behaviors of a system, its variables being partitioned into controllable ones and uncontrollable ones. For a given control objective (e.g., staying invariantly inside a given subset of states, considered "good"), the DCS algorithm automatically computes, by exploration of the state space, the constraint on controllable variables, depending on the 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. If no solution is found, because the problem is over constrained, then DCS plays the role of a verification.

The Heptagon/BZR language[2] includes a behavioral contract syntax [8]. It allows for the declaration, using the **`with`** statement, of *controllable variables*, the value of which being not defined by the programmer. These free variables can be used in the program to describe choices between several transitions. They are defined, in the final executable program, by the controller computed off-line by DCS, according to the expression given in the **`enforce`** statement. Knowledge about the environment such as, for instance event occurrence order can be declared in an **`assume`** statement. This is taken into account during the computation of the controller with DCS. Heptagon/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 1(c) shows an example of contract coordinating two instances of the `delayable` node of Figure 1(a). 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`** $(a_1$ **`and`** $a_2)$. Thus, c_1 and c_2 will be used by the synthesized controller to delay 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 Heptagon/BZR compiler generates deterministic executable code by favoring, for each controllable, value **`true`** over **`false`**, in the order of declaration.

[2]`http://bzr.inria.fr`

3. MODULAR COORDINATION

In this section, we first introduce the basic elements of modeling for coordination by discrete control in Section 3.1: the notions explored in previous work [9] are redefined in a new way, so that given the need for modularity detailed in Section 3.2, it allows for their re-use to build up the method for modular coordination in Section 3.3.

3.1 Basic AMs coordination

3.1.1 Behavior of managers

We model an autonomic manager as a reactive data-flow component. As shown in Figure 2(a), it receives a flow m of monitor inputs that it analyses in a decision process based on a representation of the managed system status. It appropriately emits a flow a of actions according to a management policy or strategy AM. Figure 2(b) shows a simple example of a manager behavior's model. It has two execution states represented by $S1$ which is the initial state, and $S2$. In $S1$ when it receives the input ms, it emits sa and stays in $S1$. We distinguish between such simple, short actions (instantaneous in the particular sense that they are completed within the execution of a step of the automaton) and long actions (asynchronous), as can be done classically with synchronous models [3]. Thus, when the automaton receives ml, it emits la and goes to $S2$ representing the processing of the action la. It returns back to $S1$ at the reception of nl notifying the completion of the asynchronous execution of la. In general, FSMs distinguish states useful for the coordination, as illustrated by concrete cases further in Section 4.

3.1.2 Controllability of managers

The controllability of the managers is considered here only at large-grain and consists in allowing or inhibiting the trigger of the management processes inherent to their management decisions. In the models, the control is represented by control variables, however its real implementation can be done in several different ways: for example the manager can be suspended and re-activated, or it can have an event interception mechanism. As shown in Figure 2(c), we exhibit the controllability of the manager by adding additional control variables c that allow to inhibit the actions a the manager can trigger. Without loss of generality, we consider one Boolean input for each action which we want to be controllable. If some actions are not controllable due to their nature (e.g., urgent recovery), or if the coordination problem does not require all of them to be controllable, then only the relevant ones can be associated with such a control condition. In general, additional outputs are also needed to exhibit an internal state s of AM, necessary for the outside controller to compute c e.g., in the case of a long action, informing that a notification nl has not arrived yet. Figure 2(d) shows how we integrate control in the previous model. We add in **ctrl-mgr** Boolean inputs cl, cs for each corresponding action, to condition transitions firing, in conjunction with the monitoring, hence giving the possibility of inhibiting actions. Output s exhibits the current state of the manager, typically the fact that a long action is executed. Note that the long action, once started, cannot be prevented or interrupted here.

3.1.3 Coordination of managers by control

The coordination of several managers is defined by the composition of the models exhibiting the controllability of their behaviors to which we associate a behavioral contract specifying the coordination policy. Figure 3(a) shows a composite node **coord-mgrs**, its body corresponds to the parallel composition of the control models **ctrl-m**$_i$ of the managers to coordinate. The associated contract for their coordination consists of three statements. The **`with`** statement is where their control variables c_i are declared to be local to **coord-mgrs**, and to be controllable in terms of DCS, as introduced in Section 2.2. The **`enforce`** statement gives the contract in the form of a Boolean expression Obj on variables from the nodes inputs or internal state s_i. The **`assume`** statement is where knowledge about the environment is defined. For simplicity we do not use it.

For example a coordination objective between two manager components, AM_1 and AM_2, can be to prevent AM_2 from triggering an action a_2 when AM_1 is in some state given by s_1. This is encoded by the following expression, to be made invariant by control: **`not`** $(s_1$ **`and`** $a_2)$. The generated controller enforcing the coordination policy, as in Figure 3(b), is in charge of producing appropriate values for the c_i control inputs to the managers. The coordination logic acts as an additional component. It enforces a policy defined in the contract for managing the interactions between the AM_i based on their inputs m_i, n_i and state s_i. At this level, the DCS problem formally encoding the coordination problem can be solved using monolithic DCS. In case of a hierarchical structure, the main Heptagon/BZR node is constructed with the contract enforcing the conjunction of all the local objectives, declaring the union of all local controllable variables, and with a body composing all manager control automata in parallel. Hence the control is centralized since only one controller is in charge of enforcing the overall objectives, if the synthesis succeeds.

3.2 The need and means for modularity

3.2.1 Limitations and need for modularity

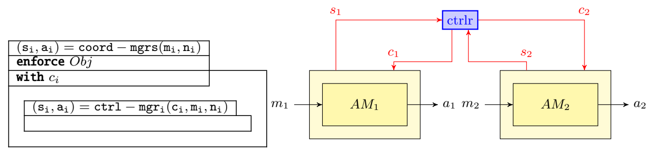
Advantages of our DCS-based approach are manifold: (i) high-level language support for controller design (tedious and error-prone to code manually at lower level) ; (ii) automated formal synthesis of controllers, correct by design (hard to guarantee manually) ; (iii) maximal permissiveness of controllers : they are minimally constraining, and in that sense optimal (even more difficult to obtain manually). However, until now the approach had not been leveraged to hierarchical modularity, and remained monolithic.

This produces a unique controller enforcing the overall control objectives. However, when considering a large number of managers, this monolithic approach might not succeed, because exploring the large state space would be very time consuming. This can take several days and can fail due to computing resource limits. This limits the scalability of the approach. Furthermore, a modification, even partial, leads to a recompilation of the overall coordinated composition invalidating previous generated codes which limits the re-usability of management components.

To address this issue, we want to exploit modular DCS, where the control objectives can be decomposed in several parts, each part managed by a controller. Each controller manages a limited number of components. This decreases the state space to explore for the synthesis of each controller. The recompilation of a controller that has no impact on other controllers does not require the recompilation of the latter. This makes possible the re-use of controllers generated codes. Not only autonomic managers are available for re-use but coordinated assemblies of managers can also be made available for further re-use. In the following Sections we detail how modular DCS is used to obtain this scalability and re-usability of management components.

3.2.2 Modular contracts in Heptagon/BZR

Modular DCS consists in taking advantage of the modular structure of the system to control locally some subparts of this system [19]. The benefits of this technique is firstly, to allow computing the controller only once for specific components, independently of the context where this component is used, hence being able to reuse the computed controller in other contexts. Secondly, as DCS itself is performed on a subpart of the system, the model from which the controller is synthesized can be much smaller than the global model of the system. Therefore, as DCS is of practical exponential complexity, the gain in synthesis time can be high and it can be applied on larger and more complex systems.


$(s_i, a_i) = \text{coord} - \text{mgrs}(m_i, n_i)$
enforce Obj
with c_i
$(s_i, a_i) = \text{ctrl} - \text{mgr}_i(c_i, m_i, n_i)$
s_1
ctrlr
c_2
c_1
s_2
m_1
AM_1
a_1
m_2
AM_2
a_2

(a) coordination model (b) coordinated managers

Figure 3: Single-level coordination of managers

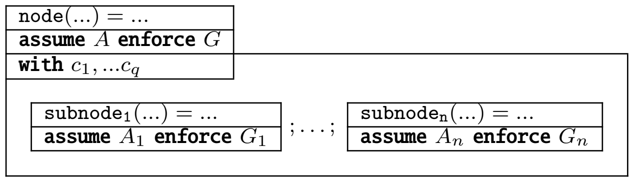
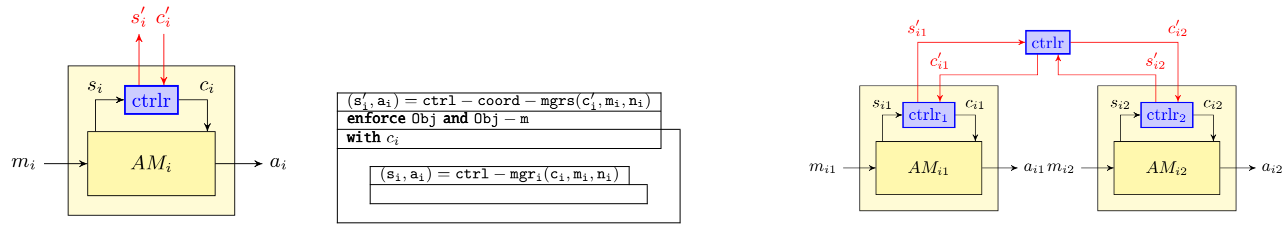


Figure 4: Modular contracts in Heptagon/BZR.



s'_i c'_i
s_i c_i
ctrlr
AM_i
m_i
a_i
$(s'_i, a_i) = \mathrm{ctrl-coord-mgrs}(c'_i, m_i, n_i)$
enforce Obj and Obj − m
with c_i
$(s_i, a_i) = \mathrm{ctrl-mgr}_i(c_i, m_i, n_i)$
s'_{i1} c'_{i2}
ctrlr
c'_{i1} s'_{i2}
s_{i1} c_{i1} s_{i2} c_{i2}
ctrlr_1 ctrlr_2
m_{i1} AM_{i1} a_{i1} m_{i2} AM_{i2} a_{i2}

(a) controllable composite (b) controllable composite model (c) Hierarchical modular control

Figure 5: Modular coordination of managers

Heptagon/BZR benefits from the modular compilation of the nodes: each node is compiled towards one sequential function, regardless of its calling context, the inside called nodes being abstracted. Thus, modular DCS is performed by using the contracts as abstraction of the sub-nodes. One controller is synthesized for each node supplied with local controllable variables. The contracts of the sub-nodes are used as environment model, as abstraction of the contents of these nodes, to synthesize the local controller. As shown in Figure 4, the objective is to control the body and coordinate sub-nodes, using controllable variables $c_1, ..., c_q$, given as inputs to the sub-nodes, so that G is true, assuming that A is true. Here, we have information on sub-nodes, so that we can assume not only A, but also that the n sub-nodes each do enforce their contract : $\bigwedge_{i=1}^{n}(A_i \implies G_i)$. Accordingly, the problem becomes that: assuming the above, we want to enforce G as well as $\bigwedge_{i=1}^{n} A_i$. Control at composite level takes care of enforcing assumptions of the sub-nodes. This synthesis considers the outputs of local abstracted nodes as uncontrollable variables, constrained by the nodes' contracts. A formal description, out of our scope here, is available [8].

3.3 Modular coordination principle

With modularity, we can decompose the coordination policy in several parts structured in a hierarchical way. This involves to make coordinated assemblies themselves controllable. In contrast to the monolithic DCS, the modular DCS allows to construct local controllers so that they can be reused in an assembly composite to form a global control. These local controllers can also be the composition of sub-controllers themselves. The control is decentralized in the sense that each part of the assembly handles part of the control. The first step to achieve a modular control is to make a coordinated assembly composite controllable. This can be seen as making the latter expose their controllability (like AMs before) in order to allow to enforce further additional coordination policy for a global management.

3.3.1 *Controllable coordinated managers*

In order for a controller to be controllable, it must enforce local objectives defining the local control strategy, as well as outside objectives. The enforcement of the outside objectives is required to allow the re-use of the controller in different contexts in which additional control strategies have to be enforced beside the predefined local one. Hence the outside objectives describe the guarantee of a control strategy received from elsewhere. This must be explicitly part of the contract of the controller. Starting from a coordinated composite as before, making the latter controllable is achieved by first equipping it with controllable Boolean inputs c'_i for each of the actions to be controlled. The second step is to install a way for the node to exhibit information about its state to outside. It can be done by outputting state information s'_i directly as suggested informally in Figure 5(a). Alternately, in order to formalize things in a way enabling modular DCS, we transform the **enforce** part of the contract, so that it can be used in an upper-level contract as environment model, as explained above and in Section 3.2.2. We modify the objective into the conjunction of the previously seen local objective Obj, **and** a term Obj_m, formalizing the fact that when the new control variable c'_i is **false**, it does inhibit its associated action a_i, i.e., it implies that it is **false**. For each action, its associated outside control objective for its inhibition is formulated as follows: $(\neg c'_i \Rightarrow \neg a_i)$. However depending on the type (short or long) of the action the objective is translated differently. For short action it is translated directly to: $Obj_m = (c'_i$ **or not** $a_i)$.

Long actions must be handled differently, because once a_i is triggered, c'_i can no longer prevent or interrupt it. Therefore, in order to make this explicit in the local contract, to be used by upper-level contracts, we must link the values of a_i (triggering of the action) and s_i (current state of the action). This is done by saying that, if the action was not active at the previous instant, i.e., that s_i was false (**not** (**false fby** s_i) [3], and a_i is not true at the current instant, then s_i will remain false. As before, c'_i can prevent the triggering of the action, i.e., that a_i becomes true. Hence, $Obj_m = \mathrm{LongActions}(c'_i, a_i, s_i)$ defined by:

$$\mathrm{LongActions}(c'_i, a_i, s_i) \overset{\mathrm{def}}{=} (c'_i \textbf{ or not } a_i) \textbf{ and } \Big((\textbf{not } (\textbf{false fby } s_i) \textbf{ and not } a_i) \Rightarrow \textbf{not } s_i\Big)$$

As illustrated in Figure 5(b), in the node **ctrl-coord-mgrs** a DCS problem will be solved, by taking as control objective to be made invariant: $Obj \wedge Obj_m$, where Obj_m of this level of contract is defined as previously explained. The sub-nodes M_i each exhibit their contract Obj_i, which includes the local modularity term as above. Hence, the DCS problem at this level will make the assumption that $\bigwedge_i Obj_i$ is enforced by lower-level contracts and coordination controllers, as explained in Section 3.2.2.

3.3.2 *Modular coordination of managers*

As composites have been made controllable in the same way as managers, they can be used to construct coordinated assemblies modularly. Re-use of instances of composites is made seamless in new assemblies. For example, the previous

[3] **fby** is an Heptagon operator introducing an initialized delay: v **fby** x denotes the previous value of x, initialized with v at the first instant.

c'_i variables will be used as controllable variables w.r.t. a policy at the upper-level composite (in Figure 5(c)), possibly combined with a controllable manager with its corresponding c_i variables.
Note that transformations above, adding modularity terms, concern only the interface and contract, i.e., the signature of the node, and not at all the internals of the managers. This is ensuring modularity in that nodes can be re-used, and combined in different contexts without modification.
The other, even more important impact of modularity is to break down the algorithmic complexity of DCS, from exponential on the global system, i.e., the parallel composition of all automata, down to a sum of DCS problems, local to much smaller models. This is enabling the scalability of the approach: comparative evaluations on the case-study are given below in section 4.3.2.

4. MULTI-LOOP MULTI-TIER SYSTEMS

We apply and validate our approach to multi-loop multi-tier systems, typical of the domain of data centers administration. This work is done in the framework of the Ctrl-Green project, in cooperation with Eolas, who make a business in providing Cloud services.

4.1 Datacenter management

4.1.1 Multi-tier replication based system

The JEE multi-tier applications we consider, as shown in Figure 6, consists of: an apache web server[4] receiving incoming requests, and distributing them with load balancing to a tier of replicated tomcat servers[5]. The latter access to the database through a mysql-proxy server[6] which distributes the sql queries, with load balancing, to a tier of replicated mysql servers[7]. The global system running in the data-center consists of a set of such applications in parallel.

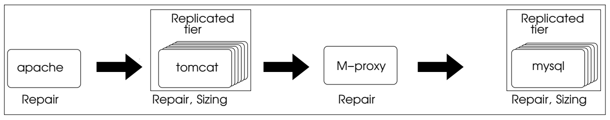


Figure 6: Multi-loop JEE Multi-tiers application.

4.1.2 Autonomic managers

For each AM, we describe its target, aim, input sensors, output actions (short or long), and controllability.

Self-sizing targets replicated servers. It aims at lowering the resources usage while preserving the performance. It automates the process of adapting the degree of replication depending of the system load measured through the CPU usage of the servers hosts. The desired state is delimited by thresholds, `minimum` and `maximum`. Periodically, an exponentially weighted moving average (EWMA), `cpu_Avg`, is computed. When `cpu_Avg` is higher than the `maximum` threshold (i.e., `overload`), it triggers size-up (a long action) for the provision of a new replica. When `cpu_Avg` is lower than the `minimum` threshold (i.e., `underload`), it triggers size-down (short action) for the removal of a replica. Each of these two actions can be inhibited.

Self-repair targets a server as well as replicated servers. It aims at preserving the availability of the server service. It manages `fail-stop` failure detected through `heartbeat`, and automates the process of restoring a server when it fails. It triggers the repair (long action, can be inhibited) of a failed server which consists in deploying the server on a new host, configuring it, and launching it. For replicated servers, the degree of redundancy is restored to tolerate up to *m-1* failures of m servers during the mean time to repair.

Consolidation targets the global virtualized data-center. It aims at optimizing global resource usage while preserving system performance. It automates the process of adapting the computing capacity made available in a virtualized data-center. It periodically evaluates the resources allocated to the virtual machines (`VM`) and the available computing capacity, and plans long actions to either reduce (`Decr`) or increase (`Incr`) the capacity. In this work, we use `VMWare DPM` for power management in a virtualized data-center. It plans migration actions to deliver more resources to the overloaded `VM`s, which can require to turn physical servers on. When the physical servers are under-utilized, it plans migration actions to turn some servers off. It can be controlled by delaying or cancelling the actions. Controllability of the consolidation manager is considered here only at large-grain: an interesting perspective is finer-grain control, between the sequential phases of this complex operation, but it requires difficult determination of appropriate synchronization points.

4.1.3 Coordination problems

As seen in Figure 6, within a multi-tier application, the failure of a server in a replicated tier can cause a saturation (hence temporary overload) of the remaining servers due to the fail-over mechanism. Furthermore, each tier depends on its predecessor (e.g., load balancer) since its service is requested by the latter. An increase of the requests received from its predecessor increases its activity and reciprocally. However the decrease of the requests can be caused by a failure, which can cause a temporary underload, and useless sizing operations. At the global level of the data-center, the uncoordinated execution of instances of self-sizing and self-repair at the same time as consolidation can lead to failures of actions triggered by the managers. The execution of a consolidation plan can take a long time to complete and its success as well as its efficiency depends on the consistency of the state of the data-center along the process. The adding, repair and removal actions, occurring at any time, can invalidate a consolidation plan being executed, which did not anticipate them. This can cause failure of migration operations or inefficiency of the consolidation. Consolidation can also cause failure of adding and repair actions e.g., it can reduce the computing capacity of the VMs.

4.1.4 Coordination policy

To avoid the above interferences, policies are defined, to be enforced by inhibiting some managers accordingly.

1. Within a replicated tier, avoid size-up when repairing.

[4]http://httpd.apache.org/
[5]http://tomcat.apache.org/
[6]http://dev.mysql.com/doc/refman/5.1/en/mysql-proxy.html
[7]http://www.mysql.com/

2. Within a load-balanced replicated tier, avoid size-down when repairing the load-balancer.

3. In multi-tiers, more generally, avoid size-down in a successor replicated tier when repairing in a predecessor.

4. At global data-center level, when consolidating, avoid self-sizing or repairing.

5. Wait until repairs or add finish before consolidation decreasing, and until removals finish before increasing.

4.2 Modular control model

In this Section we formalize the previous description, by modelling the behaviors of individual managers, and their coordination policy, in the form of a DCS problem, following the method of Section 3.3.

4.2.1 *Modelling the managers control behaviors*

Self-sizing control is actually an instance of the general pattern of Figure 2(d), node `ctrl-mgr`, with outputs : long action `add`, short action `rem` and busy state `adding` ; with inputs : controls `ca` and `crm` for the actions, monitoring overload `o` and underload `u`, and adding notification `na`:
`(add, rem, adding) = self-sizing(ca, crm, o, u, na)`

Self-repair control is a simpler case, with only a long action of repairing. This can also be defined as an instance of the node `ctrl-mgr` of Figure 2(d) with outputs : long action `rep`, and busy state `repairing` ; and inputs : control `ca` , monitoring failure `fail` , and notification of repair done `nr`. Unused parameters for short actions of the `ctrl-mgr` node can be, for inputs, given the constant value `false`, and for outputs be left unused. This defines the new node:
`(rep, repairing) = self-repair(cr, fail, nr)`

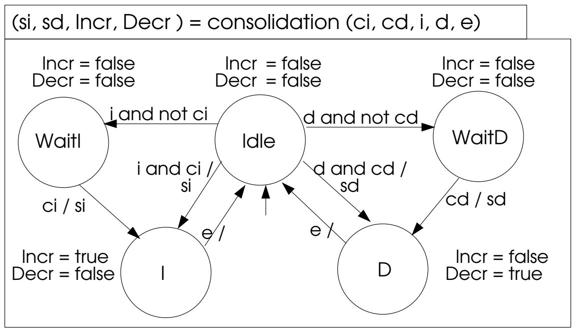


Figure 7: Consolidation control behavior model.

Consolidation control is an example showing that different control patterns can be modelled in our approach, according to specific managers behaviors, and also depending on the relevant states to be exhibited for different control purposes. In Figure 7, its automaton presents essentially the waiting mechanism of the delayable action, as in Figure 1(a), for each of its two long actions, the activity of which is given by `Incr` and `Decr`. In the initial Idle state, when i is **`true`** (increase of the computing capacity is required), if ci is **`true`** it goes to I and emits si to start the increasing plan, otherwise it goes to WaitI. There, it awaits ci to be **`true`** to go to Incr and emits si. When in Incr, it awaits until the completion of the execution (e is **`true`**) then returns back to the Idle state. The case for decrease is similar.

4.2.2 *Coordination objectives*

The parallel composition of instantiations of the above automata describes the coexistence of the instances of the corresponding managers. The control is specified on this composed behavior. We formalize the strategy of Section 4.1.4.

1. Within each replicated tier, avoid size-up when repairing: **not** (repairing **and** add)

2. Avoid size-down when repairing the load-balancer: **not** (repairingL **and** rem)

3. In multi-tiers, more generally, between predecessors and successors: **not** (repairing$_{pred}$ **and** rem$_{succ}$)

4. When consolidating, avoid repair and sizing: **not** ((Incr **or** Decr) **and** (repairing* **or** adding* **or** rem*))

5. Wait for consolidation decreasing until repairs or add finish: **not** ((repairing* **or** or adding*) **and** sd) and for increasing when removals: **not** (rem* **and** si)

4.3 Exploiting the models with DCS

4.3.1 *Monolithic synthesis*

In order to evaluate the benefit of modularity, we make the exercise of performing DCS the classical way.

```
(...) = Main_node (...)
enforce all contracts
with all controllable variables
  (rep_1, repairing_1)   = self-repair (c'_1, fail_1, nr_1);
  ...
  (rep_N, repairing_N)   = self-repair (c'_N, fail_N, nr_N);
  (add_1, rem_1, adding_1) = self-sizing (ca_1, ...);
  ...
  (add_M, rem_M, adding_M) = self-sizing (ca_M, ...);
  (si, sd, Incr, Decr) = consolidation (ci, cd, i, d,e);
```

Figure 8: Monolithic node.

The specification of the monolithic control is encoded in a single composite node, shown in Figure 8, grouping all instances of involved managers composed in parallel in its body, and a conjunction of all control objectives in its contracts. This can be tedious and complex when a huge number of managers are considered. It does not allow a decentralized control because the overall control objectives are grouped in the single upper-level node. The structure of this coordination is shown in Figure 14.

4.3.2 *Modular synthesis*

We present reusable nodes bottom-up, as shown in Figure 9 from left to right: we first build the coordination controller for `self-sizing` and `self-repair` in a replicated tier. The latter controller is re-used for the coordination of managers in two consecutive tiers, the front tier being a load balancer for the second tier constituted of replicated servers. The resulting controller is re-used for the coordination of a mult-tier system.

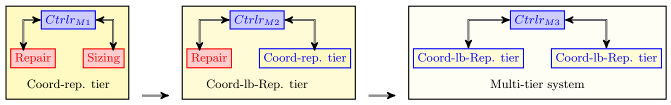


Figure 9: Bottom-up re-use of nodes.

Replicated servers tier. The composite node shown in Figure 10, specifies the control of instances of `self-sizing` and `self-repair` managing the same replicated tier. Its contract is composed of four objectives, one for the local coordination: (**not** (`repairing` **and** `add`)), while the rest concerns the guarantee of the enforcement of a coordination strategy from outside the node. The control from outside is received through the input variables `cr'`, `ca'` and `crm'`. As can be seen here, the modularity objective is very systematic and could easily be covered in syntactic sugar.

```
(...) = coord_repl-tier (cr', fail, nr, ca', crm', o, u, na)
enforce (not (repairing and add))
        and LongActions(cr', rep, repairing)
        and LongActions(ca', add, adding)
        and (crm' or not rem)
with cr, ca, crm
    (rep, repairing)   = self-repair (cr, fail, nr);
    (add, rem, adding) = self-sizing (ca, crm, o, u, na);
```

Figure 10: Replicated tier node.

Load balancer and replicated servers tier. In this node, shown in Figure 11, we re-use an instance of the above composite node from Figure 10, and an instance of self-repair, dedicated to the management of a load balancer in front of the replicated servers which distributes the incoming load to them. Here also, the local coordination strategy to be enforced : **not** (`repairingL` **and** `remove`), is complemented with modularity objectives.

```
(...) = coord_lb-repl-tier (cL', failL, nrL,
                            c', fail, nr,
                            ca', crm', o, u, na)
enforce (not (repairingL and rem))
        and LongActions(cL', repL, repairingL)
        and LongActions(c', rep, repairing)
        and LongActions(ca', add, adding)
        and (crm' or not rem)
with cL, c, ca, crm

 (repL, repairingL)   = self-repair (cL, failL, nrL);
 (rep, repairing, add, rem, adding)
    = coord_repl-tier (c, fail, nr, ca, crm, o, u, na);
```

Figure 11: Load-balanced replicated tiers node.

Application. The node of Figure 12 coordinates two instances of the previous node from Figure 11, for the control of instances of self-sizing and self-repair managing two consecutive load-balanced replicated tiers. The coordination strategy consists in preventing size-down in the back-end load-balanced replicated tier ("successor") when a failure is being repaired in the front. This is expressed as follows: (**not** ((`repairingL`$_1$ **or** `repairing`$_1$) **and** `rem`$_2$))

```
(...) = coord_appli (cL'_1, failL_1, nrL_1,
                     c'_1, fail_1, nr_1, ca'_1, crm'_1, o_1, u_1, na_1)
                     cL'_2, failL_2, nrL_2,
                     c'_2, fail_2, nr_2, ca'_2, crm'_2, o_2, u_2, na_2)
enforce (not ((repairingL_1 or repairing_1) and rem_2))
        and LongActions(cL'_i, repL_i, repairingL_i)
        and LongActions(c'_i, rep_i, repairing_i)
        and LongActions(ca'_i, add_i, adding_i)
        and (crm'_i or not rem_i)      i = 1, 2
with cL_1, c_1, ca_1, crm_1, cL_2, c_2, ca_2, crm_2

 (repL_1, repairingL_1, rep_1, repairing_1, add_1, rem_1, adding_1)
    = coord_lb-repl-tier (cL_1, failL_1, nrL_1, c_1, fail_1, nr_1,
                                      ca_1, crm_1, o_1, u_1, na_1);
 (repL_2, repairingL_2, rep_2, repairing_2, add_2, rem_2, adding_2)
    = coord_lb-repl-tier (cL_2, failL_2, nrL_2, c_2, fail_2, nr_2,
                                      ca_2, crm_2, o_2, u_2, na_2);
```

Figure 12: Multi-tier application node.

Global system: data center. The whole multi-application system will be constructed progressively, by first considering the two-application case. Figure 13 shows the node and contract instantiating the previous node for each of them,

```
(...) = two-data-center (...)
enforce (not ((Incr or Decr) and
                 (repairing_ij or adding_ij or rem_ij))
        and (not ((repairing_ij or adding_ij) and sd)
                 and not (rem_ij and si))
        and LongActions(cL'_ij, repL_ij, repairingL_ij)
        and LongActions(c'_ij, rep_ij, repairing_ij)
        and LongActions(ca'_ij, add_ij, adding_ij)
        and (crm'_ij or not rem_ij)      i = 1, 2; j = 1..2   )
with cL_11, c_11, ..., crm_22, ci, cd

 (...) = coord_appli (cL_11, c_11, ca_11, crm_11, ...,
                      cL_21, c_21, ca_21, crm_21, ...)
 (...) = coord_appli (cL_12, c_12, ca_12, crm_12, ...,
                      cL_22, c_22, ca_22, crm_22, ...)
 (si, sd, Incr, Decr) = consolidation (ci, cd, i, d, e);
```

Figure 13: Two-application data-center.

as well as a consolidation manager. At this level of control, only the coordination strategy between the multi-tiers applications and the consolidation manager is specified, the control within multi-tier applications being delegated to the instance of the previous node modelling it. Having more applications in a data-center is done by composing an instantiation re-using the previous node, with a new instantiation re-using the application node. The contract of this new composition is similar to the one in Figure 13. This enables a hierarchical construction of the control of an N-application.

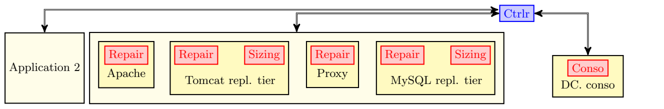


Figure 14: Monolithic coordination design

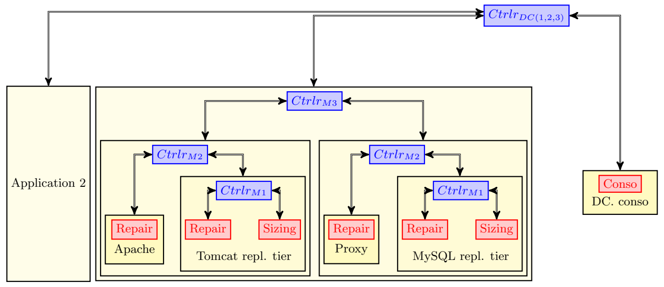


Figure 15: Modular coordination design

Comparisons and discussion.

Advantages of modularity can be seen here, in terms of the objectives of Section 1.3. Regarding the specification aspect (objective (i) of Section 1.3): tiers and groups of tiers are described locally, including their control, and assembled hierarchically, as shown in Figure 15; instead of having all automata on one side, and all contracts on the other side, in the monolithic case as shown in Figure 14. This favors the re-use of Heptagon/BZR nodes in different contexts. In particular, the repair manager is re-used in the replicated tier and for the load balancer. More significantly, because it has a contract and controller, the coordinated load-balanced and replicated tier is used twice in an application, with a difference in the controls, in that the downstream one is submitted to more constraints than the upstream one.

On the other aspect, the combinatorial complexity of DCS and the cost of compilation of the controllers (objective (ii) of Section 1.3): for various sizes of the system (i.e, various number of applications), we have performed Heptagon/BZR compilations and synthesis, the results of which are shown in Table 1. Comparative costs of DCS, monolithic and modular, for the different cases varying in number of applications in the data-center, are given in terms of compilation CPU time, and memory usage. For small numbers of applications, values are not significant for memory; at 4 applications, the monolithic approach reaches the limits of the natural combinatorial explosion of the state-space exploration techniques : the computation was not finished after more than two days, and no values were sought for larger systems. The other approach, benefiting from modularity, goes significantly further, even if still presenting growing costs. In brief, we can see that monolithic DCS is exponentially costly in the size on the system, whereas modular DCS keeps producing results, showing scalability.

nb.	Synthesis time		Memory usage	
app.	monolithic	modular	monolithic	modular
1	0s	5s	-	-
2	49s	11s	-	-
3	42m24s	24s	34.81MB	-
4	> 2 days	1m22s	>149,56MB	-
5	-	4m30s	-	20,37MB
6	-	13m24s	-	53,31MB
7	-	25m57s	-	77,50MB
8	-	50m36s	-	115,59MB
9	-	2h11m	-	236,59MB
10	-	9h4m	-	479,15MB

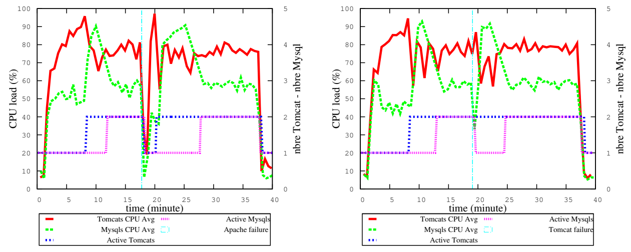
Table 1: DCS : duration and memory usage.

Although we show the total compilation time in Table 1, the synthesis of the control logic of each node equipped with a contract is performed independently. A composite node which is the assembly of sub-nodes equipped with a contract, requires just the contract defined in the sub-nodes at compilation for the synthesis of its control logic. Therefore the compilations can be run in parallel. Furthermore, the recompilation of the composite node is necessary only when their interface (inputs, outputs) and their contract are modified, otherwise it can be re-used as such.

4.4 Implementation

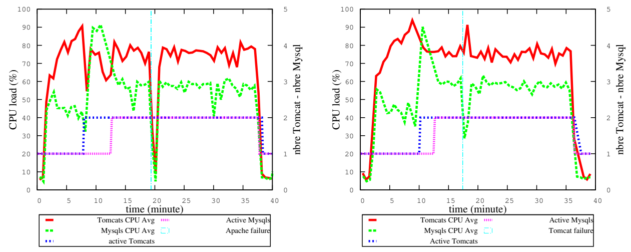
The system we described has been implemented on our experimental data-center. Figure 16 shows uncoordinated executions in which failures occur in 16(b) at 17 min (`Apache` server fails), and in 16(b) at 19 min (`Tomcat` server fails). In 16(b), the failure leads to an underload in `Tomcat` and `Mysql` tier causing the removal of a replicated server in each tier. In 16(b) the failure causes an underload in `Mysql` tier which leads to the removal of a replica., as seen in the square-edged curve (numbers of replica) going down. However, the degree of replication is restored after the repair of the failed server, by re-adding the uselessly removed server as shown in 16(a) at 21 min and 28 min, and in 16(b) at 25 min.

By contrast, in Figure 17 for executions coordinated by the controllers as expected, reaction to the underloads during the failure repair (in 17(a): 20min, and in 17(b): 17min) is inhibited, square-edged curves remaining flat, hence the system administration saves unnecessary operations.



(a) app 1: `Apache` failure (b) app 2: `Tomcat` failure

Figure 16: Uncoordinated execution



(a) app 1: `Apache` failure (b) app 2: `Tomcat` failure

Figure 17: Coordinated execution

5. RELATED WORK

The general question of coordinating autonomic managers remains an important challenge in Autonomic Computing [17] 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 [23], which can be e.g., through the access to a common knowledge [2]. A distinctive aspect of our approach is to rely on explicit automata-based behavioral models, amenable to formal techniques like verification or the more constructive DCS. Coordination of multiple energy management loops is done in various ways, e.g., by defining power vs. performance trade-offs based on a multi-criteria utility function in a non-virtualized environment [6], or also tuning mechanisms as in OptiTuner [15]. 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 controlling their logical activity state, rather than modifying them.

Concerning decision and control of autonomic systems, some approaches rely upon Artificial Intelligence and planning [22] 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 Discrete Event Systems, applied to computing systems [14]. 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. 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 not been many until now; it has been applied to address the problem of deadlock avoidance [24]. Compared to this, we consider more user-defined objectives.

Works on compositional verification have brought some issues which can be related to modular controller synthesis. As instance, a method for automatic assumption generation have been proposed [10]. It relies on algorithms for the generation of automata based on language equivalence, in order to generate intermediary assumptions for compositional verification. Compared with modular controller synthesis, the generated automata do not act upon the system, and only helps its verification. Nevertheless, an interesting perspective would be to consider mixing the two techniques, in order to facilitate the controller synthesis, and relieve the programmer from the burden of writing intermediary assumptions. Though, this technique cannot be applied as is, as assumptions cannot be inferred from properties to be enforced, without knowledge about the generated controller.

6. CONCLUSIONS

We put the principle of modularity in practice for the problem of coordination in multiple-loop autonomic management, in a component-based approach. Instead of re-designing a global combined loop, we benefit from the advantages of modularity, by defining a new method. We propose a general design methodology based on formal modelling with automata, and the application of DCS to obtain automatically correct controllers. We leverage modularity in this approach, and confront it to commensurate experiment on a real-world multi-tier, multi-service-level system. We achieve our objectives of Section 1.3 by:

1. enabling re-use and coordination of complex administration managers, through their control specifications,
2. modularizing the DCS, thereby breaking down the exponential complexity of the basic algorithms

On the latter point, the gain in compilation-time synthesis opens new perspectives on the scalability of our method, and its applicability to larger systems.

Perspectives are at different levels. The general method is systematic enough to form the basis of an administration management-level Domain Specific Language (DSL), allowing for a designer to construct systems for which the formal automata models and control objectives can be generated automatically. Improvement of the DCS technique is ongoing to integrate not only logical but also quantitative aspects in the synthesis algorithms, like consumption or load. Also the compilation using modular DCS produces a modular code which opens perspectives for a distributed execution which is an ongoing work.

7. REFERENCES

[1] T. Abdelzaher. Research challenges in feedback computing: An interdisciplinary agenda. In *Proc. Workshop Feedback Computing*, 2013.

[2] F. Alvares de Oliveira Jr., R. Sharrock, and T. Ledoux. Synchronization of multiple autonomic control loops: Application to cloud computing. In *Proc. Conf. Coordination*, 2012.

[3] T. Bouhadiba, Q. Sabah, G. Delaval, and E. Rutten. Synchronous control of reconfiguration in Fractal component-based systems – a case study. In *Proc. Conf. EMSOFT*, 2011.

[4] F. Boyer, O. Gruber, and D. Pous. Robust reconfigurations of component assemblies. In *Proc. Conf. ICSE*, 2013.

[5] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.-B. Stefani. The Fractal component model and its support in java. *Software – Practice and Experience (SP&E)*, 36(11-12), sep 2006.

[6] 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. Conf. AAMAS*, 2008.

[7] G. Delaval and E. Rutten. Reactive model-based control of reconfiguration in the Fractal component-based model. In *Proc. Symp. CBSE*, 2010.

[8] G. Delaval, E. Rutten, and H. Marchand. Integrating discrete controller synthesis into a reactive programming language compiler. *Discrete Event Dynamic Systems*, 23(4):385–418, Dec. 2013.

[9] S. M.-K. Gueye, N. de Palma, and E. Rutten. Coordination control of component-based autonomic administration loops. In *Proc. Conf. Coordination*, 2013.

[10] A. Gupta, K. McMillan, and Z. Fu. Automated assumption generation for compositional verification. In W. Damm and H. Hermanns, editors, *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 420–432. Springer Berlin Heidelberg, 2007.

[11] N. Halbwachs. Synchronous programming of reactive systems, a tutorial and commented bibliography. In *Proc. Conf. CAV*, 1998.

[12] N. Halbwachs and S. Baghdadi. Synchronous modeling of asynchronous systems. In *Proc. Conf. EMSOFT*, Grenoble, Oct. 2002.

[13] D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, Oct. 1996.

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

[15] 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.

[16] F. Hermenier, X. Lorca, J.-M. Menaud, G. Muller, and J. Lawall. Entropy: A consolidation manager for clusters. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pages 41–50, New York, NY, USA, 2009. ACM.

[17] J. Kephart. Autonomic computing: The first decade. In *Proc. Conf. ICAC*, 2011.

[18] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1), 2003.

[19] H. Marchand and M. Samaan. Incremental design of a power transformer station controller using a controller synthesis methodology. *IEEE Trans. on Soft. Eng.*, 26(8):729 –741, 2000.

[20] P. Ramadge and W. Wonham. On the supervisory control of discrete event systems. *Proc. IEEE*, 77(1), Jan. 1989.

[21] S. Sicard, F. Boyer, and N. De Palma. Using components for architecture-based management: the self-repair case. In *Proc. Conf. ICSE*, 2008.

[22] D. Sykes, W. Heaven, J. Magee, and J. Kramer. Plan-directed architectural change for autonomous systems. In *Proc. Workshop SAVCBS*, 2007.

[23] P. Vromant, D. Weyns, S. Malek, and J. Andersson. On interacting control loops in self-adaptive systems. In *Proc. Conf. SEAMS*, 2011.

[24] Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, and S. Mahlke. The theory of deadlock avoidance via discrete control. In *Proc. ACM Conf. POPL*, 2009.