Discrete Control of Computing Systems Administration: a Programming Language supported Approach

Gwenaël Delaval, Noel De Palma, Soguy Mak-Karé Gueye, Hervé Marchand, Éric Rutten

Discrete Control of Computing Systems Administration: a Programming Language supported Approach

Gwenaël Delaval, Noël De Palma, Soguy Mak-karé Gueye, Hervé Marchand and Eric Rutten

Abstract— **We address the problem of using Discrete Controller Synthesis for the administration of Computing Systems, following an approach supported by a programming language. We present a mixed imperative/declarative programming language, where declarative contracts are enforced upon imperatively described behaviors. Its compilation is based on the notion of supervisory control of discrete event systems. It targets the programming of closed-loop reconfiguration controllers in computing systems. We apply our method to the problem of coordinating several administration loops in a data center (number of servers, repair, and local processor frequencies) : we formulate it as an invariance controller synthesis problem.**

I. INTRODUCTION

A. Supervisory Control of Computing Systems

Discrete Controller Synthesis (DCS) is a branch of control theory which aim is to ensure by construction some required qualitative properties on the dynamic behavior of a transition system, by coupling it in a closed-loop to a controller that determines the set of actions which may be taken without compromising the properties [3], [20]. Discrete Control Theory is quite developed theoretically, but not often applied yet. We use the tool Sigali [18], which is connected to reactive synchronous languages to automatically compute controllers enforcing properties like invariance or reachability.

The application of control theory to computing systems represents a new emerging application domain, where there is a very important potential. Indeed, computing systems are more and more reconfigurable, able to adapt themselves to changing conditions in their environment or the management of their execution platform. A recent trend, called Autonomic Computing [13], defines a notion of closed loop for the control of such reconfigurations. Concerned systems range from small embedded hardware to large-scale data centers.

However, as was noted by other authors [15], while classical control theory has been readily applied to computing systems [10], applying Discrete Control Theory to computing systems is more recent and less developed. There exist particular works focussing on controlling multi-thread code [15], [1], [8] or workflow scheduling [21], or on the use of Petri nets [12], [11], [16] or finite state automata [19]. There is no general methodology yet, to our knowledge.

B. Programming language support

Our motivation w.r.t. DCS concerns the integration of DCS into a compiler for a reactive programming language. We want to improve usability of DCS by programmers, not experts in discrete control, so as to more easily solve the automatic control of computing systems at runtime. DCS techniques have rarely been integrated with computer programming languages. Yet, our approach can be related with [15], in which DCS on Petri nets can be used to automatically derive controllers avoiding dead-lock configurations in a multi-thread program. We want to allow for the specification of more varied control objectives and behaviors.

The BZR programming language results from this motivation. BZR is a synchronous language provided with a contract construct. The properties described in a BZR contract are enforced at compilation time by means of a DCS phase. Thus, BZR can be viewed as a mixed imperative/declarative programming language, where entire logical parts of programs are automatically synthesized from the specification, instead of being manually (or rather, “brainually”) designed, occasionally with much effort.

C. Application to the coordination of administration loops

One major challenge in Autonomic Computing is the coordination of autonomic managers. There are many well-designed autonomic managers that address each a specific aspect in system administration. However, in order to manage a real system, there have to be several such managers active, to address all aspects of a complex system administration. Coordinating autonomic managers is necessary to avoid side effects inherent to their co-existence. Indeed, managers have been designed independently, without any knowledge about each other, and possibly with conflicting objectives.

Few works have investigated managers coordination. For example Kephart [6] addresses the coordination of multiple autonomic managers for power/performance tradeoffs based on a utility function. However, these works propose adhoc specific solutions that have to be implemented by hand. If new managers have to be added in the system the whole coordination manager needs to be redesigned. The design of the coordination infrastructure becomes complex for the co-existence of a large number of autonomic managers.

The reason *why* we use DCS for this coordination problem is that it involves synchronization and logical control of administration operations. These operations can be performed, suspended or delayed according to the sequences of events observed in the managed system, in order to avoid that it evolves to an undesired state. *How* we address this issue is by investigating the use of reactive models with events and states, and discrete control techniques to build a controller able to control autonomic managers at runtime.

This work was partly supported by the ANR project Ctrl-Green.

G. Delaval and N. De Palma and S. Gueye are with LIG/ UJF, Grenoble, France ; H. Marchand is with INRIA Rennes, France ; E. Rutten is with LIG/ INRIA Grenoble, France ; `{firstname.lastname}@inria.fr`

Outline. We present the BZR language, its compilation and the underlying DCS techniques in Section II. In Section III, we use it for the design of a controller that coordinates three autonomic managers (Self-Sizing, Dynamic Voltage and Frequency Scaling (DVFS) and Self-Repair) in a replicated web-server system for the management of the energy consumption and the availability of the system.

II. The BZR Programming Language

BZR (`http://bzr.inria.fr`) and its compilation involve DCS on DES models, as in Figure 1:

- the extraction of the logic DES control part from the body of the program and contract, and its compilation into an uncontrolled transition system;
- the extraction of the control objectives from contracts;
- the application of DCS upon the previous two elements;
- the composition of this controller with the uncontrolled program, producing the correct controlled automaton;
- the resulting composition is compiled towards target code, e.g., C or Java, and consists of a step function, to be called at each reaction of the reactive system.

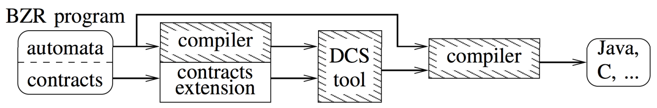


Fig. 1: Overview of the BZR design.

We briefly recall informally BZR [7], before introducing its formal semantics, in Section II-C (not published before).

A. Programming constructs

BZR is in the family of synchronous languages, very close to Lustre or Scade [2]; having our own compiler allows us to experiment for our specific research. The basic execution scheme of a BZR program is that at each reaction, a step is performed, taking current inputs as parameters, computing the transition to be taken, updating the state, triggering the appropriate actions, and emitting the output flows.

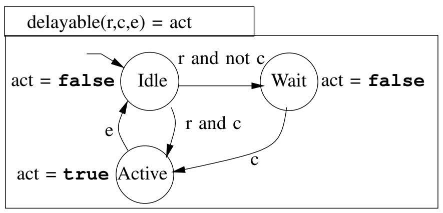


Fig. 2: Simple BZR node.

1) Data-flow nodes and mode automata: Figure 2 shows a simple example of a BZR node, for the control of a task that can be activated by a request `r`, and according to a control flow `c`, put in a waiting state; input `e` signals the end of the task. Its signature is defined first, with a name, a list of input flows (here, simple events coded as Boolean flows), and outputs (here: the Boolean act). In the body of this node we have a mode automaton : upon occurrence of inputs, each step consists of a transition according to their values; when no transition condition is satisfied, the state remains the same. In the example, `Idle` is the initial state. From there transitions can be taken towards further states, upon the condition given by the expression on inputs in the label. Here: when `r` and `c` are true then the control goes to state `Active`, until `e` becomes true, upon which it goes back to `Idle`; if `c` is false it goes towards state `Wait`, until `c` becomes true. This is a mode automaton [17]: to each state we associate equations to define the output flows. In the example, the output `act` is defined by different equation in each of the states, and is true when the task is active.

We can build hierarchical and parallel automata. In the parallel automaton, the global behaviour is defined from the local ones: a global step is performed synchronously, by having each automaton making a local step, within the same global logical instant. In the case of hierarchy, the sub-automata define the behaviour of the node as long as the upper-level automaton remains in its state.

2) Contracts in the BZR language: The new contract construct is a major add-on w.r.t. Lustre or Esterel [2]. It encapsulates DCS in the compilation of BZR [7]. Models of the possible behaviours of the managed system are specified in terms of mode automata, and adaptation policies are specified in terms of contracts, on invariance properties to be enforced. Compiling BZR yields a correct-by-construction controller, produced by DCS, in a user-friendly way: the programmer does not need to know formalisms of DCS.

$f(x_1,\dots,x_n) = (y_1,\dots,y_p)$

$(e_A, e_G) = c_f(x_1,\dots,x_n,y_1,\dots,y_p)$

assume e_A

enforce e_G

with $c_1,\dots,c_q$

$y_1 = f_1(x_1,\dots,x_n,c_1,\dots,c_q)$

$\dots$

$y_p = f_p(x_1,\dots,x_n,c_1,\dots,c_q)$

Fig. 3: BZR contract node.

Figure 3 illustrates the association of a *contract* to a node. It is itself a program c_f, with its internal state, e.g., automata, observing traces, and defining states (for example an error state where e_G is false, to be kept outside an invariant subspace). It has two outputs: e_A, *assumption* on the node environment, and e_G, to be guaranteed or *enforced* by the node. A set $C = \{c_1,\dots,c_q\}$ of local controllable variables will be used for ensuring this objective. This contract means that the node will be controlled, i.e., that values will be given to $c_1,\dots,c_q$ such that, given any input trace yielding e_A, the output trace will yield the true value for e_G. This will be obtained automatically, at compilation, using DCS. Also, one can define several such nodes with the same body, that differ in assumptions and enforcements.

We compile such a BZR contract node into a DCS problem as in Figure 4. The body and the contract are each encoded into a state machine with transition function (resp. *Trans* and *TrC*), state (resp. *State* and *StC*) and output function (resp. *Out* and *OutC*). The contract inputs *XC* come from

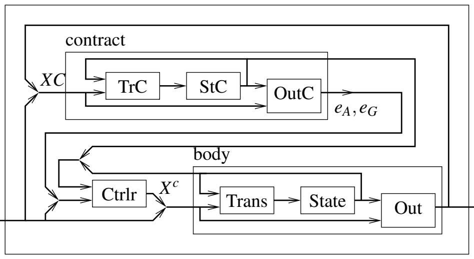


Fig. 4: BZR contract node as DCS problem

the node's input X and the body's outputs Y, and it outputs e_A, e_C. DCS computes a controller $Ctrlr$, assuming e_A, for the objective of enforcing e_G (i.e., making invariant the sub-set of states where $e_A \Rightarrow e_G$ is true), with controllable variables $c_1, \ldots c_q$. The controller then takes the states of the body and the contract, the node inputs X and the contract outputs e_A, e_G, and it computes the controllables X_c such that the resulting behaviour satisfies the objective.

The BZR compiler is implemented on top of the Heptagon compiler and the SIGALI DCS tool [18].

B. Discrete controller synthesis in the compilation

We now briefly describe the DCS theory. We introduce the symbolic transition system as the underlying model of a BZR node, how the contract can be described in this framework and how we use DCS to ensure the contract of a BZR node.

1) Transition system: We represent the logical behavior of a node by a symbolic transition system (STS), as illustrated in Figure 5, in its equational form. Synchronous compilers essentially compute this transition system from source programs, particularly handling the synchronous parallel composition of nodes. For a node f, a transition function T takes the inputs X and the current state value, and produces the next state value, memorized by S for the next step. The output function O takes the same inputs as T, and produces the outputs Y.

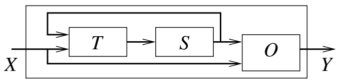


Fig. 5: Transition system for a program.

Formally, from a node f, we can automatically derive an STS given by $\mathscr{S}_f(X,S,Y)$, defining a synchronous program of state variables $S \in \mathbb{B}^m$, input variables $X \in \mathbb{B}^n$, output variables $Y \in \mathbb{B}^p$. It is a four-tuple (T, O, Q, Q_0) with two functions T and O, and two relations Q and Q_0 as in (1), where the vectors S and S' respectively encode the current and next state of the system and are called *state variables*. $T \in \mathbb{B}[S,X]$ represents the transition function.

$$\mathscr{S}_f(X,S,Y) = \begin{cases} S' = T(S,X) \\ Y = O(S,X) \\ Q(S,X) \\ Q_0(S) \end{cases} \tag{1}$$

It is a vector-valued function $[T_1, \ldots, T_n]$ from $\mathbb{B}^{n+m}$ to $\mathbb{B}^m$. Each predicate component T_i represents the evolution of the state variable S_i. $O \in \mathbb{B}[S,X]$ represents the output function. $Q_0 \in \mathbb{B}[S]$ is a relation for which solutions define the set of initial states. $Q \in \mathbb{B}[S,X]$ is the constraint between current states and events defining which transitions are admissible, i.e., the (S,X) for which T is actually defined. This constraint can be used, e.g., to encode assumptions on the inputs, i.e., assumptions on the environment. The semantics of a STS $\mathscr{S}_f$ is defined as set of sequences $\overline{(s,x,y)} = (s_i, x_i, y_i)_i$ such that : $Q_0(s_0)$ and $\forall i,\ Q(s_i,x_i) \wedge (s_{i+1} = T(s_i,x_i)) \wedge (y_i = O(s_i,x_i))$. Traces$(\mathscr{S}_f)$ denotes this set of sequences.

Operations on STS. Given two STS $\mathscr{S}_{f_1}$ and $\mathscr{S}_{f_2}$, we note by $\mathscr{S}_{f_1} \| \mathscr{S}_{f_2}$, the synchronous parallel composition of $\mathscr{S}_{f_1}$ and $\mathscr{S}_{f_2}$Formally, $\mathscr{S}_{f_1} \| \mathscr{S}_{f_2}$ is a STS $\mathscr{S}_{f_1} \| \mathscr{S}_{f_2}((X_1 \cup X_2) \setminus (Y_1 \cup Y_2), S_1 \cup S_2, Y_1 \cup Y_2)$:

$$\mathscr{S}_{f_1} \| \mathscr{S}_{f_2} = \begin{cases} S_1', S_2' = (T_1(S_1,X_1), T_2(S_2,X_2)) \\ Y_1, Y_2 = (O_1(S_1,X_1), O_2(S_2,X_2)) \\ Q_1(S_1,X_1) \wedge Q_2(S_2,X_2) \\ Q_{01}(S_1) \wedge Q_{02}(S_2) \end{cases}$$

Given a STS $\mathscr{S}_f(X,S,Y)$, we denote by $\mathscr{S}_f \triangleright A$ the extension of constraints of $\mathscr{S}_f$ with the predicate $A \in \mathbb{B}[S,X]$, namely $\mathscr{S}_f \triangleright A = (T, O, Q \wedge A, Q_o)$.

2) Contracts satisfaction: In the sequel, we shall consider properties expressed by means of contracts that are defined as follows:

Definition 1 (Contract): Given a STS $\mathscr{S}_f(X,S,Y)$, a contract is a tuple $Co = (\mathscr{S}^c, A, G)$ where $XC = X \cup Y$, $\mathscr{S}^c(XC, S^c, \emptyset)$ is a STS, $A \in \mathbb{B}[S^c]$ and $G \in \mathbb{B}[S^c]$ are predicates[1]. In the following, we will denote by $(\mathscr{S}^c_f, A_f, G_f)$ the contract associated to the node f.
Intuitively, $\mathscr{S}^c$ can be seen as a Boolean abstraction of a component program, G is the property to be satisfied by the traces of the component on which this contract is placed providing the fact that the model of the environment A is satisfied. For clarity, we define the contract predicates A and G on only state variables of the contract. We remark though that this does not restrict the expressiveness of these properties, as one can add dummy state variables constrained with inputs or outputs values, so as to be able to express properties upon inputs and outputs variables.

Definition 2 (Contract fulfilment): A STS $\mathscr{S}_f(X,S,Y)$ fulfills a contract $Co = (\mathscr{S}^c, A, G)$, noted $\mathscr{S}_f \models Co$, if $\forall \overline{(s, s^c, x, y)} \in \text{Traces}(\mathscr{S}_f \| \mathscr{S}^c)$, $\overline{(s^c)} \models \Box A \Rightarrow \overline{(s^c)} \models \Box G$ [2]. Hence, a contract is satisfied whenever the traces of $\mathscr{S}_f$, composed with $\mathscr{S}^c$ and satisfying $\Box A$, satisfy $\Box G$. As the above definition does not allow to be easily applied on STS, we give below a property on contracts: the environment model can be viewed as additional constraints of the STS composed of $\mathscr{S}$ and the contract program $\mathscr{S}^c$. We thus want to ensure that $(\mathscr{S}_f \| \mathscr{S}^c) \triangleright A \models G$ which implies that $\mathscr{S}_f \models (\mathscr{S}^c, A, G)$.

3) Contracts enforcement: Assume given a system $\mathscr{S}_f$ and a contract Co on $\mathscr{S}_f$. Our aim is to restrict the behavior of $\mathscr{S}$ in order to fulfil the contract. The control makes

[1]Predicates A and G represent the variables e_A and e_G in Figure 4.
[2]$\overline{(s^c)} \models \Box A$ stands for $\forall s^c_i \in \overline{(s^c)}, A(s^c_i)$ is satisfied

distinction between events: the *uncontrollable* event variables X^{uc} which are defined by the environment, and the *controllable* event X^c which are defined by the controller of the system ($XC = X^c \cup X^{uc}$). Now, to enforce $Co = (\mathscr{S}^c, A, G)$ with $\mathscr{S}^c(XC, S^c, \emptyset)$, $A \in \mathbb{B}[S^c]$ and $G \in \mathbb{B}[S^c]$ on $\mathscr{S}$ we consider the STS $(\mathscr{S} \| \mathscr{S}^c) \triangleright A$ and the property we wish to enforce by control is given by the invariant G. The DCS allows us to obtain predicate $K \in \mathbb{B}[S, S^c, X^c, X^{uc}]$ that constrains the set of admissible events so that the state traces of the controller system always satisfy the predicate G [3]. From this controller K we can derive a deterministic controller $Ctrl$ which is a function from $\mathbb{B}[S, S^c, X^{uc}] \rightarrow X^c$, which chooses the correct value for the controlled events with respect to the current state of the system and of the contract and the value of the uncontrollable variables so that the predicate G is always satisfied. We do not explain here how such controller can be computed but refer to [7][18] for more details regarding the underlying theory. All the DCS procedure is actually automatic, and implemented in the tool SIGALI [18], which manipulates STS using Binary Decision Diagram (BDD). From a computational point of view, the translation of a node and its associated control objective to a STS is automatic as well as the computation of the controller C. This controller is then automatically translated in the original framework by adding a new node $\mathscr{S}_{f_C}$ in the original program following the scheme of Figure 4, which is essential in our approach to build a compiler using DCS.

C. Modular Compilation of BZR

1) Formal syntax of BZR programs: To describe the compilation of BZR, we focus here on a formal kernel, into which other constructs, e.g., automata, can be compiled [5].

$$\begin{array}{lll}
P & ::= & d \dots d \\
d & ::= & \texttt{node } f(p) \texttt{ returns } (p) \\
 & & \quad [\texttt{contract } (D, e, e) \texttt{ with } p] \\
 & & \quad \texttt{let } D \texttt{ tel} \\
p & ::= & x \mid p, p \\
D & ::= & \varepsilon \mid p = e \mid D \texttt{;} D \\
e & ::= & i \mid x \mid \mathrm{op}(e) \mid (e, e) \mid f(e) \\
\mathrm{op} & ::= & \texttt{fby} \mid \texttt{not} \mid \texttt{or} \mid \texttt{and} \\
i & ::= & \texttt{true} \mid \texttt{false}
\end{array}$$

A program P is a sequence of nodes $d_1 \dots d_n$. A node is denoted:

$$\begin{array}{ll}
d = & \texttt{node } f(x_1, \dots, x_n) \texttt{ returns } (y_1, \dots, y_p) \\
 & \texttt{contract } (D_1, e_A, e_G) \texttt{ with } (c_1, \dots, c_q) \\
 & \texttt{let } D \texttt{ tel}
\end{array}$$

where f is the name of the node, x_i are its inputs, y_i its outputs. These outputs, and possible local variables, are defined in D. (D_1, e_A, e_G) **with** $(c_1, \dots, c_q)$ is the contract of the node f. Within a contract, D_1 represents the exported definitions, e_A an expression for the “assume” part of the contract, e_G the “guarantee” part, and c_i the controllable variables. The contract can be ommited, and then is considered to be defined as $(\varepsilon, \texttt{true}, \texttt{true})$ **with** $()$.

Definitions D are a set of equations, separated by **;**, each defining a pattern of variables $(x_1, \dots, x_n)$ by an expression e. Expressions can be Boolean constants (i), variables (x), operations op on sub-expressions, pairs of expressions, and applications of a node f on an expression. Operations are:

- e_1 **fby** e_2 which defines a new flow with the first element of flow e_1 *followed by* the whole flow e_2: this puts a delay on a flow e_2, with an initial value e_1
- **not**, **or** and **and** are Boolean operators, applied point-to-point.

2) Principle and corresponding DCS problem: The purpose of the compilation principle presented here is to show how to use a DCS tool, without modularity, within the modular compilation process of our language. Following Figure 1, we want to obtain, from each node, a STS as defined in Section II-B, in order to apply DCS on it. The obtained controller is itself a node of equations, recomposed in the target language. The compilation process is modular.

To compile a single contract node, we encode it as a DCS problem where, assuming e_A (produced by the contract program, which will be part of the transition system), we will obtain a controller for the objective of enforcing e_G (i.e., *making invariant* the sub-set of states where $e_A \Rightarrow e_G$ is true), with controllable variables X^c.

When compiling a composite contract node, the control objective is to *make invariant* the sub-set of states where, the constraint $\left(e_A \wedge (e_{A1} \Rightarrow e_{G1}) \wedge (e_{A2} \Rightarrow e_{G2})\right)$ being satisfied, $\left(e_G \wedge e_{A1} \wedge e_{A2}\right)$ is true. This objective is applied on the global transition system composing the contract and the body of the node, as well as the contracts for each of the sub-nodes. Note that the bodies of the sub-nodes are not used for the controller computation.

3) Formal compilation rules: We describe the compilation towards STS through a function Tr, from BZR equations and expressions towards tuples $(\mathscr{S}, X^u, G)$ where

- $\mathscr{S}(X, S, Y) = (T, O, Q, Q_0)$ denotes the STS obtained: for expressions, it only defines one output value. For equations, the outputs are the variables they define.
- X^u denotes additional uncontrollable inputs of the obtained STS, corresponding to the outputs of the applied sub-nodes, invoked in the program body.
- G denotes synthesis objectives from contracts of sub-nodes.

This compilation function Tr, applied on nodes, produces nodes without contracts. We consider the compilation on normalized programs, following the restricted syntax given below, defined such that the expressions e correspond to those allowed in STS.

$$\begin{array}{lll}
D & ::= & x = e \mid D \texttt{;} D \mid x = f(x) \mid x = v \texttt{ fby } x \\
e & ::= & i \mid x \mid \mathrm{op}(e) \mid (e, e) \\
\mathrm{op} & ::= & \texttt{not} \mid \texttt{or} \mid \texttt{and} \\
i & ::= & \texttt{true} \mid \texttt{false}
\end{array}$$

[3]Given a STS $\mathscr{S}$ with X^c as controllable variables and G the predicate to be made invariant, we denote $C = \mathrm{DCS}(\mathscr{S}, X^c, G)$ the operation which consists in computing a controller C so that in $\mathscr{S}_f / C$, the predicate G is always true.

Particularly, equations with subnodes applications in the expression are decomposed into equations defining intermediate variables, with either an expression or a subnode application. Compilation rules are given in Figure 6.

Rule (C-Exp): expressions are directly translated to a STS (T,O,Q,Q_0) where only $Q \neq \emptyset$, in the form of an output function for y.

Rule (C-Fby) introduces a fresh state variable s, with appropriate transition and initialization.

The rule (C-App) abstracts the application of the node f by its contract $(\mathscr{S}^c_f, A_f, G_f)$. The application are translated by composing: $\mathscr{S}^c_f$, the STS of the contract of f; $\mathscr{S}$, where the output y of the applied sub-node is considered as an additional uncontrollable variable: as the body of f is abstracted, the value of y cannot be known. This composition represents the abstraction of the application. The assume/guarantee part of the contract of the applied sub-node, as in **C-Node**, $(A_f \Rightarrow G_f)$ is added as a constraint Q of the STS. It can be noted that the point of this is to favor DCS, by giving some information of behaviors of sub-nodes: this can enable to find control solutions, which a black box abstraction would not allow. Hence it is an optimization of the modular control generation, not a necessity w.r.t. the language semantics, which it should of course not jeopardize.

Rule (C-Par): STSs from parallel equations are composed; additional variables from sub-nodes are gathered; the synthesis objective is the conjunction of sub-objectives. Assumption parts of contracts are gathered as constraints within $\mathscr{S}_1$ and $\mathscr{S}_2$ (see rule (C-App)).

Rule (C-Node) translates nodes with contracts to *controlled* nodes. It features the application of the DCS function of section II-B to the composition of the STSs from the contract and the body. This composition is constrained with the operation $\triangleright$ by the assumption part A of the contract. The additional variables induced by the abstractions of the applications are added as *uncontrollable inputs* to the STS on which the DCS is performed. This rule defines $\mathscr{S}^c_f$, A_f and G_f used for applications of f (rule C-App).

Rule (C-Prog) translates the sequence of node declarations of the complete program.

III. Supervisory Control of Computing Systems: coordination of administration loops

We now present the design process of a discrete controller, with the BZR programming language, for coordinating pre-existing autonomic administration loops of a computing system, considered as object of control. The managed system, as shown in Figure 7, is a replicated web-server system. In this simple case, we consider a single tier, with the perspective of considering mulit-tier systems. It is composed of one Apache server and replicated Tomcat servers. The Apache server plays the role of load balancer, it receives all the requests from outside and re-distributes them to active Tomcat servers.

A. Informal description of the computing system

1) Autonomic managers to be coordinated: As case-study, we propose to coordinate autonomic managers (AMs) for two

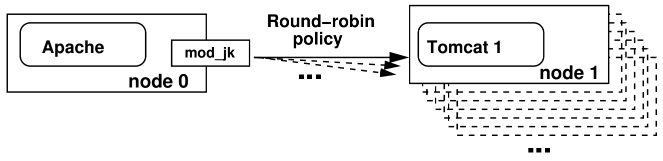


Fig. 7: Architecture of the replicated web-server system

pre-existing energy-aware administration loops, in order to ensure an efficient management of the energy consumption of a computing system with our approach, as well as as a self-repair loop. These administration loops are legacy code, in the sense that we do not design or re-design them.

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

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

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

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

3) Informal coordination policy: We want to coordinate the Ams in order to avoid the interferences described above.

$$\mathrm{Tr}(e) = (\mathscr{S}, \emptyset, \emptyset) \text{ where } \mathscr{S}(\emptyset, \emptyset, \{y\}) = \left\{ \; y = e \right. \tag{C-Exp}$$

$$\mathrm{Tr}(y = v \texttt{ fby } x) = (\mathscr{S}, \emptyset, \emptyset) \text{ where } \mathscr{S}(\{x\}, \{s\}, \{y\}) = \begin{cases} s' = x \\ y = s \\ s_0 = v \end{cases} \tag{C-Fby}$$

$$\mathrm{Tr}(y = f(x)) = (\mathscr{S} \| \mathscr{S}_f^c, \{y\}, A_f) \text{ where } \mathscr{S}(\{y\}, \emptyset, \{z\}) = \begin{cases} z = y \\ A_f \Rightarrow G_f \end{cases} \tag{C-App}$$

$$\frac{\mathrm{Tr}(D_1) = (\mathscr{S}_1, Y_1, G_1) \qquad \mathrm{Tr}(D_2) = (\mathscr{S}_2, Y_2, G_2)}{\mathrm{Tr}(D_1 \,\texttt{;}\, D_2) = (\mathscr{S}_1 \| \mathscr{S}_2, Y_1 \cup Y_2, G_1 \wedge G_2)} \tag{C-Par}$$

$$\mathrm{Tr}\left(\begin{array}{l} \texttt{node } f(X) \texttt{ returns } (Y) \\ \;\texttt{contract } (D_1, A_f, G_f) \\ \;\;\texttt{with } (c_1, \ldots, c_n) \\ \;\texttt{let } D_2 \texttt{ tel} \end{array} \right) = \left(\begin{array}{l} \texttt{node } f(X) \texttt{ returns } (Y) \\ \texttt{let} \\ \;(c_1, \ldots, c_n) = C(S, X \cup X^u)\texttt{;} \\ \;D_1 \texttt{;} D_2 \\ \texttt{tel} \end{array} \right) \text{ where } \begin{cases} \mathrm{Tr}(D_1) = (\mathscr{S}_f^c, \emptyset, \emptyset) \\ \mathrm{Tr}(D_2) = (\mathscr{S}_2, X^u, G_2) \\ \mathscr{S} = (\mathscr{S}_f^c \| \mathscr{S}_2)(X \cup X^u, S, Y \setminus X^u) \\ C = \mathrm{DCS}(\mathscr{S} \triangleright A_f, \{c_1, \ldots, c_n\}, G_2 \wedge G_f) \end{cases} \tag{C-Node}$$

$$\mathrm{Tr}(d_1 \ldots d_n) = \mathrm{Tr}(d_1)\,\mathrm{Tr}(d_2 \ldots d_n) \tag{C-Prog}$$

Fig. 6: Compilation rules of BZR towards STS and DCS application.

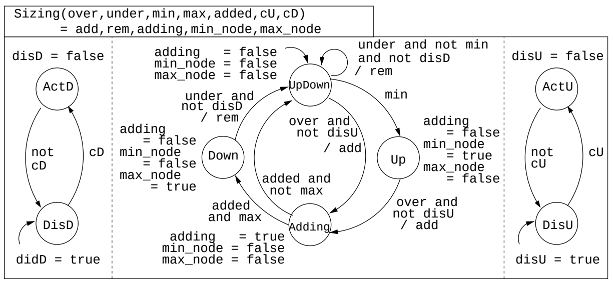


Fig. 8: Self-Sizing AM behavior

We address CPU-bound application; memory-bound applications can be regulated differently. This can be specified informally by the following policy or strategy:

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

B. Modeling the behaviors of autonomic managers

We pose the coordination problem as a control problem, first modeling the behaviors of AMs with automata, then defining controllables and a control objective in order to apply DCS. The design process of the coordination controller consists in modeling the execution phases of each autonomic manager with some controllable parts, then composing these models and applying the DCS on this composition to generate a controller that is able to force the composition to behave in accordance with the coordination policies by acting on the controllable parts on each models.

1) Model of the Self-Sizing manager: Figure 8 shows the model for the Self-Sizing AM. There are three parallel sub-automata; the two external ones manage control of the AM, and the center one is the AM, with four states, initially in `UpDown`. When an underload is notified by input `under` being **`true`**, and input `min` is **`false`**, meaning minimum number of servers is not reached, and `disD` is **`false`**, meaning downsizing is not disabled, then output `rem` triggers the removal of a server, and it goes back to `UpDown`. If `min` is **`true`**, then it goes to `Up`, where only overloads are managed upon notification by `over`: if `disU` is **`false`** (upsizing is not disabled) then `add` triggers the addition of a server, going to the `Adding` state, where neither adding or removal of a node can be requested until input `added` notifies termination. Then if input `max` is **`false`**, meaning maximum number of servers is not reached, control goes back to `UpDown`. If `max` is **`true`**, then it goes to `Down`, where only underloads are managed.

In this control automaton, the manager can be suspended: this is done with the local flows (resp.) `disU` and `disD` mentioned above, which, when **`true`**, prevent transitions where output (resp.) `add` or `rem` triggers operations. Automata on the sides of Figure 8 define the current status: disabled or not. In the case of upsizing (the downsizing is similar), initially, the automaton is in `DisU` where flow `disU` is inhibiting upsizing operations. When control input `cU` is **`true`**, it goes to the state `ActU` where operations are allowed, until `cU` is **`false`**. Hence, `cU` and `cD` define choice points, and are control interfaces made available for the coordination controller.

2) Model of the DVFS manager: The DVFS managers are local to the CPUs, and they offer no control point for an external coordinator. However they are an interesting case where it is important and useful to instrument them with *observability*, because some information on their state is necessary to take appropriate coordination decision. Figure 9 shows the corresponding automaton. Intitial state is `Normal`: at least one of the set of DVFS managers can apply both CPU-frequency increase and decrease operations.

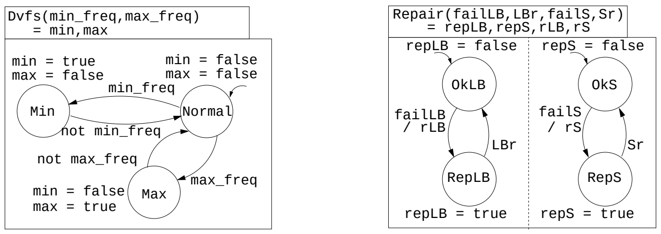


Fig. 9: Dvfs (left) and Repair (right) monitoring AMs behavior

When all nodes are in their maximum CPU-frequency, input `max_freq` notifies this with value **`true`**, and the observer goes to `Max`, where output `max` is **`true`**. Symmetrically, we have `min_freq` leading to `Min`, where output `min` is **`true`**.

3) Model of the Self-Repair manager: as shown in Figure 9 (right) there are two parallel, similar automata observing the AM for the load balancer (LB) and servers (S). The right automaton concerns servers, and is initially in `OkS`, until `failS` notifies a server failure, emitting repair order `rS` and leading to state `RepS`, where `repS` is **`true`**, until `Sr` notifies repair termination, leading back to `OkS`. Repair of the LB is similar.

C. Coordination controller

1) Automaton modeling the global behavior: Figure 10 shows how it is defined, in the body of the `main` more, by the parallel composition of automata from Figures 8 and 9, thus defining all possible behaviors, in the absence of control, of the three AMs. The interface of `main` is the union of interfaces of sub-nodes, except for `cU, cD` locals used for coordination.

```
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. 10: Coordination node: control policy on the composed models.

2) Contract: It is given by the declaration, in the **`with`** statement, of the controllables: `cU, cD`, and the Boolean expression for the control objective in the **`enforce`** statement: w.r.t. the policy of Section III-A.3, the conjunction of:

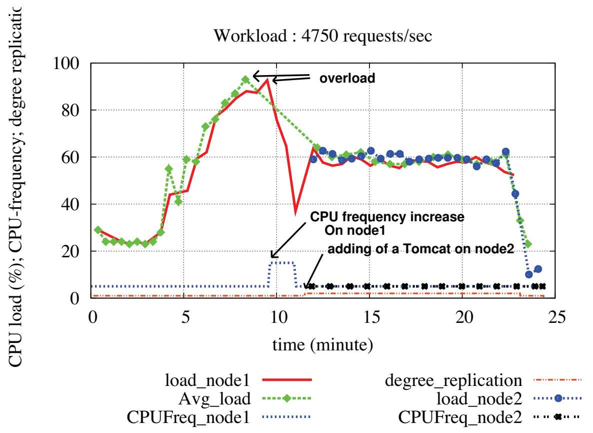
1) `Obj1 = not max` **`implies`** `disU`
2) `Obj2 = repLB` **`implies`** `disD`
3) `Obj3 = repS` **`implies`** `disU`

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

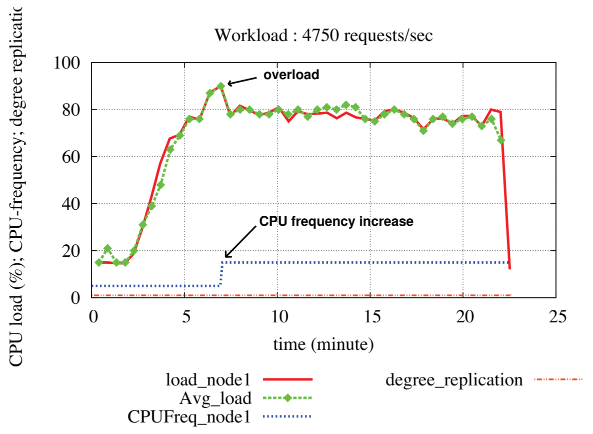
D. Compilation

The composition of automata and the coordination policy constitute the coordination controller in the composite component. The BZR programming language is compiled, generating the corresponding Java code which, given notifications of overloads and underloads, state of the local DVFS Ams, and failures of either LB or servers, will perform reconfiguration actions, enforcing the rules of Section III-A.3, and executing additions and removals of servers, and repairs of LB or servers. The control problem in our case study is simple, but illustrates the approach completely, and is implemented for the two energy-aware AMs.

We use DCS which computes a controller able to ensure the respect of this coordination policy by acting on `cU` and `cD` (Figure 10). The assembly of the set of models with the generated controller will constitute the coordination controller. BZR language allows to generate the composition in C or Java programming language, which allows to directly integrate it into a system.



(a)



(b)

Fig. 11: Workload supportable by one Tomcat Server at higher CPU-frequency: uncoordinated (a) vs. coordinated (b) coexistence.

E. Experimental evaluation

This section presents experimentations of our approach for coordinating autonomic managers. We evaluate the efficacy of the above coordination controller designed for the Self-Sizing and Dfvs managers compared to the uncoordinated coexistence of the latter as shown in Figure 11. The managed system is a CPU-bound system composed of replicated Tomcat servers. An Apache server is used as front to balance the workload between the active Tomcat servers. Each node that hosts a Tomcat server is equipped with a Dvfs. The objective of the coordination controller is to prevent Self-Sizing from increasing the number of active Tomcat servers when it is possible to increase the CPU frequency of current active Tomcat server nodes in case of overload ,i.e., the CPU load exceeds the maximum threshold. The maximum threshold for The CPU load is fixed at 90% for both managers.

Initially, during each execution, one Tomcat server is launched and its node (**node1**) is at its minimum CPU frequency. Curve **avg_load** corresponds to the average CPU load computed by the Probe for self-Sizing, curves starting by **load_node** correspond to the CPU load computed by probe for Dvfs of Tomcat node. Curves starting by **CPUFreq_node** are the CPU frequency level and curve **degree_replication** is the number of current Tomcat servers currently running. The load decrease observed in **load_node1** (near 40%) after an overload is due to the restarting of the Apache to update its list of Tomcat servers, during that period node1 does not receive any request from Apache which leads to CPU frequency decrease shown in **CPUFreq_node1**.

IV. Conclusions and Future Work

In this paper, we have been interested in the discrete control of computing systems administration, and how it can be addressed in an approach based on a programming language, which makes it concretely useable by non-specialists.

The contributions of the paper are twofold:

- the definition of a a new construct for behavioral contracts in reactive programs, enabling mixed imperative/declarative programming, and using discrete controller synthesis integrated in its compilation, using a contract methodology. This programming language feature is formally defined in terms of a DCS problem, and the semantics of its compilation is given.
- the concrete application of this technique to the design of a controller responsible of coordinating the execution of two energy-aware autonomic managers and a Self-Repair autonomic manager, which have to collaborate in order to ensure a correct and coherent management of a replicated web-server system. One advantage inherent to this technique is the computation/synthesis of the controller from the description of the autonomic managers in automata-based models with coordination policies (in this case, expressed as invariance properties).

Future and ongoing work feature language-level expressiveness, notably w.r.t. quantitative aspects: we already have features of cost functions for bounding or (one-step) optimal control [9], but timed aspects would be an improvement (see e.g. [4]). We will also consider more powerful DCS techniques, e.g., dynamic controller synthesis, and combination with static analysis and abstract interpretation techniques as in [14] to be able to consider systems handling data. We also want to explore the application of modular control synthesis [7] in order to address scalability. W.r.t. computing systems, we have ongoing work on the modeling and control of multi-tier systems, re-using the single-tier control, and additionaly considering interferences between the tiers.

References

[1] A. Auer, J. Dingel, and K. Rudie. Concurrency control generation for dynamic threads using discrete-event systems. In *Communication, Control, and Computing, 2009. Allerton 2009. 47th Annual Allerton Conference on*, pages 927 –934, 30 2009-oct. 2 2009.

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

[3] C. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer, 2007.

[4] F. Cassez, A. David, E. Fleury, K. Larsen, and D. Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *Conf. on Concurrency Theory (CONCUR)*, August 2005.

[5] J. Colaço, B. Pagano, and M. Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. In *ACM International Conference on Embedded Software (EMSOFT'05)*, September 2005.

[6] R. Das, J.O. Kephart, C. Lefurgy, G. Tesauro, D.W. Levine, and H. Chan. Autonomic multi-agent management of power and performance in data centers. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, AAMAS '08, pages 107–114, 2008.

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

[8] Christopher Dragert, Juergen Dingel, and Karen Rudie. Generation of concurrency control code using discrete-event systems theory. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 146–157, New York, NY, USA, 2008. ACM.

[9] E. Dumitrescu, A. Girault, H. Marchand, and E. Rutten. Multicriteria optimal discrete controller synthesis for fault-tolerant tasks. In *Proc. 10th Int Workshop on Discrete Event Systems (WODES 2010)*, 2010.

[10] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. Wiley-IEEE, 2004.

[11] M. Iordache and P. Antsaklis. Concurrent program synthesis based on supervisory control. In *2010 American Control Conference*, 2010.

[12] M. V. Iordache and P. J. Antsaklis. Petri nets and programming: A survey. In *Proceedings of the 2009 American Control Conference*, pages 4994–4999, 2009.

[13] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, January 2003.

[14] T. Le Gall, B. Jeannet, and H. Marchand. Supervisory control of infinite symbolic systems using abstract interpretation. In *Conference on Decision and Control (CDC'05)*, Seville (Spain), Dec. 2005.

[15] H. Liao, Y. Wang, H. K. Cho, J. Stanley, T. Kelly, S. Lafortune, S. Mahlke, and S. Reveliotis. Concurrency bugs in multithreaded software: modeling and analysis using petri nets. *j. Discrete Event Dynamic System*, 2012. to appear, `http://www.springerlink.com/content/6700x02r314300x3/`.

[16] Cong Liu, A. Kondratyev, Y. Watanabe, J. Desel, and A. Sangiovanni-Vincentelli. Schedulability analysis of petri nets based on structural properties. In *Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on*, pages 69 –78, june 2006.

[17] F. Maraninchi, Y. Rémond, and E. Rutten. Effective programming language support for discrete-continuous mode-switching control systems. In *40th IEEE Conference on Decision and Control (CDC)*, nov 2001.

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

[19] V.V. Phoha, A.U. Nadgar, A. Ray, and S. Phoha. Supervisory control of software systems. *Computers, IEEE Transactions on*, 53(9):1187 – 1199, sept. 2004.

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

[21] C. Wallace, P. Jensen, and N. Soparkar. Supervisory control of workflow scheduling. In *Advanced Transaction Models and Architectures Workshop (ATMA), Goa, India*, 1996.