Reactive model-based control of reconfiguration in the Fractal component-based model *

Gwenaël Delaval, Eric Rutten

INRIA / LIG, Grenoble, France – {gwenael.delaval | eric.rutten}@inria.fr

Abstract. We present a technique for designing reconfiguration controllers in the Fractal component-based framework. We obtain discrete control loops that automatically enforce safety properties on the interactions between components, concerning, e.g., mutual exclusions, forbidden or imposed sequences. We use a reactive programming language, with a new mechanism of behavioural contracts. Its compilation involves discrete controller synthesis, which automatically generates the correct adaptation controllers. We apply our approach to the problem of adaptive ressource management, illustrated by the example of a HTTP server.

Keywords: adaptive systems, reconfiguration control, components, contracts, model-based approach, reactive programming, discrete controller synthesis, resource management.

1 Motivation and example application

1.1 Model-based control for Fractal

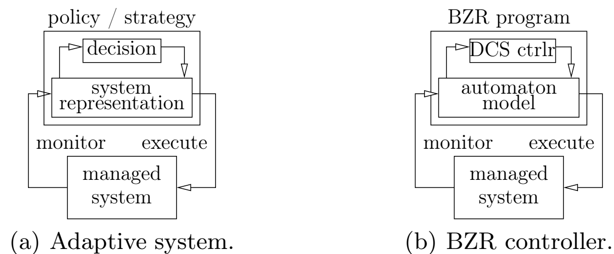
The Fractal component-based approach Fractal [4] is a modular component model that can be used with various programming languages, to design, implement, deploy and reconfigure systems and applications, from operating systems to middleware platforms and to graphical user interfaces. It is equipped with a hierarchical structure, and puts an emphasis on reflexivity, in order to support adaptation and reconfiguration. Components are the basic construct enabling the separation of interface and implementation. They support the explicit representation of the software architecture, which is essential for adaptivity and manageability. It is the basis for performing run-time software reconfiguration and system supervision.

Management of components then consists of monitoring, control and dynamical reconfiguration of the architecture. The composite structure offers a uniform construct for this: introspection functionalities enable monitoring the state of system, while re-configuration actions allow to change it. A whole range of levels of control is supported, from black box with no control, to full fledged introspection. A lifecycle controller defines the adaptive behavior of components.

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

Adaptive systems and resource management Computing systems are proliferating, in a great variety of environments, typically embedded systems. They have to be more and more *adaptive*: they must perform reconfigurations in reaction to changes in their environment concerning e.g., power supply, communication bandwidth, quality of service, or also typically dependability and fault tolerance for a *safe execution*. Another motivation for adaptative and autonomic systems is the complexity of administration, and the need for automated techniques replacing manual or *ad hoc* management [16]. The run-time management of this *dynamical adaptivity* is the object of research on ways to design and implement adaptation strategies. One approach is autonomic computing [15], where functionalities are defined at operating system or middleware level, for sensing the state of a system, deciding upon and performing reconfiguration actions.

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 monitor 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 [16]. 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 [14]. In contrast, logical aspects, as addressed by discrete control theory, or even by hybrid systems combining continuous and discrete dynamics, have been considered only recently for adaptive computing systems [23]. Even if qualitative aspects are considered for long e.g., in quality of service issues (QoS) [17], the technical approach did not involve the benefits of control techniques.



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

Fig. 1. Adaptation control and its BZR programming.

We address this with the BZR programming language [10] as shown in Figure 1(b). The class of dynamical changes addressed is the run-time switching between configurations characterized by stable states, in which a given computing activity is executed. As an example of application, 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. We concentrate on logical aspects of the adaptation control, with abstract modelling of discrete levels of consumption of quantitative resources.

Control based on reactive models 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 are widely used for these aspects, like StateCharts [13], or StateFlow in Matlab/Simulink , or UML variants. 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 systems [6]. Different reactive languages exist, like StateCharts mentioned before, and the languages of the synchronous approach [3]: Lustre, Esterel or Lucid Synchrone [8]. They are used industrially in avionics and safety-critical embedded applications design [22]. They offer a coherent framework for specification languages, their compilers, with distributed code generation, test generation and verification.

In this framework, a basic technique used for the design of control loops is Discrete Controller Synthesis (DCS) [21,6]. 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 [18], connected to reactive languages. It has been applied to the automatic generation of task handlers [19], and integrated in a domain-specific language [11]. It was also applied to fault-tolerance, in an approach where fault recovery is seen as the reconfiguration of computing activities from a given placement on the execution architecture, by exploiting its redundancy, and switching and migrating to another one where the faulty processor is not used any more [12]. More recently the BZR language has been defined with a contract mechanism, which is a language-level integration of DCS [1,10]. 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. It is briefly explained in (see Section 3), with more detail in Appendix A.

Contributions We present an integration of reactive model-based techniques for the control of reconfiguration, in the Fractal component-based framework. We concentrate on the lifecycle control, and present a structural association with reactive nodes in the BZR language. It is a language-based solution, to generate correct by construction controllers for the discrete loop, for safety properties on the interactions of components. 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. This way, designers

can benefit from, on the one hand, the Fractal approach to component-based systems, and on the other hand, the BZR language for the automated synthesis of reactive control.

In the following, the example application is presented in Section 1.2. Brief background on the Fractal component-based model and on reactive models and DCS is given in Sections 2 and 3. The structural integration of reactive control in Fractal is described in Section 4, illustrated with the application, and Section 5 sketches execution-level integration.

1.2 Example of a HTTP server

We consider a HTTP server, illustrated in Figure 2, with its adaptation requirements. It is a variation [7] of the *Comanche* HTTP server used as an example in tutorials[1] for the Fractal component-based middleware platform [4]. 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 latter can monitor e.g., a high number of similar requests.

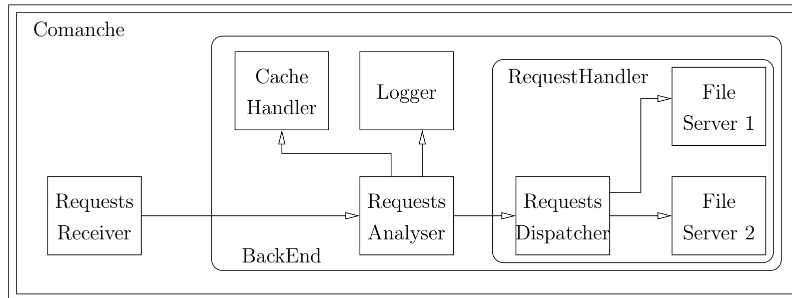


Fig. 2. The *Comanche* HTTP server architecture.

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. the *CacheHandler* is activated in case of high number of similar requests;

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

2. the number of deployed file servers must be adapted w.r.t to the overall load;
3. a logging request by the system administrator, 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 by the adaption controller as in Figure 1(a).

2 The Fractal component-based model

We briefly introduce the basics of the Fractal component model [4], in order to define the structures to which we propose a behavioral extension.

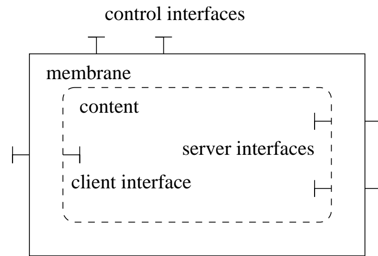


Fig. 3. A Fractal component

2.1 Components and composites

A Fractal component, as shown in Figure 3, is equipped with an interface giving accesses to the component, of two kinds: server interfaces accept incoming operations invocations, while client interfaces support outgoing operation invocations. It has a content, which can consist of a finite set of sub-components. Special interfaces concern control aspects, which are handled in the membrane.

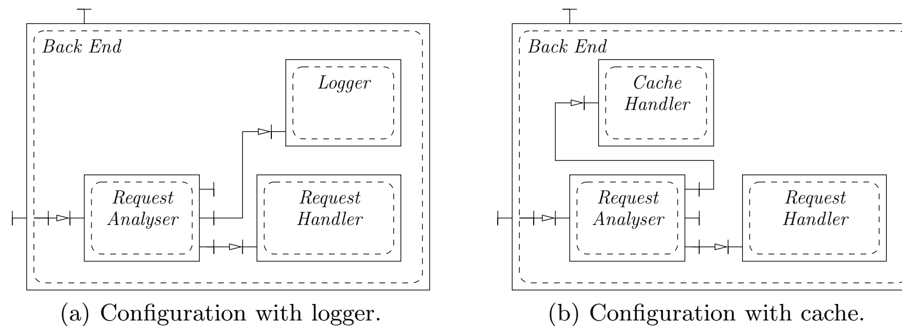
The other mechanism in Fractal to define architectures is binding, which is connecting interfaces of components: this is the only way to make them communicate. A primitive binding connects one client interface with one server interface; composite bindings can be defined also: bindings are components themselves. Figure 4(a) gives an example for the *BackEnd* component of Section 1.2. It features three sub-components, connected by appropriate bindings. *Request-Analyser* and *Logger* are base components, while *RequestHandler* is a composite, itself decomposed into binded sub-components.

A Fractal component is equipped with a membrane, which supports interfaces to introspect and reconfigure its internal features. It is composed of several controllers, provides an explicit and causally connected representation of the content, and performs control on the sub-scomponents, e.g., suspending, checkpointing, resuming activities, installing bindings.

2.2 Reconfiguration control in Fractal

There are several levels of control, from base components (black boxes, with no introspection or reconfiguration capability), to components exposing their external structures (clients and servers available), and to components exposing their internal structures, and providing for reconfiguration actions. Examples of possible controllers are managing:

- attributes (through get and set operations),
- bindings (binding and unbinding client interfaces to server interfaces),
- contents (adding and removing subcomponents),
- and, most interestingly to us, lifecycle, where explicit control is given over the main behavioral phases of a component.



(a) Configuration with logger. (b) Configuration with cache.

Fig. 4. Two configurations of the *Back End* component

Reconfiguration actions which we will consider in this work will be adding and removing a component, and binding and unbinding primitive connections.

Implementations of Fractal exist in different contexts, academic and industrial, and embedded in different host languages, namely C (with several variant, like Cecilia, or Think targeted at embedded systems, or their industrialization MIND[2]) or Java [4] (Julia). Concerning reconfiguration mechanisms, libraries for introspection and actions have been proposed in Fscript[3] [9].

Modelling the example of Section 1.2 in Fractal involves constructing a component architecture simply following the informal drawing of Figure 2, for the different configurations. For the case of the *Back End* component, Figure 4(a) shows, in a classical Fractal graphical syntax, a configuration with the logger active, while Figure 4(b) shows another configuration, where only the cache is active. The reconfiguration themselves are described by giving the actions requested in order to perform them. In this case, reconfiguring the *Back End*

[2] `http://mind.ow2.org`
[3] `http://fractal.ow2.org/fscript/`

component from configuration 4(a) to configuration 4(b) involves the following sequence: remove the *Logger*, unbind it from the *RequestAnalyser*, add the *CacheHandler*, bind it with the *RequestAnalyser*.

Fractal and adaptive systems, and behavioral models have been associated in the litterature, following a variety of approaches e.g. parallel frameworks [5] or formal models [2,20]. Our work is specific in that it concentrates on reconfiguration control, and proposes to relate this aspect of Fractal with the synchronous approach to reactive systems and particularly DCS techniques, which we present next, in order to design correct by construction control loops.

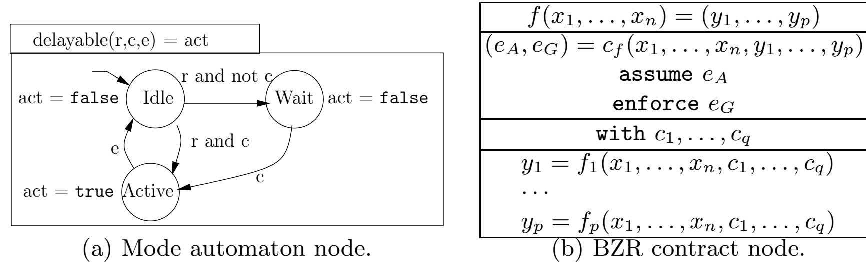
3 Programming reactive systems in BZR

In this section we first briefly introduce the basics of the Heptagon language, to program data-flow nodes and hierarchical parallel automata [8]. We then describe the BZR language, which extends Heptagon with a new contract construct [1,10]. As for all reactive languages introduced in Section 1.1, 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.

3.1 Data-flow nodes and mode automata

Figure 5(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 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 [8] 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, 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.



(a) Mode automaton node. (b) BZR contract node.

Fig. 5. Example of programs in graphical syntax.

3.2 Contracts in the BZR language

This new contract construct encapsulates DCS in the compilation of BZR [1,10]. 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, as illustrated in Figure 1(b), in a user-friendly way: the programmer does not need to know technicalities of DCS.

As illustrated in Figure 5(b), we associate 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 instances of the same body, that differ in assumptions and enforcements.

Without giving details [10] out of the scope of this paper, we compile such a BZR contract node into a DCS problem as in Figure 6. 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 subset 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.

The BZR compiler is implemented on top of the Heptagon compiler and the Sigali DCS tool [18]. Its performance is subject to the natural complexity of

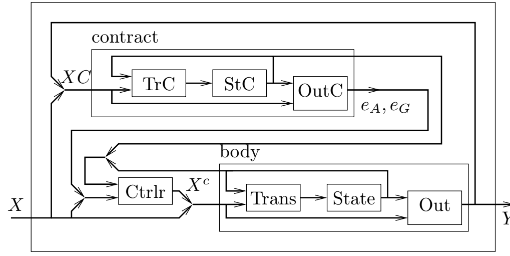


Fig. 6. BZR contract node as DCS problem

the algorithms, which is exponential just as the model checking algorithms are, but there are arguments in favor of its scalable use:

– the technique is applied to a relatively small fragment of the complete system, which is only its reactive, state-based control part; all the data-oriented code of the system, which is usually its vast majority, is not taking part in this controller synthesis, thanks to the separation of concerns offered by the components structure;
– state space exploration algorithms used in model-checking as well as in DCS have known notable progress, due to years of research on efficient codings, such as symbolic techniques and BDDs (Binary Decision Diagrams); as a result the size of systems amenable to these techniques has grown substantially; this point, related to the previous one, makes us claim that we can handle the specialized control part extracted from large systems;
– it automatically generates an executable control solution, correct by construction, which is to be compared with manual programming, verification and debugging, which would be extremely difficult, as soon as the system is too large to be designed by a small team. It is then even more costly in time, and can involve days or weeks of engineering time.

Moreover, the use of modular DCS can help to reduce significantly this cost [10].

The execution cost of the controller is very small. Integration of our target-independent language and compiler in a development process follows the general scheme as in Figure 12 in the case of Fractal [4], as explained in Section 5. 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.

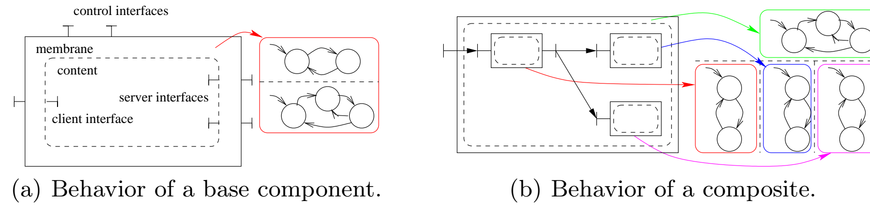
More detail on the BZR language is given in examples in the next sections, illustrated with nodes and contracts, and in Section 5.1 for its implementation. Essentials on DCS are given in Appendix A, and a concrete BZR syntax example in Appendix B.

4 Associating reactive control with a Fractal model

4.1 General approach

Our extension to Fractal consists of the addition of elaborate behavioral controllers, in the form of labelled transition systems and contracts, which were not present previously in Fractal, where only an informal life cycle was defined.

We follow the Fractal hierarchical components structure, and describe the way we associate, at each level of component, automata-based models of the behavior of the part relevant for reconfiguration in this control scope i.e., the activation and deactivation, and binding and unbinding of the direct sub-components. The simple principle is illustrated for base components in Figure 7(a): the control defined in the membrane is modelled with automata, which can be composed in parallel when they describe different aspects of the behavior. In particular, there is an explicit representation even of sub-components that are not activated but could be. In order for a predictive control to be applied, the behavioral model must feature a representation of absence. For composites, which can be dynamical or not, Figure 7(b) sketches the composition of the behaviors of the component itself, with the parallel composition of behaviors of sub-components. It can be noted that the synchronous composition is associative and commutative.



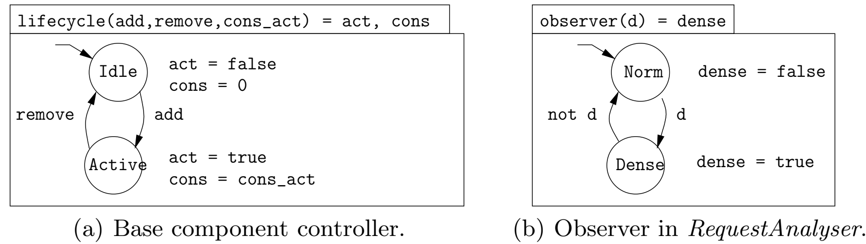
(a) Behavior of a base component. (b) Behavior of a composite.

Fig. 7. Automata modelling the behavior of Fractal components.

4.2 Base components

Every component has a lifecycle controller, which indicates its activation state, as shown in Figure 8(a). The `adds` and `removes`, received from the reconfiguration automata, lead respectively to state `Active` and to the inactive state `Idle`. For optimisation, this is useful essentially if the upper level composite is dynamic, i.e., does perform activations and deactivations of this component.

Other aspects of the component behavior, even if they are not distinguished in Fractal component architecture, can be meaningfully modelled in automata or equations. In our example, Figure 8(a) features equations associated to the states of the mode automaton, defining `cons`, which indicates the level of consumption of a resource (here, it is related to energy); in the active state it is defined by a



(a) Base component controller. (b) Observer in *RequestAnalyser*.

Fig. 8. Models of base components behaviors.

constant `cons_act`, whereas in the inactive state it is null. In the application example, costs when active are given the following values, for the cache: 50, for the logger: 30, and for the file server 2: 20. This lifecycle node will be instanciated for each of the components, as shown in the case of the *CacheHandler* in Figure 9 in concrete BZR textual syntax. This is done similarly for `requests_dispatcher`, `logger`, `file_server1`, `file_server2`, and `requests_receiver`.

The *RequestAnalyser* component can detect phases with a high number of similar requests: it has a second automaton shown in Figure 8(b), which distinguishes the two states `Norm` and `Dense`, upon input `d`.

```
node cache_handler(add,remove:bool) returns (active:bool;cons:int)
let    (active,cons) = lifecycle(add,remove,50);
tel
```

Fig. 9. Instanciation of the `lifecycle` node for *CacheHandler*.

4.3 Composites

Static composites. Composites can be associated with the same behavioral information as base components. The transitions of their lifecycle controller, have to be propagated to the sub-components, for them to be added and removed, according to the composition semantics chosen for Fractal.

Behavior models of sub-components are composed in parallel, as in the right part of Figure 10 for the example of the *RequestHandler*, where sub-nodes are invoked for the requests dispatcher, file servers and request analyser. Additional equations and automata can be defined as well. Typically, in our example, the costs of sub-components are summed up in order to define the composite cost, with the equation defining `cons` in terms of values coming from sub-nodes.

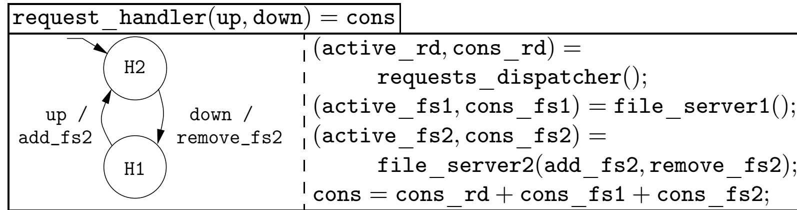


Fig. 10. Model of the *Request Handler* composite.

Reconfigurable composites. If the composite is static i.e., not reconfiguring explicitely its sub-components architecture, then its behavior is sufficiently defined by the elements described above. For a dynamically reconfigurable component, we associate an additional explicit automaton where, basically, states correspond to configurations, and transitions describe which reconfigurations are possible. Parallel automata can handle independent aspects or dimensions of the configurations. Exchanges of events between parallel automata can define synchronizations between allowed reconfigurations. We apply the BZR programming methodology: first describe possible behaviours with imperative automata, then specify control objectives in the declarative contract. In the framework of Fractal, we can have several levels of specification for a reconfiguration policy

Reconfiguration policy by automata. This consists simply in programming in terms of automata i.e., specifying explicitely the requested behavior. The left part of Figure 10 shows the reconfiguration automaton for *RequestHandler*, that handles two configurations for the *file servers* that are deployed or shut down; in `H2` two are up, in `H1` just one; transitions are taken w.r.t. inputs `up` and `down`, and the file server 2 is added or removed accordingly. There is no contract at this level, but we will see later that `up` and `down` will be used as controllables.

Another example is in Figure 11 for *BackEnd*. The concrete code for this part of the example can be seen in appendix B. Possible behaviors are described in a reconfiguration automaton, handling logging and cache with three configurations: cache active (`C`), logging active (`L`), or none (`N`). The fact that this automaton is programmed with no state with both active takes care of the *exclusion requirement 4* of Section 1.2. Transitions are conditioned by two variables: the uncontrollable `l` (coming from the user), and `c`, which will be used as a controllable. 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 takes care of *requirement 3*.

Such programming, not making use of contracts, or DCS, can be validated with the classical methodology of verification, typically with model-checking.

Reconfiguration policy by logical contract. More originally, specifications with contracts amount to specify declaratively the control objective, and to have an automaton describing possible behaviors, rather than writing down the complete

correct control solution. The basic case is that of contracts on logical properties i.e., involving only Boolean conditions on states and events.

In the upper part of Figure 11, the contract is itself a program, with three controllable variables, defined in the `with` part, used for enforcing the objectives, and its own equations. One of them, the cache policy (*requirement 1* of Section 1.2), is an example of simple logical property, and is encoded as :

```
pcache = (dense and not active_logger) implies active-cache
```

which can be encoded in primitive Boolean operators as[4]:

```
pcache = not (dense and not active_logger) or active_cache
```

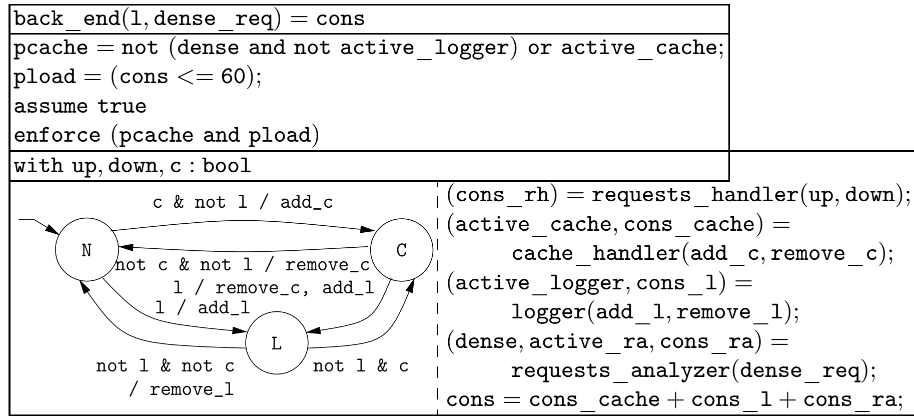


Fig. 11. The BZR node for the *Back end* component.

The control objective then consists in making this predicate invariantly `true` i.e., constraining behaviors to stay invariantly within the states where this predicate is `true`. There is no special assumption made in the environment, on the expected value of inputs. This is simply stated as:

```
assume true
enforce pcache
```

Here, BZR compilation and DCS produce the dynamical, state-dependent constraint on `c` such that the cache will be controlled following the requirement 1.

Reconfiguration policy by contract with weights. A more elaborate form of contract can be used, involving weight functions associated to states. This technique is less powerful than timed or hybrid automata, but also less costly w.r.t. the synthesis algorithms, and it has the benefit of allowing for some expression of quantitative aspects of a system, in terms of static constant values and expressions on them. They are transformed back into logical properties, in the sense that we consider invariants on the respect of bounds.

[4] $a \Rightarrow b \equiv \neg a \vee b$

An example of a weights contract is seen in the upper part of Figure 11. One equation of the contract program encodes the load-related adaptation aspect of the requirements of Section 1.2 (*requirement 2*):

```
pload = (cons ≤ 60)
```

This control objective concerns quantitative weights, but is actually also a logical invariance, so it can be treated as above:

```
assume true
enforce pload
```

Here, BZR compilation and DCS will produce the constraint on `up` and `down` such that the file server will be controlled according to requirement 2.

4.4 Structural control of Fractal components

This shows how a controller for adaptation policies defined by the requirements above, involving mutual exclusion and insertion of reconfiguration tasks, can be obtained with our mixed imperative / declarative method and language. Behaviors are described locally, structurally following the hierachy of components in Fractal. Their composition is made by the compilation. The adaptation policy can be programmed in the automata, but it can also be described declarativly, and the controller is derived automatically and correctly by DCS. Improvements in user-friendly useability could be the definition of a library of predefined patterns for reconfiguration automata and objectives; further in that direction, they could serve to define a domain-specific language, where BZR would be hidden as an internal format [11]. On the side of control techniques, exploiting weights for optimal control, is not yet covered in BZR compilation, but the tools and models exist and could be integrated [19].

5 Execution-level integration

Figure 12 describes the development process of our method. The complete BZR program, comprising both automata and contracts, is first extracted from the Fractal specification. The BZR compiler then produces both sequential C code, and Boolean equations (modelling the BZR program) and synthesis objectives. These equations and objectives allow the DCS tool to produce a constraint on free controllable variables. These constraints are resolved at execution time by a resolver embedded in the C code. This C code is then linked back with the one generated by the Fractal compilation.

5.1 BZR code and compilation

As we saw before, each Fractal component is given a BZR node. The modular compilation of this language allow then to obtain, through the process exposed in Figure 12, one C function "step" for each Fractal component. It handles the lifecycle of its associated component, calling in turn the "step" functions of the active sub-components. The DCS is performed on the synchronous composition,

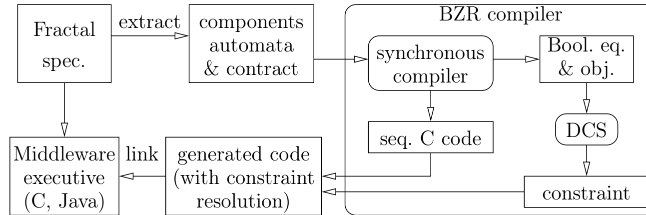


Fig. 12. Development process for BZR, in the case of Fractal.

performed at compilation time, of automata of all sub-nodes. The semantics of the language guarantees the equivalence of the compilation towards executed C code, and the one towards Boolean equations for DCS.

Concerning performance aspects, the synthesis time is clearly the bottleneck of our approach, and the final controller size is, for the same reasons as the synthesis time, exponential in the size of the initial program, but, as we mentioned in Section 3.2, we can handle non-trivial systems in practice. T This controller consists of sequences of conditionals (translation of binary decision diagrams), and its online evaluation is polynomial in the size of the initial program. For the example of this paper, both DCS and on-line resolution of the controller take only few *ms* (standard Pentium, 2.33 GHz).

5.2 Linking executable codes

The step function obtained by BZR compilation handles the lifecycle by:

1. keeping a persistent local state, modelling the life cycle of the actual component, to help the decisions of the synthesized controller,
2. actuating, on each call, on the actual state of the component via the outputs of the node, e.g, by means of calls of reconfiguration scripts, so as to keep this actual state coherent with the internal state of the BZR node.

This second point can be fullfilled, e.g., by associating to outputs of each local node reconfiguration actions programmed in FScript [9]. We give the action associated to the output `add_logger` of the `back_end` node in Figure 13.

5.3 Simulation and typical scenario

Our BZR program can be compiled and executed, or simulated with a chronogram graphical simulator[5] as shown in Figure 14. In the left part, on top, the user interface comprises buttons where the user clicks inputs (`true` or `false` values for the two variables). Outputs of the current step are displayed just below, as well as the current step number since the initial state. The lower part shows the

[5] courtesy of Verimag.

```
action addFS2(root) {
  logger = new("Logger");
  set-name($logger,"logger");
  add($root,$logger);
  bind($root/child::analyzer/interface::logger,
       $logger/interface::logger);
  start($logger);
}
```

Fig. 13. Reconfiguration actions programmed in FScript.

simulation control panel, with a button triggering one step. The right part shows the graphical chronogram display of execution traces, with values at each step.

A typical scenario illustrates the intervention of the controller on the system, so that control objectives are preserved. 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 would be overshot). Hence we go in state (`Dense`, `H1`, `C`). When `l` occurs (step 17), then by requirements 3 and 4, programmed in *Back-End*, 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`).

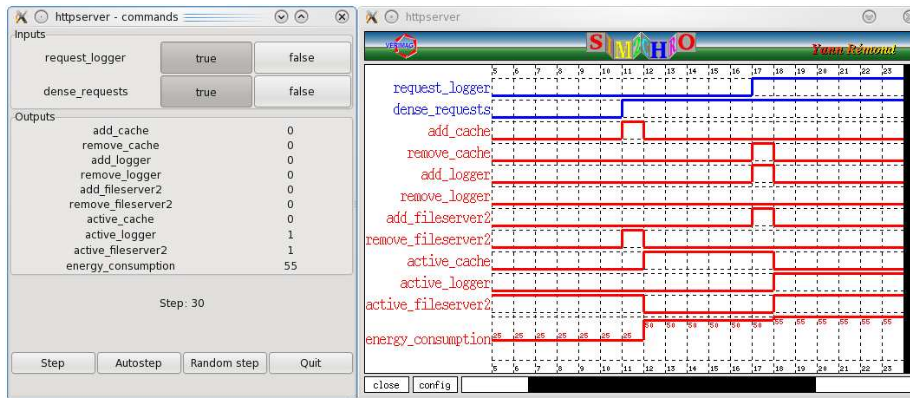


Fig. 14. Simulation.

6 Conclusion

Contributions We propose a technique to design reactive model-based controllers for reconfiguration in the Fractal component-based framework. We consider them

in terms of a discrete controller synthesis problem, solved in the compilation of a programming language ensuring logical safety properties. We obtain discrete control loops for adaptive systems, that can be used e.g., for the safe management of resources: we illustrate the approach with a HTTP server example.

Discussions It can be noted that in the current Fractal, there seems to be no other way to describe reconfiguration than by enumerations the sequences of actions performing them, rather than having a higher-level formalim to describe the reconfiguration (what to reconfigure) independently of their implementation of actions (how to do it). We believe that a Mode Automata formalism [8] might be an interesting, well structured way to define configurations as graphs encapsulated in states of automata, that can be composed hierarchically and in parallel; but this perspective is out of the scope of this paper.

As DCS is a costly algorithmic operation by nature, typically exponential in the number of state variables, it is important to consider techniques to face the combinatorial explosion. It can be noted however that modern symbolic techniques are able to handle in short computation times, in the order of minutes, state spaces of millions of states. Still, scaling up involves specific efforts, and our language is defined in such a way that when a contract node is decomposed into sub-nodes, themselves equipped with sub contracts, DCS is decomposed modularly into local DCS problems, which can be solved independently [10]. At each level, sub-contracts serve as an abstracted model of sub-components in the computation of the local controller. Although not explicitely exploited in this paper, this modularity support is expected to fit particularly well in the hierarchical component-based architectures of Fractal.

Perspectives Ongoing work on the integration of FRACTAL and BZR includes building a more elaborate and refined behavioral model of Fractal components, implementing a concrete integration at the ADL level with the C implementation of Fractal, exploiting hierarchy and modularity in BZR [10], treating concrete case-studies in cooperation with industrial partners in the MIND project, and enriching the models with e.g., reachability or optimization aspects [19].

Perspectives concern ongoing work on the integration of our technique with several targets, other than Fractal [4] object of this paper: FPGA-based reconfigurable architectures, administration loops in a Java virtual machine, and the Orccad control systems design environment.

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.
2. T. Barros, R. Ameur-Boulifa, A. Cansado, L. Henrio, and E. Madelaine. Behavioural models for distributed fractal components. *Annals of Telecommunications*, 64(1):25–43, February 2009.

3. 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.
4. 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.
5. J. Buisson, F. André, and J.-L. Pazat. A framework for dynamic adaptation of parallel components. In *ParCo 2005*, Málaga, Spain, 13-16 September 2005.
6. C. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Kluwer Acad. Publ., 1999.
7. F. Chauvel, O. Barais, I. Borne, and J.-M. Jézéquel. Composition of qualitative adaptation policies. In *23rd IEEE/ACM Int. Conf. on Automated Software Engineering - ASE'08*, L'Aquila, Italy, sep 2008.
8. 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.
9. P.-C. David, T. Ledoux, M. Léger, and T. Coupaye. FPath and FScript: Language support for navigation and reliable reconfiguration of fractal architectures. *Annals of Telecommunications*, 64(1):45–63, February 2009.
10. 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`.
11. G. Delaval and E. Rutten. A domain-specific language for multi-task systems, applying discrete controller synthesis. *J. on Embedded Systems*, 2007. `http://dx.doi.org/10.1155/2007/84192`.
12. Alain Girault and Eric Rutten. Automating the addition of fault tolerance with discrete controller synthesis. *Int. j. on Formal Methods in System Design*, 35(2), october 2009. `http://dx.doi.org/10.1007/s10703-009-0084-y`.
13. D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Meth.*, 5(4), 1996.
14. J. Hellerstein, Y. Diao, S. Parekh, and D. Tilbury. *Feedback Control of Computing Systems*. Wiley-IEEE, 2004.
15. J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1), 2003.
16. S. Krakowiak. *Middleware Architecture with Patterns and Frameworks*. electronic book, 2009. Chap. 10, `http://sardes.inrialpes.fr/~krakowia/MW-Book`.
17. C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar, and J. Hansen. A scalable solution to the multi-resource qos problem. In *RTSS '99: Proc. of the 20th IEEE Real-Time Systems Symposium*, page 315, Washington, DC, USA, 1999.
18. 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.
19. 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.
20. M. Poulhiès, J. Pulou, and J. Sifakis. Buzz: analyzable embedded component-based software. In *Workshop on Component Models for Embedded Systems (COMES)*, Sweden, June 2008.
21. P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. on Control and Optimization*, 25(1):206–230, January 1987.

22. Esterel tech. Scade: model-based development environment dedicated to safety-critical embedded software, 2010. http://www.esterel-technologies.com/.
23. Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, and S. Mahlke. The Theory of Deadlock Avoidance via Discrete Control. In *ACM Symp. on Principles of Programming Languages (POPL '09)*, Jan. 2009.

A The BZR language, and Discrete controller Synthesis

Behavior of BZR programs can be represented by a transition system, as illustrated in the inside box of Figure 15: a transition function $Trans$ takes as inputs X as well as the current state value, and produces the next state value. The latter is memorized by *State* for the next step. The output function *Out* takes the same inputs as $Trans$, and produces the outputs Y. Discrete controller synthesis (DCS) allows to use constructive methods, that ensure, off-line, required properties on the system behavior. DCS is an operation that applies on a transition system (originally uncontrolled), where inputs X are partitioned into uncontrollable (X^u) and controllable variables (X^c). It is applied with a given control objective: a property that has to be enforced by control. In this work, we consider essentially invariance of a subset of the state space. The purpose of DCS is to obtain a controller, which is a constraint on values of controllable variables X^c, function of the current state and the values of uncontrollable inputs X^u, such that all remaining behaviors satisfy the property given as objective. The synthesized controller is maximally permissive, it is therefore *a priori* a relation; it can be transformed into a control function. Figure 15 shows the transition system of the inside box, as yet uncontrolled, composed with the synthesized controller *Ctrlr*, which is fed with uncontrollable inputs X^u and the current state value from *State*, in order to produce the values of controllables X_c which are enforcing the control objective. The transition system then takes $X = X^u \cup X^c$ as input and performs a step.

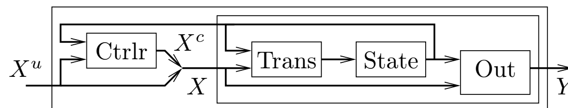


Fig. 15. Controlled transition system

B BZR model of the control of the HTTP server

The node in Figure 16 illustrates the concrete syntax of the BZR language, by showing the code for Figure 11. After the input/output signature, the contract part is based on local Boolean variables, and defines, in the `with` part, the controllable variables that will be used in the body. The body has its own local variables e.g., integers used to compute the cumulated cost for consumption management. It begins with invocations of sub-nodes for the controllers of the sub-components. An equation defines consumption at this level in the component architecture simply as the sum of consumptions underneath. Then an automaton is programmed in the textual syntax: each state is named, and it

```
node back_end (request_logger,dense_requests:bool)
returns (add_cache,remove_cache,add_logger,remove_logger,add_fs2,
        remove_fs2,active_fs2,active_cache,active_logger:bool; cons:int)
contract    var pcache,pload:bool;
  let  pcache = not ((false fby dense_requests)
                  & not (false fby request_logger))  or active_cache;
       pload = cons <= 60;
  tel
  assume true
  enforce (pcache & pload)
  with (up,down,c:bool)
var cons_rh,cons_cache,cons_logger,cons_ra:int; active_ra:bool;
let  (cons_rh,add_fs2,remove_fs2,active_fs2) = requests_handler(up,down);
     (active_cache,cons_cache) = cache_handler(add_cache,remove_cache);
     (active_logger,cons_logger) = logger(add_logger,remove_logger);
     (active_ra,cons_ra) = requests_analyzer();
     cons = cons_rh + cons_cache + cons_logger + cons_ra;
     automaton
        state Nothing do add_cache = not request_logger & not c;
              remove_cache = false;   add_logger = request_logger;
              remove_logger = false;
          until request_logger then Logger | not c then Cache
        state Logger do add_cache = not request_logger & not c;
              remove_cache = false;  add_logger = false;
              remove_logger = not request_logger;
          until not request_logger & c then Nothing
              | not request_logger & not c then Cache
        state Cache do add_cache = false;
              remove_cache = request_logger or c;
              add_logger = request_logger;  remove_logger = false;
          until request_logger then Logger | c then Nothing
     end     tel
```

Fig. 16. Concrete BZR code for the controller of the *BackEnd* component.

is associated with equations executed at each step as long as the control is in the state. The `until` part specifies the transition conditions and targets: when the condition evaluates to value `true`, the `then` part gives the name of the state where the control will be from next step; several transitions going out of a state are separated by `|`.