A Programming Language for Adaptation Control: Case Study

Soufyane Aboubekr, Gwenaël Delaval, Éric Rutten
INRIA Grenoble
{Soufyane.Aboubekr, Gwenael.Delaval, Eric.Rutten}@inria.fr

ABSTRACT

We illustrate an approach for the safe design of adaptive embedded systems. It applies the BZR programming language, featuring a special new contract mechanism: its compilation involves automatical discrete controller synthesis. The contribution of this paper is to illustrate how it can be used to enforce the correct adaptation control of the application, meeting execution constraints, with the case study of a video module of a multimedia cellular phone.

1. ADAPTIVE COMPUTATION AND ITS CONTROL

The management of dynamical adaptivity can be considered as a control loop, on continuous or discrete criteria. It is illustrated in Figure 1 (left): on the basis of monitor information and of an internal representation of the system, a control component enforces the adaptation policy or strategy, by taking decisions w.r.t. the adaptation or reconfiguration actions to be executed, forming a closed control loop.

Embedded systems are also by nature safety-critical, and must be statically checkable for guarantees of predictability. The design of control loops with known behavior and properties is the classical object of control theory. Applications of continuous control theory to computing systems have been explored quite broadly [3], concerning typically quantitative problems (e.g. throughput, delays, load), based on differential equations. In contrast, qualitative or 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 [5], with models such as Petri nets or finite state automata. The reactive approach to embedded systems precisely proposes languages, tools and methods relying on such models, such as StateCharts. Formal techniques have been developed for their specification, verification, and implementation targeted at a variety of execution platforms.

The synchronous approach [1] offers all this support, and particularly a Discrete Controller Synthesis (DCS) technique and its tool, for the design of discrete closed-loop controllers [4]. Amongst other applications, the case a video processing mode controller has been treated manually [6] In our new approach, DCS is embedded in the compilation of a user-friendly programming language called BZR [2]. Models of the possible behaviors of the managed system are specified in terms of hierarchical parallel 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 (right). The contribution of this paper is to illustrate the new BZR language with a case study of a video processing mode controller.

2. CASE STUDY: A MULTIMEDIA APPLICATION

We consider a multimedia mobile device, such as a modern cellular phone, with multiple functionalities: camera, games, mp3 music, video, or even telephone. We focus on the video part, from source to display. This kind of image data processing can also be found in multimedia systems such as high-definition digital television (HDTV), biometric data processing, sonar and radar signal processing.

Functionalities available. Interface buttons (*up* and *down*) allow the user to select amongst various modes. The displayed videos *source* can be either on-line from the network, or from the camera, or from the local memory. There are different *style* modes: Black & White; Negative, a tonal inversion style; Sepia, a dark brown-grey color style; or Normal. The *resolution* of the video can be set to High, Medium or Low. Finally, the *color* can be in either of Color or Monochrome options. Besides user commands, the video

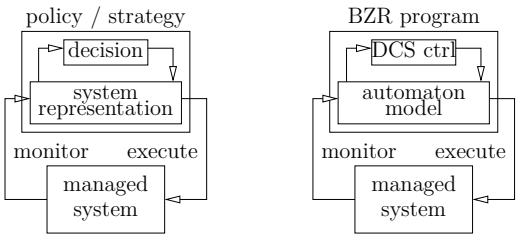


Figure 1: Adaptive system, and BZR program.

display modes are controlled by the *adaptive system* according to the status of computing resources, of the energy level, and of the available communication level.

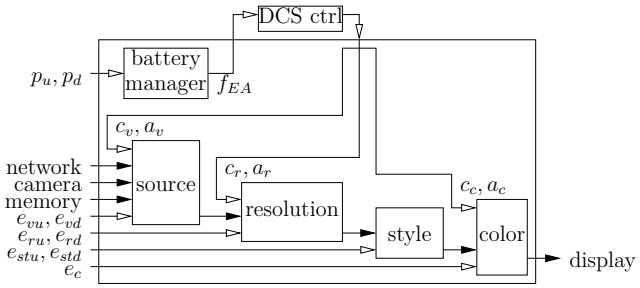


Figure 2: Components of video case study.

Our case study multimedia processing module can be structured as illustrated in Fig. 2, where black-pointed arrows indicate the flow of the video images. It is composed of the following subcomponents: *battery manager* indicates the energy level f_{EA} according to inputs p_u or p_d from the battery (power going up ou down resp.). *source* chooses video source according to user requests e_{vu}, e_{vd} and *controller*'s permission c_v, a_v. *resolution* computes resolution according to user requests e_{ru}, e_{rd} and *controller*'s acceptance c_r, a_r. *color* is similar to *resolution*, as well as *style* which does not need control. On top of this, the *controller* validates mode change requests according to current component states (not shown in the Figure for clarity) and according to the available resources (here: battery). This specific component is automatically obtained by DCS.

Consumption and Quality of Service. When embedded in mobile devices, such data processing systems have resource constraints: memory capacity, processor load, energy, etc. For an efficient resource management, adaptivity and reconfigurability mechanisms are needed so as to enable a flexible system execution w.r.t. environment and platform constraints for e.g., power-aware management, fault-tolerance and recovery. Possible behaviors involving these characteristics are, e.g., that the consumption of a resource must respect the bounds defined by its capacity. Therefore, if a new functionality is executed, then the other tasks that are already running should switch to lower consumption modes, and possibly reduce their quality as well. Or, if battery level goes down, the control should switch task modes so that the lower energy capacity is respected.

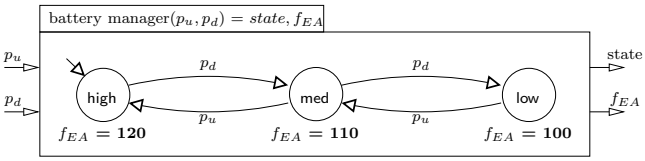
This video processing module has different configuration modes, where components execute different algorithms for image display. Hence, each mode is associated with non-functional properties, which must be satisfied in order to display images at a good quality level. In our example, the modes defined in the components are characterized by quantitative attributes representing the following non-functional properties: energy consumption (E), communication quality required (CQ), computing resource consumption (CR) and memory consumption (M). These non-functional requirements are to be understood as instantaneous values of quantitative resources that may vary from one system mode to another. The values associated locally with the modes are combined additionally when components are composed in parallel, so as to obtain global costs for the whole system from the local costs of its components.

In the next section, we describe the BZR specification of this case study (behaviors as well as non-funtional aspects), where the adaptation controller is obtained automatically.

3. BZR AUTOMATON MODEL

For clarity reason, the BZR specification is described here with an ad-hoc graphical syntax. Some extracts of the whole example in concrete syntax can be found in appendix A.

We propose a a safe design approach, that guarantees that the designed multimedia functionality module actually meets its specification requirements, with easy specification and modifiablity. As shown in Figure 1 (right), the system model is programmed in terms of reactive automata, describing possible behaviors of the systems, where states represent each configuration, and transitions indicate which switches are *a priori* allowed. The decision part of the adaptation controller will be specified in BZR in a declarative way, in terms of contracts: invariance properties that should be satisfied by the sequences of reconfigurations, and it will be produced automatically by DCS.




```
node energy_status (down, up:bool)
            returns (high, medium, low:bool;
                     energy_quantity:int)
let
  automaton
    state High
      do
        (high,medium,low) = (true,false,false);
        energy_quantity = 120;
      until down then Medium
    state Medium
      do
        (high,medium,low) = (false,true,false);
        energy_quantity = 110;
      until up then High
          | down then Low
    state Low
      do
        (high,medium,low) = (false,false,true);
        energy_quantity = 100;
      until up then Medium
  end
tel
```

Figure 3: Energy resource model

Basic behaviors. They can be represented as nodes, as usual in the synchronous languages [1]. A simple example is the battery manager component seen in Figure 2, now detailed in Figure 3: it shows the interface, with inputs and outputs, and the body, consisting of a mode automaton

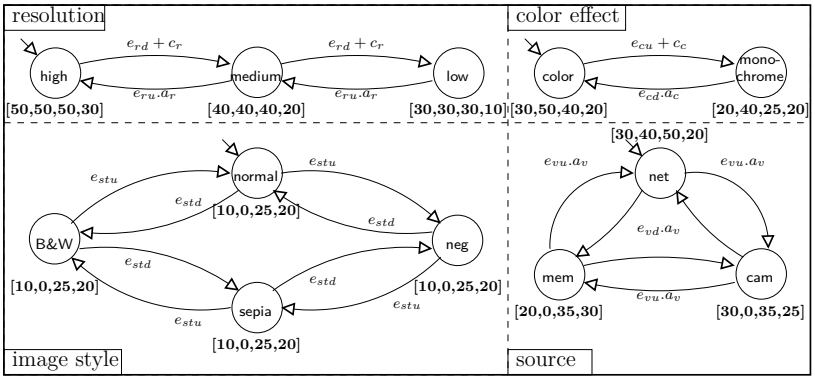


Figure 4: Mode automata for *VideoSource*, *ColorEffect*, *ImageStyle* and *Resolution* components.

with three states. The intial state is high. 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 each state, an equation defines the output flow f_{EA}: it is an integer value characterizing the available power, which will be used as an upper bound, as part of the policy to be enforced by the control of the adaptive system.

Composed behaviors. Such nodes and automata can be composed in parallel, according to the synchronous composition [1]. Fig. 4 shows the composition of the automata of the four video processing components.

There is an assumption on interface buttons (*up* and *down*): they are managed so that they are not present at the same time, hence corresponding inputs (subscribed with u and d resp.) are not true at the same time. Therefore, in particular, in the automaton of *imagestyle*, there are no ambiguities and behavior is deterministic; also, there are no diagonal transitions, taking into account that buttons are used to navigate in a circular list of modes.

In the composed automaton, the global behavior 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.

A tuple [E, CQ, CR, M] is associated with each mode, denoted as a state, characterizing the quantitative attributes associated with the resources defined before. The global characterization of the non-functional aspects is simply defined by the sum of the local ones for each term.

When assembled according to Fig. 2, these specifications define all possible behaviors of the system, before any controller is defined. The correct adaptation control will be derived from this model, using the BZR compiler encapsulating its formalization as a transition system and DCS.

Equational model and DCS. A transition system such as specified before, can be seen equivalently as a sequential transition function, as in classical Boolean circuits, illustrated e.g., in the *body* box in the lower part of Fig. 5. A transition function $Trans$ takes the current value of the $State$ and the inputs, and computes the new value of the $State$; another function Out computes outputs. Logical properties of such automata typically concern reachability of states, or invariance of subsets of states, or required or forbidden sequences of transitions.

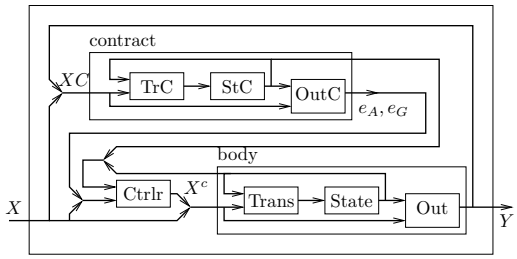


Figure 5: BZR contract node as DCS problem

On the other hand, DCS consists of considering on the one hand, the set of possible behaviors of a discrete event system, where inputs are partitioned into uncontrollables (X) and controllable (X^c), as illustrated in the lower part of Fig. 5. On the other hand, it requires the specification of a control objective: a property, typically an invariance. DCS consists of the computation of the necessary constraint $Ctrlr$ on controllable events X^c, in function of the current state and all possible uncontrollable inputs X, so that the objective properties are satisfied by the resulting controlled system. These computed constraints yield a *controller* which defines, together with the initial system, a *controlled system*, satisfying the synthesis objectives. This can be formulated as:

Whatever the uncontrollable inputs sequences, the controlled behavior satisfies the objectives.

We encapsulate these notions in the compilation of the BZR language [2], thereby making them more user-friendly.

4. BZR CONTRACTS AND ADAPTATION POLICY

BZR contracts and corresponding DCS problem. As illustrated in Figure 6, we associate a *contract* to a node: it is itself a program, with its internal state, and with 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

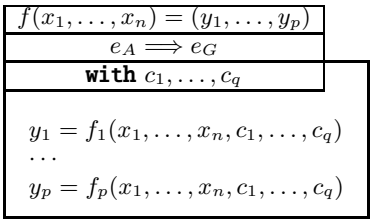


Figure 6: BZR contract node graphical syntax

ensuring this objective. This contract means that the node will be controlled, i.e., that values will be given to $c_1, \ldots, c_q$ such that, given any input trace yielding e_A, the output trace will yield the true value for e_G. This will be done by DCS. One can remark that the contract can itself feature a program, e.g., automata, observing traces, and defining states.

Without giving details [2] out of the scope of this case study, we compile such a BZR contract node into a DCS problem as in Figure 5, where the contract part has its inputs XC coming from the node's input X and the body's outputs Y, its own transition function TrC and its own state StC, and its output function $OutC$ computing e_A, e_C. Assuming e_A produced by the contract program, , we will obtain a controller $Ctrlr$ 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 $c_1, ...c_q$. As shown in Figure 5, the controller takes the state of the body and the state of the contract, and the inputs of the node X and the outputs of the contracts e_A, e_G, and it computes the controllables X_c such that the resulting behavior satisfies the objective.

Adaptation policy in the case study. We will specify it by a BZR contract of a simplified form, as in our case there is no assumption on the environment i.e., e_A is **true**, and the policy can be decomposed in different parts i.e., e_G is a conjunction & of terms as follows.

A first example of a state property is the *exclusivity of two modes* of two components. For example, in order to avoid waste of resources, it can be useful to specify that the modes B&W (*ImageStyle* component) and Color (*ColorEffect* component) are never active at the same instant. This is noted in the code of the contract simply by a flow:

```
excl_modes = not (mode_color & style_bw);
```

Another example involves cost functions, e.g. for memory footprint, there is a *bound on cost* defined by the size of the memory e.g., 90. We have a BZR equation defining the value of the global consumption in a flow by that of the components: `m_global = m1+m2+m3+m4;` hence we want to enforce ($m\_global \leq 90$):

```
bounded_memory = (m_global <= 90);
```

On some of the resources, we have a *bound varying in time*, as is the case for the available power. We model this as seen in Fig. 3. We have the value of the currently available power, which we use as a bound for the energy consumption: we want to enforce ($f_E \leq f_{EA}$):

```
bounded_energy = (e_global <= e_available);
```

It is also interesting to define *conditioned objectives*, e.g., according to the battery status, which will lead to adaptive strategies. An example of such adaptive strategy is to forbid costly modes in medium or low battery states, so as to limit the energy consumption in time. We identify two modes, high resolution and color mode, which will be enforced exclusive when not in high battery state: we want $\Big(\big(\mathrm{High_{Res}} \wedge \mathrm{Color}\big) \Rightarrow \mathrm{High_{Energy}}\Big)$ which is written:

```
cond_obj = (not (resol_high & mode_color)
            or energy_high);
```

In summary the contract says simply, in textual syntax:

```
assume true
enforce excl_modes & cond_obj
        & bounded_memory
        & bounded_energy
with (c_r, a_r, c_c, a_c, a_v : bool)
```

where in the **with** part relevant inputs of the different components are given as controllable variables.

5. CONCLUSION

This paper describes a case study illustrating the use of a new contract mechanism, and its compilation involving DCS. It is implemented in the BZR compiler [2], with generation of Java and C executable code. It encapsulates in a user-friendly way the formal DCS operations, which otherwise had to be done manually [6]. Different adaptation policies can be obtained *automatically* by changing the objectives, hence improving separation of concerns and reuse of models. The code can be plugged into a system as of Fig. 1, thereby providing for safe adaptation control.

6. REFERENCES

[1] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proc. of the IEEE*, 91(1):64–83, jan. 2003.

[2] G. Delaval, H. Marchand, and E. Rutten. Contracts for modular discrete controller synthesis. In *Draft available from the authors*, 2009. `http://sardes.inrialpes.fr/~rutten/docs/BZR-draft.pdf`.

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

[4] 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):325–346, October 2000.

[5] Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, and S. A. Mahlke. The theory of deadlock avoidance via discrete control. In *Proc. of the 36th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL 2009,* Savannah, GA, USA, January 21-23, pages 252–263, 2009.

[6] H. Yu, G. Delaval, A. Gamatié, and E. Rutten. A case study on controller synthesis for data-intensive embedded systems. In *Proc. of the 6th IEEE Int. Conf. on Embedded Software and Systems, ICESS'09,* HangZhou, Zhejiang, China, May 25–27, 2009, pages 75–82, 2009.

APPENDIX

A. EXAMPLE IN CONCRETE SYNTAX

```
node color_effect (ctr_color,aut_color:bool;
                   up,down:bool;
                   i:rgb_pixel)
          returns (o:rgb_pixel;
                   mode_color:bool;
                   e,cq,cr,m:int)
let
  automaton
    state Color
      do
        o = color(i);
        mode_color = true;
        (e,cq,cr,m) = (30,50,40,20);
      until up or ctr_color then Monochrome
    state Monochrome
      do
        o = monochrome(i);
        mode_color = false;
        (e,cq,cr,m) = (20,40,25,20);
      until down & aut_color then Color
  end
tel
⋮
node cell_phone (event_energy_up,
                 event_energy_down,
                 event_comm_quality_up,
                 event_comm_quality_down,
                 event_computing_resource_up,
                 event_computing_resource_down,
                 event_color_up,
                 event_color_down,
                 event_image_style_up,
                 event_image_style_down,
                 event_video_source_up,
                 event_video_source_down,
                 event_resolution_up,
                 event_resolution_down:bool;
                 camera, local, online:rgb_pixel)
        returns (mode_color,
                 style_bw,
                 resol_high,
                 mode_color,
                 energy_high:bool;
                 m_global:int;
                 e_global:int;
                 energy_quantity:int;
                 o:rgb_pixel)
  contract
    var excl_modes,
        bounded_memory,
        bounded_energy,
        cond_obj:bool;
  let
    excl_modes = not (mode_color & style_bw);
    bounded_memory = (m_global <= 90);
    bounded_energy = (e_global <= e_available);
    cond_obj = (not (resol_high & mode_color)
                or energy_high);
  tel
  assume true
  enforce excl_modes & cond_obj
          & bounded_memory
          & bounded_energy
  with (ctr_resolution,aut_resolution,
        ctr_color,aut_color,
        aut_video_source:bool)

var ...;
let
    (energy_high, energy_medium, energy_low,
     e_available) =
       inlined energy_status(event_energy_down,
                             event_energy_up);

    (ov, e1,cq1,cr1,m1) =
      inlined video_source(ctr_video_source,
                           event_video_source_up,
                           event_video_source_down,
                           camera,online,local);

    (ores,
     resol_high, resol_medium, resol_low,
     e2,cq2,cr2,m2) =
       inlined resolution(ctr_resolution,
                          aut_resolution,
                          event_resolution_up,
                          event_resolution_down,
                          ov);

    (ois,
     style_normal,style_negative,
     style_sepia,style_bw,
     e3,cq3,cr3,m3) =
       inlined image_style(event_image_style_up,
                           event_image_style_down,
                           ores);

    (o,
     mode_color,
     e4,cq4,cr4,m4) =
       inlined color_effect(ctr_color,aut_color,
                            event_color_up,
                            event_color_down,
                            ois);

    e_global = e1+e2+e3+e4;
    cq_global = cq1+cq2+cq3+cq4;
    cr_global = cr1+cr2+cr3+cr4;
    m_global = m1+m2+m3+m4;
tel
```