A Language-Based Approach to the Discrete Control of Adaptive Resource Management *

Gwenaël Delaval, Eric Rutten
INRIA Grenoble Rhône-Alpes, France
{firstname.lastname}@inria.fr

Abstract

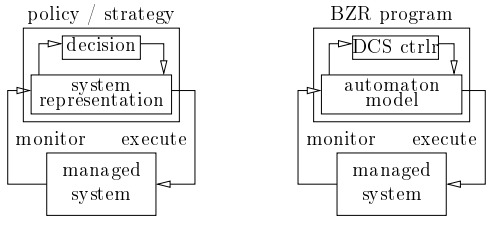
We present a novel technique for designing discrete control loops for adaptive systems. They automatically enforce safety properties on the interactions between tasks, concerning, e.g., mutual exclusions, forbidden or imposed sequences. We use a new reactive programming language, with a mechanism of behavioural contracts. Its compilation involves discrete controller synthesis, which automatically generates the correct appropriate adaptation controllers. We apply our approach to the problem of adaptive ressource management, illustrated by the example of a HTTP server.

1 Motivation: discrete control and resource management

Adaptive systems and resource management The management of dynamical adaptivity can be considered as a closed control loop, on continuous or discrete criteria. Figure 1(a) shows how, on the basis of the monitored information and of an internal representation of the system, a control component enforces the adaptation policy, by taking decisions w.r.t. the reconfiguration actions to be executed. The design of control loops with known behaviour and properties is the classical object of control theory. Applications of continuous control theory to computing systems have been explored quite broadly [10]. In contrast, logical aspects, as addressed by discrete control theory, have been considered only recently for adaptive computing systems [15].

Adaptation mechanisms can be used for the dynamical management of resources, in a variety of ways. It is a way to handle the coordination of the shared access to constrained resources, which can be exclusive or have a bounded capacity, or have constraints in the sequences in which they can be used. Such coordination can be very intricate to program in an imperative way, due to multiplication of shared resources and their independent uses by components. We will show how the BZR language, by providing hidden use of discrete controller synthesis (DCS) as shown in Figure 1(b), can help such coordination design by means of mixed imperative/declarative statements. We concentrate on logical aspects of the adaptation control, with abstract modelling of levels of quantitative consumption.

Discrete, reactive controllers One level of adaptive systems is related to events and states, defining execution modes or configurations of the system, with changes in the architecture, and in the activation of components. Reactive languages based on finite state automata, like StateCharts [9], or StateFlow in Matlab/Simulink, are widely used for these aspects. Their underlying model, transition systems, is also the basic formalism for discrete control theory, which studies closed-loop control of discrete-event and logical aspects of control systems [4]. Different reactive languages exist, like StateCharts mentioned before,



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

Figure 1: Adaptation control and BZR programming.

*This work is partially supported by the Minalogic MIND project.

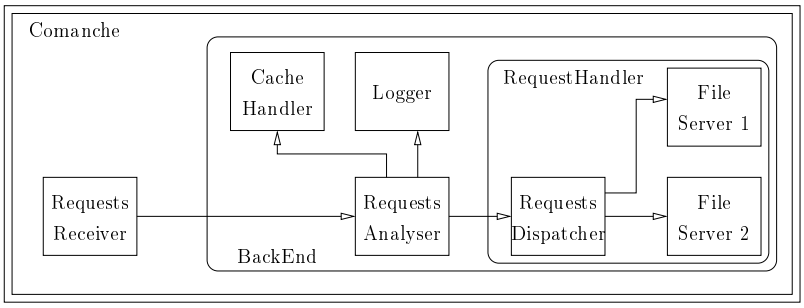


Figure 2: The *Comanche* HTTP server architecture.

and the languages of the synchronous approach [2]: Lustre, Esterel or Lucid Synchrone. They are used industrially in avionics and safety-critical embedded applications design [14]. They offer a coherent framework for specification languages, their compilers, with functionalities for distributed code generation, test generation and verification.

In the framework of discrete control, a basic technique used for the design of control loops is Discrete Controller Synthesis (DCS) [13, 4]. It consists in, from a controllable system, and a behavioural property, computing a constraint on this system so that the composition of the system and this constraint satisfies the property. An automated DCS tool exists [11], concretely connected to reactive languages. It has been applied to the automatic generation of task handlers [12], and integrated in a domain-specific language [8].

More recently the BZR language has been defined with a contract mechanism, which is a language-level integration of DCS [1, 7]. The user specifies possible behaviours of a component, as well as safety constraints, and the compiler synthesises the necessary control to enforce them. The programmer does not need to design it explicitly, neither to know about the formal technicalities of the encapsulated DCS (see Section 3).

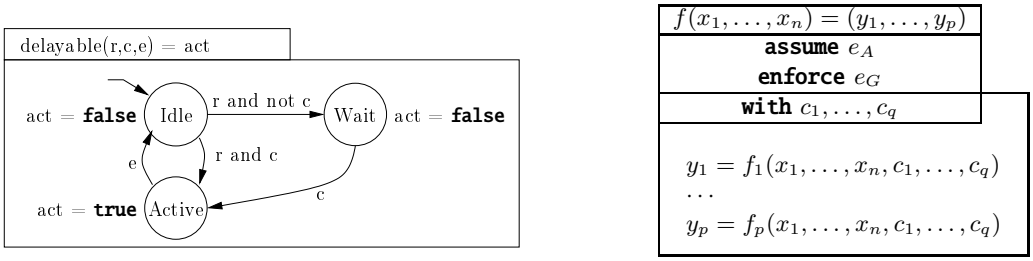
Contributions The intention of the work is to consider the adaptive management of resources as a discrete control problem. The current status of the work is a language-based solution, to generate controllers for the discrete loop; they are correct by construction, and handle safety properties on the interactions of tasks around resources. In the event and state-based aspects where it is applicable, the DCS formal method is made usable by non-experts, as it is encapsulated in a programming language and compiler. The generated code (C or Java) can be concretely integrated in the run-time executives. We make a study of the example of a component-based HTTP server, modeled with patterns inspired from [8]. Prospective results expected from ongoing work are the integration of our technique with several targets: the Fractal component-based framework [3], FPGA-based reconfigurable architectures, and the Orccad control systems design environment.

2 Example of a HTTP server

As an example application of our techniques, we consider a HTTP server, illustrated in Figure 2, with its adaptation requirements. It is a variation [5] of the *Comanche* HTTP server used as an example in tutorials[1] for the Fractal component-based middleware platform [3]. Incoming requests are read by the *RequestReceiver* component, which transmits them to the *RequestAnalyser* component. The latter can forward them to the *RequestHandler* component, which queries a farm of *file servers* to solve the request, through a *RequestsDispatcher*. *RequestAnalyzer* can also consult a cache in the *CacheHandler* component, in order to master the response time and keep it as short as possible. A *Logger* component enables logging of transactions, and can be connected to the *RequestsAnalyser*.

The available degrees of dynamical reconfiguration are that the *File Servers*, *CacheHandler* and *Logger* components can be activated or deactivated. The resources involved in the system and its dynamical management are the consumption in energy, and an exclusive resource shared by the *CacheHandler* and *Logger*. Requirements for these evolutions define the adaptation policy:

[1] `http://fractal.ow2.org/tutorial/`



(a) Mode automaton node.

(b) BZR contract node.

Figure 3: Example of programs in graphical syntax.

1. the *CacheHandler* component should be activated if there is a high number of similar requests;
2. the number of deployed file servers must be adapted w.r.t to the overall load;
3. when logging is required by the system administrator, then it should not be denied.
4. logging and cache handling should be exclusive, due to the access to some other resource.

These rules must be enforced in the adaptive system by the controller as illustrated in Figure 1(a).

3 Programming reactive systems in BZR

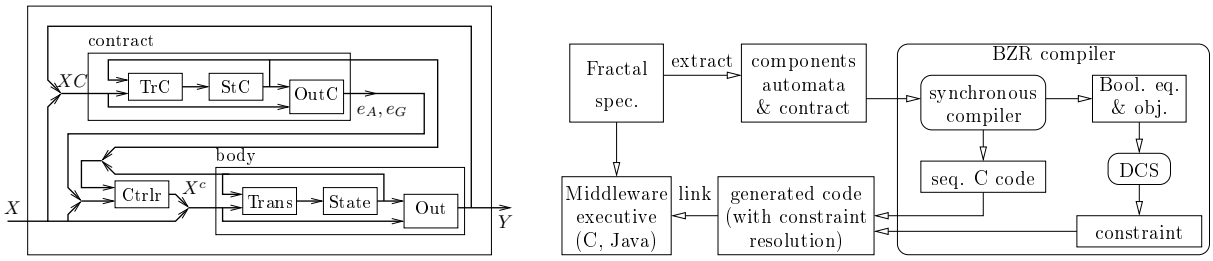
In this section we first briefly introduce the basics of the Heptagon language, to program data-flow nodes and hierarchical parallel automata [6]. We then define the BZR language, which extends Heptagon with a new contract construct [1, 7]. As for the reactive languages introduced in Section 1, the basic execution scheme is that at each reaction a step is performed, taking input flows as parameters, computing the transition to be taken, updating the state, triggering the appropriate actions, and emitting the output flows.

Data-flow nodes and mode automata Figure 3(a) shows a simple example of a Heptagon 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 which can be seen as Boolean flows), and outputs (here: the Boolean act), which is true when the task is active. 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 [6] in the sense that 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.

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 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.

Contracts in the BZR language With this new construct, DCS is encapsulated in the compilation of BZR. Models of the possible behaviours of the managed system are specified in terms of mode automata where non-determinism can be introduced by means of controllable variables ("**with**" part of contracts). These controllable variables are given free by the programmer, and their value are given by the automatically computed controller. Adaptation policies are specified in terms of contracts, on invariance properties to be enforced by the controller. Compilation yields a correct-by-construction controller, produced by DCS, as shown in Figure 1(b), in a user-friendly way: the programmer does not need to know technicalities of DCS.

As illustrated in Figure 3(b), we associate a *contract* to a node. It is itself a program, with its internal



(a) DCS problem.

(b) Development process for BZR, in the case of Fractal.

Figure 4: BZR compilation and development.

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 $C = \{c_1, \ldots, c_q\}$ of local controllable variables will be used for ensuring this objective. This contract means that the node will be controlled, i.e., values will be given to $c_1, \ldots, c_q$ such that, given any input trace yielding e_A, the output trace will yield e_G.

Without giving details [7] out of the scope of this case study, we compile such a BZR contract node into a DCS problem as in Figure 4(a). 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 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 $c_1, ...c_q$. 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.

Integration of our target-independent language and compiler in a development process follows the general scheme illustrated in Figure 4(b) in the case of Fractal [3]. The control part is extracted from the adaptive system, in the form of a BZR program. Its compilation is made in derivation of the main system development process, and produces the synthesized constraint on controllables, composed with the sequential C code for the automata. They are assembled and linked back into the global executive.

4 Solving adaptive resource management control

We apply the BZR programming methodology: first describe possible behaviours with non-deterministic imperative automata, then specify control objectives in the declarative contract.

Behaviours Locally, each component has its own activity automaton but we show only the meaningful ones, in Figure 5, from left to right. The *RequestAnalyser* has an observer automaton detecting periods with a high number of similar requests (state `Dense`) or not (state `Norm`); transitions are taken upon the uncontrollable `d` produced by the analyser. In *RequestHandler*, a reconfiguration automaton handles the *File Servers* that are deployed or shut down; in `H2` two are up, in `H1` just one; transitions are controllable upon $\mathtt{c}_h$, server $f2$ is started/stopped on their occurrence. In *BackEnd*, a reconfiguration automaton handles logging and cache, with three configurations: none of them active (`N`), cache active (`C`), logging active (`L`). The fact that this automaton is programmed with no state with both active takes care of the *exclusion requirement 4* of Section 2. Transitions are conditioned by two variables: the uncontrollable `l` (coming from the environment: e.g., logging request from a human administrator), and the controllable `c`. If `l` is true then the configuration starting the logger is taken, and if it is false, then the logger is stopped; the cache can be activated when `c` is true, only if the logger is not. This programming also takes care of the priority in *requirement 3*.

Other automata model the activation state of components, with a standard pattern given in Figure 5, where i can be c for the cache handler, l for the logger, $f2$ for the file server 2. The `start`s and `stop`s

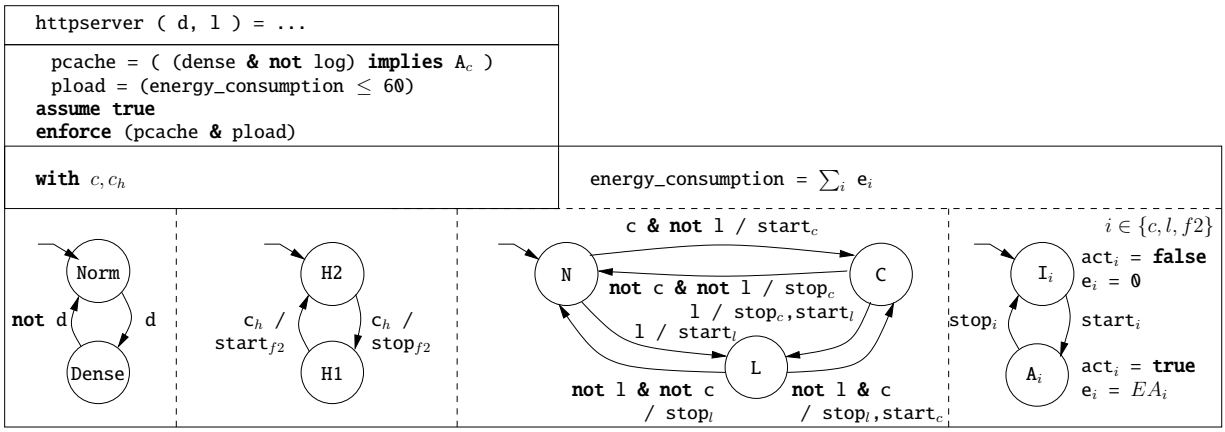


Figure 5: Behavior model of the example.

lead respectively to state `A`, with power cost EA, and to the inactive state `I`, with no cost; they are received from the reconfiguration automata. Costs when active are $EA_c = 50$, $EA_l = 30$, $EA_{f2} = 25$.

The global automaton, i.e., the complete control part of the system as in Figure 1(b), is then obtained by the parallel composition of local automata. An additional parallel equation defines the composition of the costs, here summed up at every instant.

Contract It is in the upper part of Figure 5: it is itself a program, with its own equations. Two controllable variables, defined in the **with** part, will be used for ensuring two objectives. The policy defined by the rules of Section 2 is taken care of, for 4 and 3, by programming as above. The two remaining policies are not explicitely programmed (i.e., in the imperative part) but declaratively stated:

1. the cache policy (*requirement 1*) is encoded as : `pcache = (dense & not log) implies A_c`
2. load-related adaptation (*requirement2*) is coded: `pload = (energy_consumption ≤ 60)`

The contract states that the conjunction of both requirements must be enforced by control.

Simulation and typical scenario The above BZR program can be compiled and executed, or simulated with a chronogram-like graphical simulator [2] as shown in Figure 6. The executable code generated by the compiler cana also be linked with a run-time executive as in Figure 4(b).

[2] courtesy of Verimag

A typical scenario showing the intervention of the controller on the system, so that control objectives are preserved, is as follows. Starting from (`Norm`, `H2`, `N`), when `d` occurs (step 11), by requirement 1 (first part of the contract) the cache is started, and by requirement 2 (second part of the contract) server $f2$ is stopped (otherwise the available load is overshot). Hence we go in state (`Dense`, `H1`, `C`). When `l` occurs (step 17), then by requirements 3 and 4, programmed in *BackEnd*, the cache is stopped, the log is started, and by requirement 2 (second part of the contract) the server $f2$ can be started again, and we go to (`Dense`, `H2`, `L`).

5 Conclusion and perspectives

We propose a novel technique to design discrete control loops in adaptive systems, e.g., for the safe management of resources. We use a programming language ensuring logical safety properties of the tasks sequencings and mode changes. We illustrate the approach with a HTTP server example. Its compilation performance is subject to the natural complexity of the algorithms, but we claim that it automatically generates an executable control solution, which is to be compared with manual programming, verification and debugging, which is even more costly. The execution cost of the controller is very small, as well as its computation time by DCS (a few ms on a standard Pentium, 2.33 GHz). Ongoing and further work includes integration of FRACTAL and BZR, enriching the models with optimization aspects [12], and defining libraries of standard control models and contracts.

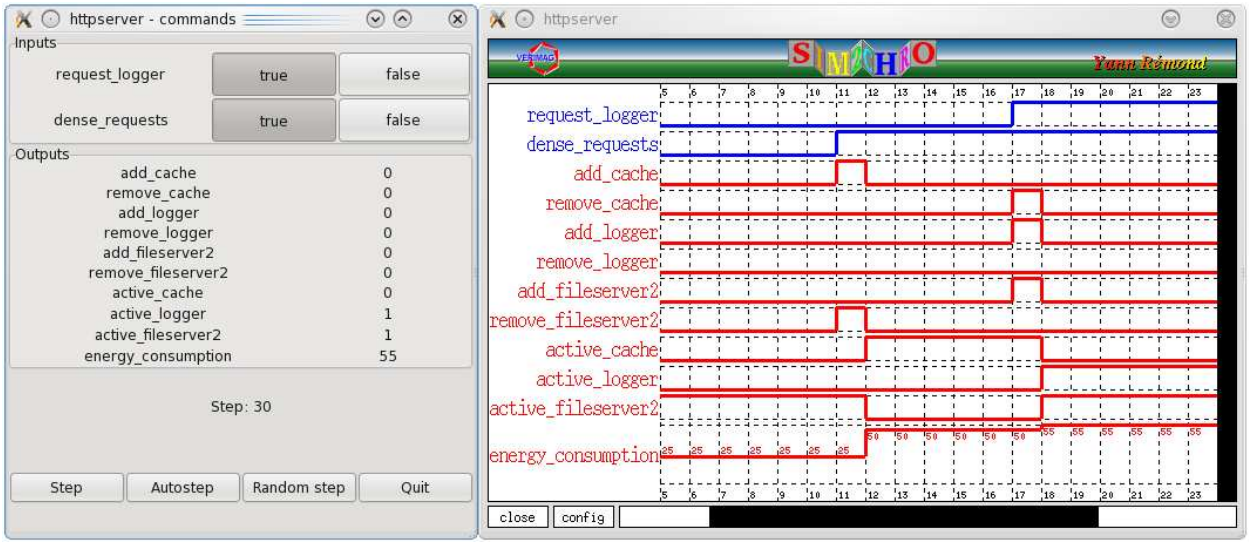


Figure 6: Simulation.

References

[1] S. Aboubekr, G. Delaval, and E. Rutten. A programming language for adaptation control: Case study. In *Proc. of the 2nd Workshop on Adaptive and Reconfigurable Embedded Systems, APRES'09*, 2009. Special Issue of SIGBED Review `http://sigbed.seas.upenn.edu/vol6_num3.html`.

[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] 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.

[4] C. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Kluwer Acad. Publ., 1999.

[5] Franck Chauvel, Olivier Barais, Isabelle Borne, and Jean-Marc Jézéquel. Composition of qualitative adaptation policies. In *23rd IEEE/ACM Int. Conf. on Automated Software Engineering - ASE'08*, L'Aquila, Italy, sep 2008.

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

[7] G. Delaval, H. Marchand, and E. Rutten. Contracts for modular discrete controller synthesis. In *Proc. of the ACM Conf. on Languages, Compilers and Tools for Embedded Systems, LCTES*, 2010. `http://hal.inria.fr/inria-00436560`.

[8] Gwenaël Delaval and Eric Rutten. A domain-specific language for multi-task systems, applying discrete controller synthesisd. *Journal on Embedded Systems* (special issue on Synchronous Paradigm in Embedded Systems), 2007(84192):17, January 2007.

[9] D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM TOSEM*, 5(4), 1996.

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

[11] 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.

[12] 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.

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

[14] Esterel technologies. Scade: model-based development environment dedicated to safety-critical embedded software, 2010. `http://www.esterel-technologies.com/`.

[15] Y. Wang, T. Kelly, and S. Lafortune. Discrete control for safe execution of IT automation workflows. In *Proc. of the 2007 EuroSys Conf.*, 2007.