Case studies in discrete control for autonomic system administration

Fabienne Boyer, Noël De Palma, Gwenaël Delaval, Olivier Gruber
UJF, LIG, France

Eric Rutten
INRIA, LIG, France

{ fabienne.boyer , noel.de_palma , gwenael.delaval , olivier.gruber, eric.rutten }@inria.fr

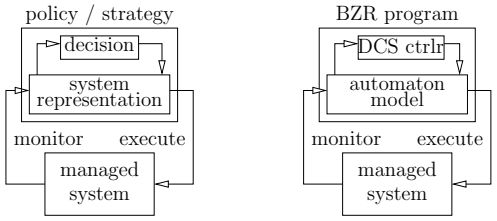
ABSTRACT

This paper presents examples of autonomic system administration issues that can be addressed and solved as discrete control problems. This shows evidence of the relevance of control techniques for the discrete aspects of closed-loop control of computing systems. The model-based control of adaptive and reconfigurable systems is considered via a reactive programming language, based on discrete controller synthesis (DCS) techniques. We identify control problems in autonomic systems belonging to the class of logical, discrete systems, and illustrate how to solve them using DCS.

1. INTRODUCTION

Autonomic computing systems [7] are adaptive systems that reconfigure themselves through the presence of feedback loops, as depicted in Figure 1(a). A feedback loop feeds on monitoring information, updates a representation of the monitored system, and decides to reconfigure the monitored system if necessary. Such feedback loops are often designed using continuous control techniques [6], more rarely using discrete control techniques [12]. This paper presents work in progress on the prospective topic of using discrete control techniques in computing systems. Previous work explored this topic at the level of software engineering for components assembly [4]. Here we describe first result of discussions on integrating the approach at an event-driven level. We follow a methodology in order to program these loops in terms of Discrete Controller Synthesis (DCS) problems [10], and illustrate it on examples of system administration loops. As in Figure 1(b), it involves modeling the system by automata, identifying configurations as states, transitions as reactions to monitored events, with appropriate reconfigurations, and finally adaptation strategies as logical control objectives.

Discrete Event Systems [1] are focused on logical aspects of systems. They concern managing properties related for example to mutual exclusions or co-locations, to sequences of events or actions which are forbidden or required. The models to capture the dynamic behavior for this class of systems are based upon language theory or labeled transition systems (Petri nets, automata, finite state machines). On the basis of such models, supervisory control techniques exist, adapting the notions classical in continuous control theory. Discrete Controller Synthesis (DCS) is a fully automated and tool-supported technique [8] that can be applied if given an automaton and a control objective. The automaton describes the potential dynamic behaviors of the system, where some of the variables conditioning the transitions are controllable. The control objective is a property to be enforced such as making the system remain in certain states characterized by a predicate, which we call *making invariant*. DCS produces, when it exists, the maximally permissive constraint on the values of controllables, such that the resulting inhibited behavior satisfies the objective.



(a) Adaptive system. (b) BZR controller.

Figure 1: BZR programming of adaptation control.

Feedback controllers in autonomic computing systems are classically designed manually, programming and debugging them in Java or C, which is tedious and error-prone. In comparison, the main advantages of using automata are that : (i) they are well-suited for the specification and expression of event sequences, (ii) they are amenable to automated analysis and verification techniques for debugging or correctness proof. Our technical background is in automata-based modeling of reactive systems, such as in synchronous programming. It is classically used in safety-critical real-time systems e.g., avionics as in the case of Airbus flight controllers.

The advantage of DCS techniques is that they are more constructive than verification: (i) they generate automatically the part of the control logic of a system which is in charge of coordinating assemblies of components : they replace tedious, difficult and error-prone hand-writing of complex synchronization automata, while ensuring the satisfaction of the control properties by construction. (ii) the generated controller is maximally permissive, which means that the synchronization constraints imposed on the assembly of

This work was partially supported by the Minalogic project MIND, and the ANR project FAMOUS.

Sixth International Workshop on Feedback Control Implementation and Design in Computing Systems and Networks (FeBID 2011)
June 14, 2011, Karlsruhe, Germany
In conjunction with The 8th International Conference on Autonomic Computing (ICAC 2011)

components is the least necessary which insures the control property; writing manually such an optimal controller would be even harder than a just correct one.

In this paper, we identify control problems in autonomic systems belonging to the class of logical discrete problems and we illustrate how they can be solved using DCS and BZR, summarized in Section 2. We first consider the simple control of a network interface, in relation with its power supply, in Section 3. We then show in Section 4 a larger model for a servers system, with problems of power and CPU consumption, quality of service and fault-tolerance. Our case studies are presented as follows: (i) describe local automata for each of the components, (ii) compose them into an assembly, and declare their interactions properties in the form of contracts, (iii) generate the controller using DCS. It shows how the coordination between the local automata propagates indirect effects of the uncontrollable inputs, following the declarative rules, through the automatically generated controller.

2. A LANGUAGE-LEVEL APPROACH

This section discusses our language-based approach for designing discrete feedback controllers, using the BZR reactive language[1] [3]. As depicted in Figure 1(b), BZR is a reactive data-flow language that is suited to program feedback loops in autonomic systems. The different components of an autonomic system are described as automata with controllable variables. The control objectives are given in the form of what we call *contracts*, in terms of predicates on the variables, with the possible addition of observers. Programmers use this mixed imperative-declarative style and the BZR compilation, involving a phase of DCS, produces automatically a controller (the decision) such that the resulting controlled automaton satisfies the control objectives. Hence, programmers benefit from DCS without mastering its formal technicality.

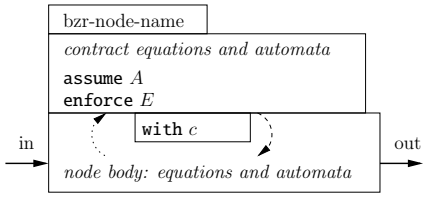


Figure 2: BZR node.

Figure 2 introduces the simplified graphical syntax for the BZR language, used in this paper. The basic structure is inherited from the Heptagon language[2]. The program is structured in data-flow nodes, which are given a name. Each has input and output flows (resp. *in*, *out*) can have local flows. The body of the node describes how input flows are transformed into output flows, in the form of a set of equations and/or automata. They are evaluated, all together at each *step* of the reactive system (hence the composition is called synchronous), taking all inputs, computing the transition, and producing the outputs. Automata, from their current state, evaluate conditions of outgoing transitions, and take the one for which it is true, if any.

The novelty of BZR is the contract construct, which is associated to a node. A contract can have its own equations and automata, to define expressions and observers. It uses flows computed in the body of the node (dotted arrow), and it will produce values for the variables defined in the `with` statement, local to the node, which are declared to be controllable (dashed arrow), thereby closing the loop. The contract can make an assumption, which can be used to model some knowledge on the environment of the node. For example, the assumption on some input might be that it satisfies a Boolean expression A which is considered to be true. The semantics of the contract is that it enforces that the Boolean expression E is maintained true in all evolutions of the system, i.e., the sub-set of states where it holds is *made invariant* for the transition system, by constraining the values of controllables.

In our development process, the language is used for specifying the discrete control part of the system with automata. Other, more data-related parts of the adaptive system are best developed in appropriate host languages like C or Java. If explicitely defined in the host language, this control part can be automatically extracted from the global specification, freeing programmers from knowing the technicities of BZR or of DCS. This way, the discrete feedback control loop of the computing system is in place, as in the upper box of Figure 1(b). Our approach is target independent, in the sense that the compilation process from automata to target code.concerns the transition function of the controller. The general structure of the generated code consists of two functions: a *reset* function for initialization purposes and a *step* function that performs one transition, with input events as parameters, producing output values, and updating the internal state of the automaton. To facilitate the integration in the target platform, we have several back-ends, generating code for various targets: C, Java, and CAML. It is possible to consider also possibly VHDL or other languages; it is essentially a matter of encoding a Boolean function and enabling the calling of functions and use of data types, from the BZR program, through linking in the host language.

An important challenge is to incorporate such automata-based controllers in actual, practical operating systems. We have ongoing work on this, but it remains an open issue to know in general how to identify and correctly use, in the different complex layers of an operating system such as e.g. Linux : (i) the practical sensors and monitors providing for reliable and significant information; (ii) the control points and actuators available in the API of the OS, enabling enforcement of a management policy; (iii) the rate and firing conditions for the transitions of the automata, and their distribution. Our prospective work on these issues gets inspiration from the specialized context of embedded systems. The challenge is to extend and generalize this to general-purpose operating systems.

3. NETWORK MANAGEMENT

3.1 Network control problem

This section focuses on the management of communications through the network, typically how messages are sent out to the network or buffered, according to their urgency and to environment conditions. The system is embedded, and its power supply is a critical feature, which is monitored. It has a CPU performing computations locally and sometimes sends and receives information to and from the outside, through a communication network, accessed with a

[1] `http://bzr.inria.fr/`
[2] `http://synchronics.wiki.irisa.fr/`

network card. The card can be turned on or put in stand-by and network availability is monitored. Messages are of two types: urgent and nomal, the latter can be delayed. The communication manager can be in either of three modes: transferring all messages, or only urgent ones, or none. Messages not transferred are accumulated in buffers, and flushed at the next possible occasion. The adaptation policy is that:

1. when the network is off, all transfer must be delayed;
2. when net load is high, transfer is not kept nominal;
3. when power is low, the network card is turned down.

3.2 Behavioral model

The dynamics of the system is modeled in terms of labelled automata, making transitions in reaction to their input flows; they can emit flows of outputs. In the BZR reactive language, such automata are written as the body of data-flow nodes, as illustrated below.

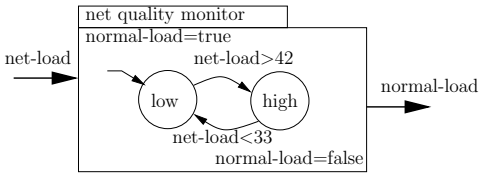


Figure 3: Component `net-quality-monitor`.

Net quality monitor. The *net_quality_monitor* component is monitoring the network connection quality, based on the input flow $net - load$. It is illustrated in Figure 3, where we can see the graphical notation used in this paper. Initially in normal load, it can go to high load if the network load, received as input flow, exceeds a given value (given here a bit arbitrarily for the example). It goes back to normal load when it is below some lower value, for hysteresis. The Boolean output flow $normal-load$ tells whether the current load level is normal or not. This model will be used further to adapt the message management.

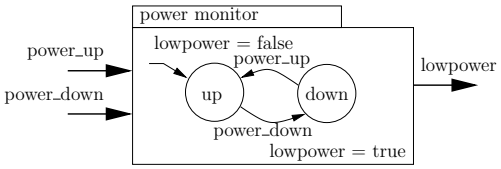


Figure 4: Component `power-monitor`.

Power monitor. The *power_monitor* component, illustrated in Figure 4, is a simple monitoring of the battery charge, based on inputs *power_up* and *power_down*. The Boolean output flow *lowpower* tells whether the current power level is low or not. This automaton can be used to control the network card (see below) or it could be used e.g., for dynamic voltage scaling (not in this paper).

Network card monitor. The *network_monitor* component, illustrated in Figure 5, is maintaining a model of the availability of the network and of the control of the card in charge of network communication. Initially it is on, in state $NetOn$. It can go to stand-by mode, in $NetSB$, either when the connection is lost (e.g. in a tunnel), upon value $false$ of input flow $netOn$, or when the network card is turned off, upon $false$ value of flow c_{on}. It goes on again when both

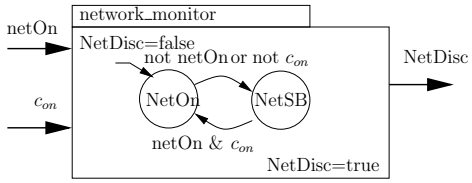


Figure 5: Component `net-monitor`.

input $netOn$ and c_{on} are true. This automaton is used to monitor network availability, and it has a control input c_{on} to be used e.g., for power management purposes.

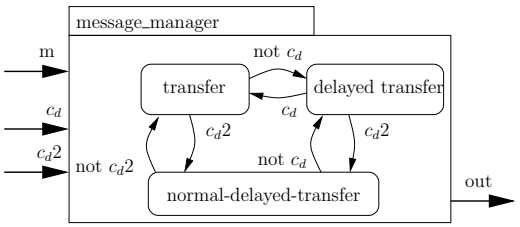


Figure 6: Component `message_manager`.

Message manager. This component transmits messages towards the network, according to their type (urgent or normal), and following one of three different modes:

- the nominal mode, $transfer$: all messages m are transferred to output *out*; when coming from the two other modes, this includes flushing those previously accumulated in a set $acc - m$ (not shown for simplicity).
- the $delayed\_transfer$ mode, where all messages are buffered, accumulated in $acc - m$, and none output.
- the $normal\_delayed\_transfer$ mode, where only urgent messages are transferred. When coming from mode $delayed\_transfer$, urgent messages accumulated are extracted from $acc - m$ and flushed. Normal ones are buffered in $acc - m$.

Each of them can be implemented with code in C or Java implementing the message transfer, accumulation, extraction, and flushing functions—not detailed here. The automaton in Figure 6 simply describes the switchings between the three modes. They are controlled by the two Boolean input flows c_d and c_d2, which are necessary and sufficient to encode the combinatorial possibilities. This automaton will later be related to reactions to network availability and power; but for now the local behavior is described independently of them.

3.3 Composition and control

Complete behavior model. The previous automata can be assembled by synchronous composition, in order to represent all possible behaviors of this assembly of components. This is equivalent to a cartesian product of automata and does not yet feature any control of their interactions. The possible behaviors will be restricted through a control policy that is specified in the contract layer. The composition is shown in the body of the node in Figure 7. Following the general scheme of Figure 2, the dotted arrows show how the outputs and states of local components will be used by the contract layer, to control the values of the controllable

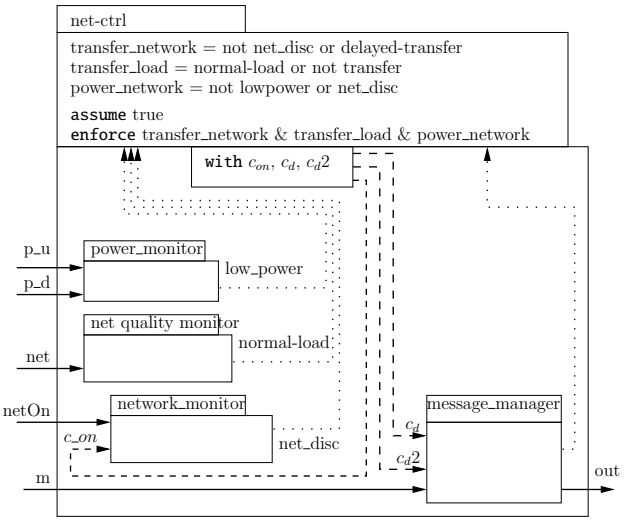


Figure 7: Composition of the local controllers.

flows c_{on}, c_d, c_d2 (in the `with` statement). They will flow back to the local components following the dashed arrows.

Control objective as declarative contract. We can illustrate how the three points from Section 3.1 can be encoded into the contract as in Figure 7. The contract makes no further assumption on its environment and enforces that the conjunction of the three properties is maintained true in all evolutions of the system, i.e., the sub-set of states where it holds is *made invariant* for the transition system.

Transfer and network: policy 1 states that when the network is off, all transfer must be delayed, which can be formulated as: $(network\ off \Rightarrow delayed\ transfer)$ which can be coded as a contract enforcing the value `true` for the Boolean `transfer_network` defined by the expression `not net_disc or delayed_transfer`

Transfer and load: policy 2 says that when the load is higher than normal, transfer is affected, and can not be nominal: $\neg normal\ load \Rightarrow \neg transfer$, coded as `transfer_load` by: `normal-load or not transfer`

Power and network card: policy 3 says that when the power is low, the network card is turned down i.e., $low\ power \Rightarrow network\ off$ which can be coded as variable `power_network` by: `not lowpower or net_disc`

Here, power is represented by a very simple model, a more refined treatment of power issues can be done using cost functions as shown in next Section.

Using DCS-based compilation, this contract is enforced, using the variables defined in the `with` statement, on the automata in the body of the node. The result is the controlled automaton, as in the upper box of Figure 1(b). We can illustrate in detail how the automata react to the inputs and how the contracts do their magics by describing step by step a simulation illustrated in the trace of Figure 8.

As long as the input *net_on* is true, in the absence of other events, the system remains in nominal behavior. When the network load exceeds its bound 42 (at instant 4), the reaction consists of a global step where the activity monitor switches to a state where *normal_load* is false, and the second term of the contract imposes quitting the $transfer$

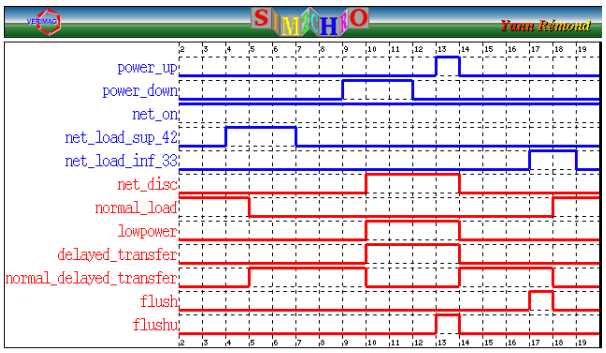


Figure 8: Simulation for the network controller.

state, towards the $system\_delayed\_transfer$ state, where only urgent messages are transferred, and others are buffered.

Upon reception of *power_down* (at instant 8), the reaction is that the power monitor switches to a state where *lowpower* is true. Consequently, the third term of the contract imposes that *net_disc* is true, meaning that the network monitor switches to state $NetSB$, using controllable c_{on}. Then, in the same reaction step, enforcing the first term of the contract imposes that the network driver goes into state $delayed\_transfer$, using controllable c_d, as well as c_d2. There, all requests are accumulated, none is output.

When *power_up* is received (instant 12), the system can switch to $normal\_delayed\_transfer$ due to the second part of the contract; it flushes only the accumulated urgent messages (event $flushu$). When later (instant 15) the the network load has returned to normal, below the given bound, the activity monitor switches back to *normal_load* true. Hence, according to the contract terms, controllable c_2 can be given value true, and the network driver returns to the state $transfer$. In this step, all the requests accumulated in the buffer are flushed, i.e. added to the output set ($flush$).

This scenario shows how the coordination between the different local automata propagates indirect effects of the uncontrollable inputs, following the declarative rules, in a way automatically generated by the DCS-based compilation.

4. SERVERS MANAGEMENT

4.1 Servers control problem

This section discusses the challenge of running a variable number of servers on several physical machines while controlling several fundamental facets: load management, provisioning, degraded modes, and fault-tolerance. It presents control aspects with sequence (before/after failure), and one-step optimal control for Quality of Service (QoS).

This system has three machine. They can be turned on or suspended (stand-by), according to resource and power management. Machine failures are fail-stop. We have three servers that can be executed on either machines 1 or 2. Servers may migrate between machines. Servers can execute in a degraded mode, costing half the resource. We have one load balancer that can be executed on machines 0, 1 or 2. Notice that the machine 0 is not capable of running servers, only the load balancer. The rules defining the policy to be enforced by the control are the following:

1. when several servers share a machine, degrade mode;
2. when a machine runs no server, switch it to stand-by;

3. restore servers of faulty machine to another one (we consider system support of repair as in e.g. [11]);

4. quality of service: add new machine if possible.

4.2 Behavioral model

Following our methodology, we model each component by its local possible behaviors and control interface.

Server. Each server $i \in [0..2]$ has a behavior modelled in Figure 9. Initially inactive in state *Inact*, upon reception of a request *req_add*, a transition is taken to state *tba* (to be added), waiting for the input *added* signaling the actual starting of the server. Then, a transition is taken, depending on input $m1$, either to the machine 1 (state $M1$) when $m1$ is true, or to machine 2 (state $M2$) when $m1$ is false. During activity of the server, it can migrate between these two machines upon corresponding values of $m1$, which is controllable. It goes back to *Inact* when *removed* is received.

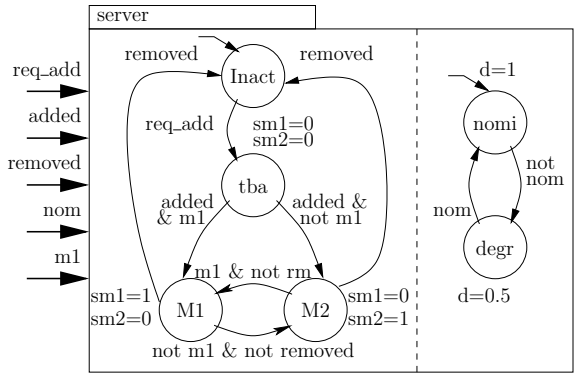


Figure 9: Model of the control of a server i.

In parallel, during activity of the server, the mode can switch between two states, according to the controllable *nom*. The server is in the nominal mode *nomi* when *nom* is true or the degraded mode *degr* when *nom* is false.

We use weights associated to states to represent computation cost and degradation level. The basic computation cost of the server i is c_i; notice that in the degraded mode it is lower than in the nominal mode. The computation cost of the server i is associated to the machine j executing the server i. The weight c_{ij} is: $c_{ij} = c_i \times sm_j \times d_i$. Taking into account heterogeneity would be having different computation costs according to different types of machines.

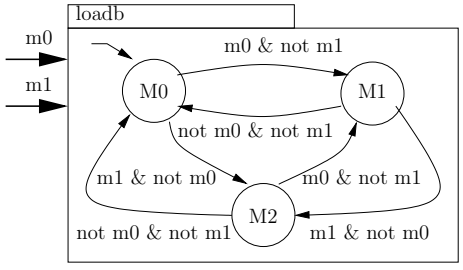
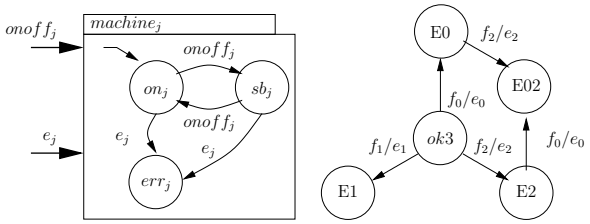


Figure 10: Model of the control of the load balancer.

Load balancer. The load balancer is initially executed on machine 0. It can migrate on machine 1 or machine 2, according to the value of the (controllable) inputs $m0$, $m1$, as shown in Figure 10. The basic consumption of the load balancer is c_{lb}, and its weight is $c_{lbj} =$ if M_j then c_{lb} else 0.

Architecture: machine j and fault model. A machine $j \in [0..2]$ is modelled as in Figure 11(a), where it is initally on; an easy variant is to have a machine initially in stand-by. A machine can be suspended (stand-by) and resumed through the controllable $onoff_j$. An error, signalled by input e_j, forces the machine in the state err_j, from where there is no recovery in this model. Each machine j has a capacity bound: B_j.

As a complement, we have a fault model, describing the assumptions about the considered faults. Indeed, if all machines can fail, then there is no possibility to ensure fault tolerance [5]. In our case, only machine 1 and 2 can host servers, machine 0 being too small. Hence, we will assume that machines 1 and 2 do not fail simultaneously. The goal of the automaton shown in Figure 11(b) is to describe exactly this hypothesis. Upon input f_j, signaling the fault of machine j, the model goes to a state E_j, transmitting the event e_j to the machine model. If $j \neq 1$, when a second fault event f_k occurs, the corresponding error event is transmitted, and the state is E_{jk}. Further fault events are then filtered out.



(a) Model of machine j (b) Fault model

Figure 11: Model of the architecture.

4.3 Composition and control

Complete behavior model. It is defined by the parallel composition of three instances of the machine model, one instance of the fault model three instances of the server model, and one instance of the load balancer model. In addition to automata, computation weights are defined in equations to describe the load for a machine j as the sum for all servers executed on j, plus possibly the load balancer: $C_j = (\sum_i c_i \times sm_{ij}) + c_{lb} \times sm_{lbj}$.

The controllable variables (i.e., to be constrained by the controller) are the following. For the machines: $onoff_0$, $onoff_1, onoff_2$. For the load balancer: m_0, m_1. For the servers: $m1_0, m1_1, m1_2$ and nom_0, nom_1, nom_2.

Control objective as declarative contract. We encode each of the four points of Section 4.1 as BZR contracts.

Degraded modes. Objective 1 states that: when several servers share a machine, they should be in degraded mode. We could approach this in the form of a purely logical objective. It could be done by excluding having two servers in mode *nomi* when they share the same machine. But a finer approach is to degrade only if it becomes necessary because of machine computing capacity bounds. For this, we will use the weights associated to states and apply the making invariant of the bounding of a cost function [9]. On every machine j, the load C_j should respect the capacity bound B_j: $\bigwedge_j (C_j \leq B_j)$. Concretely:

```
enforce (cM0 <= 10) & (cM1 <= 40) & (cM2 <= 45)
```

Machine economy. Objective 2 states that: when a machine runs no server then it should be turned off. That is, when no server i is active on machine j then the latter should not be *on*: $\bigwedge_j(\neg(\bigvee_i M_{ij}) \Rightarrow \neg on_j)$. Concretely, we have:

```
someoneon0 = onM0lb ;
someoneon1 = onM10 or onM11 or onM12 or onM1lb ;
someoneon2 = onM20 or onM21 or onM22 or onM2lb ;
economy0 = not (not someoneon0 & on0) ;
economy1 = not (not someoneon1 & on1) ;
economy2 = not (not someoneon2 & on2) ;
```

and the `enforce` statement is:

```
enforce economy0 & economy1 & economy2
```

Fault tolerance. Objective 3 defines fault-tolerance as follows: migrate and restore servers of faulty machine to another one. Migration possibilities are modelled in the tasks automata; all we have to ensure is that no task executes on a faulty machine: $\bigwedge_j(err_j \Rightarrow \neg \bigvee_i M_{ij})$ When this is made invariant by control, a machine failure will force migrations to other ones. This is an application of a general approach to migration as fault recovery [5]. In order to have a solution for this objective, bounds on machine capacity must be wide enough to accommodate all active servers at least in degraded mode. Concretely, we have ($j = [0..2]$):

```
evacuate_j = not ( err_j & someoneon_j ) ;
```

and the `enforce` statement is:

```
enforce evacuate0 & evacuate1 & evacuate2
```

When considering objectives 2 and 3 together, we can see that they are complementary. They could be combined into: $\bigwedge_j(\neg(\bigvee_i M_{ij}) \Leftrightarrow \neg on_j)$.

High quality of service. The objective 4 states that: a new machine should be added if possible in order to accomodate all servers in nominal mode. This can be seen as going to the next global state where $Q = \sum_i d_i$ is maximal, which is an application of a method for maximizing quality of service [9]. This operation of optimal synthesis is not integrated in the BZR language, but it is defined and implemented in Sigali, for advanced users, as: one-step-maximize(Q). Given that invariance is not preserved by such optimal control, the latter must be treated after the invariances.

The complete contract is the conjunction of the previously described partial contracts. The three first ones are invariances and can therefore be applied in any order. The combinatorial set of possible values for the controllable variables is gradually constrained by each of them. The last one can be applied on the result. For example, we can see how the addition of a new server can be managed, depending on the current configuration. If machine 1 is up and hosting two servers in nominal mode, while machine 2 is in stand-by, then in the complete possible behaviors, there is a choice between several possibilities. One is executing the new server on the same machine. Following objective 1, this would involving degrading modes in order to accommodate computing resource for the newcomer. Another possibility is to execute the new server on machine 2 that must be resumed from stand-by; this possibility will be chosen following objective 4. From this latter configuration, if one machines fails, by rule 3, the servers will have to share the remaining machine in degraded modes by rule 1. Or, if some severs are removed, a machine can become unused and therefore be suspended to stand-by by rule 2.

5. CONCLUSION

We propose a programming language-supported method for the design of discrete feedback controllers of autonomic computing systems, and illustrate it on examples of system administration loops. We follow a methodology in order to program them in terms of a discrete controller synthesis problem [10]. It involves identifying configurations as states, transitions as event-based reactions to monitoring by appropriate reconfigurations, adaptation strategies as logical properties and control objectives. In other works, we explore applications in a variety of contexts, on the logical coordination aspects in green computing [2], component-based systems [4], and FPGA-based reconfigurable architectures.

Perspectives are in integrating this formal reactive systems design in general-purpose operating systems (i.e., non-critical). We propose it as a form of autonomic management, and consider extension towards complex event-driven distributed systems (e.g. multicore) where control is especially difficult to express and design. The examples shown here were treated in simulation, we currently work on execution of controllers on real-world systems, by integrating the very same executable code in running system platforms. Considering complexity issues, compilation and synthesis take a mere few seconds in these case studies, but in order to evaluate potential scalability of more realistic systems, we want to apply modularity in compilation and DCS [3].

6. REFERENCES

[1] C. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, 1999.

[2] N. de Palma, G. Delaval, and E. Rutten. Qos and energy management coordination using discrete controller synthesis. In *Proc. Int. Workshop on Green Computing Middleware (GCM),* Bangalore, India, Nov., 2010.

[3] G. Delaval, H. Marchand, and E. Rutten. Contracts for modular discrete controller synthesis. In *Proc. ACM Conf. LCTES 2010,* Stockholm, Sweden, April 12-16, 2010.

[4] G. Delaval and E. Rutten. Reactive model-based control of reconfiguration in the Fractal component-based model. In *Proc. of the Int. Conf. on Component Based Software Engineering (CBSE),* Prague, June, 2010.

[5] A. Girault and E. Rutten. Automating the addition of fault tolerance with discrete controller synthesis. *Int. j. on Formal Methods in System Design*, 35(2), october 2009.

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

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

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

[9] H. Marchand and E. Rutten. Managing multi-mode tasks with time cost and quality levels using optimal discrete control synthesis. In *Proc. of the 14th Euromicro Conf. on Real-Time Systems, ECRTS'02*, 2002.

[10] E. Rutten. Supervisory control of adaptive and reconfigurable computing systems. In *Proc. 13th IFAC Symp. on Information Control Problems in Manufacturing, INCOM'09,* Moscow, Russia, June 3–5, 2009.

[11] S. Sicard, F. Boyer, and N. De Palma. Using components for architecture-based management: the self-repair case. In *Proc. of IEEE International Conference on Software Engineering (ICSE)*, 2008.

[12] Y. Wang, , H.K. Cho, H. Liao, A. Nazeem, T. Kelly, S. Lafortune, S. Mahlke, and S. A. Reveliotis. Supervisory control of software execution for failure avoidance: Experience from the gadara project. In *Proc. of WODES, 10th IFAC Int. Workshop on Discrete Event Systems*, Berlin, Germany, Sept. 2010.