INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Synchronous Control of Reconfiguration in Fractal Component-based Systems – a Case Study

Tayeb Bouhadiba — Quentin Sabah — Gwenaël Delaval — Eric Rutten

N° 7631

May 2011

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--7631--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE - RHÔNE-ALPES

Synchronous Control of Reconfiguration in Fractal Component-based Systems – a Case Study

Tayeb Bouhadiba , Quentin Sabah , Gwenaël Delaval , Eric Rutten

Theme :
Équipe-Projet SARDES

Rapport de recherche n° 7631 — May 2011 — 32 pages

Abstract: In the context of component-based embedded systems, the management of dynamic reconfiguration in adaptive systems is an increasingly important feature. The Fractal component-based framework, and its industrial instantiation MIND, provide for support for control operations in the lifecycle of components. Nevertheless, the use of complex and integrated architectures make the management of this reconfiguration operations difficult to handle by programmers. To address this issue, we propose to use synchronous languages, which are a complete approach to the design of reactive systems, based on behavior models in the form of transition systems. Furthermore, the design of closed-loop reactive managers of reconfigurations can benefit from formal tools like Discrete Controller Synthesis. In this paper we describe an approach to concretely integrate synchronous reconfiguration managers in Fractal component-based systems. We describe how to model the state space of the control problem, and how to specify the control objectives. We describe the implementation of the resulting manager with the Fractal/Cecilia programming environment, taking advantage of the Comete distributed middleware. We illustrate and validate it with the case study of the Comanche HTTP server on a multi-core execution platform.

Key-words: Component-based systems, synchronous programming, reconfigurable systems, discrete controller synthesis.

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Synchronous Control of Reconfiguration in Fractal Component-based Systems – a Case Study

Résumé : Dans le contexte des composants pour systèmes embarqués, la gestion de la reconfiguration dynamique devient de plus en plus importante. Le modèle à composants Fractal et son implémentation MIND, fournissent des moyens de contrôle de cycle de vie des composants ainsi que des moyen pour le contrôle des architectures. L'utilisation des architectures intégrées de plus en plus complexes, rend la gestion des opérations de reconfiguration difficile à maintenir par le programmeur. Cette gestion devient plus complexe quand des propriétés globales sur le systèmes doivent être assurées.

Nous proposons d'utiliser des langages synchrones réactifs, reposant sur des modèles comportementaux sous la forme de systèmes de transitions. De plus, notre approches, qui produit un manager synchrone pour la reconfiguration dynamique profite des techniques formelles comme la Synthèse de Contrôleurs Discrets.

Ce papier décrit l'intégration concrète d'un manager synchrone pour la reconfiguration de systèmes-à-composants Fractal. Nous détaillerons notre approche en commençant par la partie modélisation du problème de contrôle sous forme d'espace d'états de configurations, ainsi que la description des propriétés de contrôle. Ensuite, nous aborderons la partie implémentation du manager résultant en Fractal/Cecilia et son intégration dans des applications Fractal distribuées en utilisant le middleware Comete. Nous validerons notre approche au moyen d'un cas d'étude sur le serveur HTTP Comanche sur une plateforme d'exécution multicoeurs.

Mots-clés : Systèmes à base de composants, Programmation Synchrone, Systèmes reconfigurables, Synthèse de contrôleurs discrets.

Contents

1 Introduction

In the context of component-based embedded systems, the management of reconfiguration in adaptive systems is an increasingly important feature. The Fractal component-based framework, and its industrial instantiation MIND, provide for support for control operations in the lifecycle of components. Nevertheless, the use of complex and integrated architectures make the management of this reconfiguration operations difficult to handle by programmers. To address this issue, we propose to use synchronous languages, which are a complete approach to the design of reactive systems, based on behavior models in the form of transition systems. Furthermore, the design of closed-loop reactive controllers of reconfigurations can benefit from formal tools like Discrete Controller Synthesis (DCS).

Using DCS, integrated in a programming language, provides designers for support in the correct design of controllers. This method is different from the usual method of first programming and then verifying. It involves the automated generation of part of the control logic of a system. Discrete control has until now been applied to computing systems only very rarely [20]. An important challenge is to integrate this formal reactive systems design in actual, practical operating systems. An open issue is the identification and correct use of the practical sensors and monitors providing for reliable and significant information; the control points and actuators available in the API of the OS, enabling enforcement of a management policy; the firing conditions for the transitions of the automata.

In this paper we describe an approach to concretely integrate synchronous reconfiguration controllers in Fractal component-based systems. Our contribution is: (i) a synchronous model of the behavior of the reconfigurable components, in the form of the state space of the control problem, and the specification of the control objectives (Section 4); (ii) a component-based architecture for the implementation of the resulting controller with the Fractal/Cecilia programming environment, taking advantage of the Comete middleware. (Sections 5 and 6). We validate it by the case study of the Comanche HTTP server deployed on a multi-core execution platform.

2 Background

2.1 The Fractal Component Model

We introduce Fractal [3][9], a hierarchical and reflective component model and Cecilia [1], a component-base software engineering framework providing a C implementation of this model. Fractal defines components as entities encompassing behaviours and data. A component can be dismentled in two parts: a *membrane* and a *content*. The content is either a set of operations or a finite number of sub-components, which are under the control of the enclosing membrane. Components can be nested at an arbitrary level in a recursive fashion.

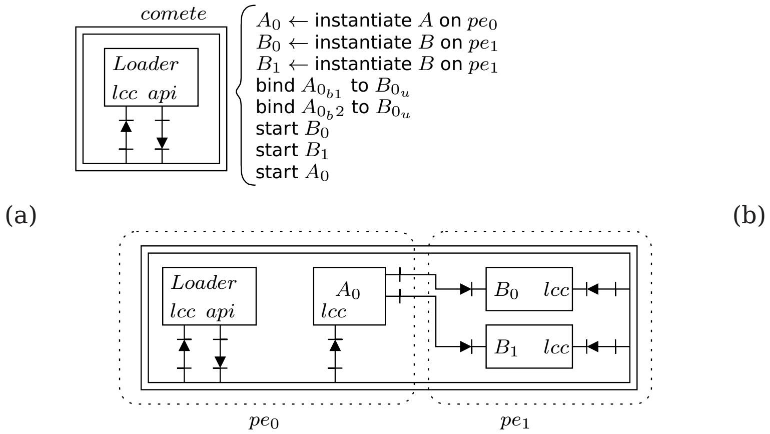

comete
Loader
lcc api
$A_0 \leftarrow$ instantiate A on pe_0
$B_0 \leftarrow$ instantiate B on pe_1
$B_1 \leftarrow$ instantiate B on pe_1
bind $A_{0_{b1}}$ to B_{0_u}
bind $A_{0_b 2}$ to B_{0_u}
start B_0
start B_1
start A_0
(a)
(b)
Loader
lcc api
A_0
lcc
B_0 *lcc*
B_1 *lcc*
pe_0
pe_1

Figure 1: The deployment process. (a) the *loader* is in charge of building the initial configuration (b)

Components interact with their environment through *interfaces*. Interfaces are typed collections of operations. They can be of two sorts: *client interfaces* emit operation invocations, *server interfaces* receive operation invocations. One-way operation invocations may carry arguments, two-way operations consist of an invocation followed by the return of a result. Components can expose several interfaces. Client and server interfaces are connected through explicit *bindings*. *Functional interfaces* are access points to content operations, while *Controller interfaces* define membrane operations. The membrane embodies the control behaviour associated with a particular component.

The Fractal model defines a set of optional controller interfaces to adress minimal requirements in terms of introspection, composition and life-cycle. Among others are:

- *Life-Cycle Controller*: controls component's behavioural phases such as starting and stopping.
- *Binding Controller*: establish/break bindings between component's interfaces and its environment.

The Cecilia framework is a coherent toolchain to design, compile and deploy component-based applications. It allows the description of hierar-

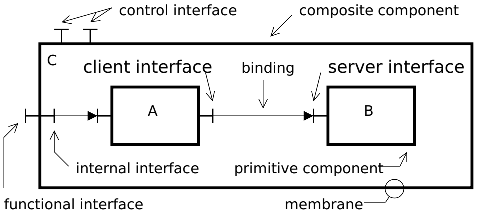


Figure 2: Fractal vocabulary.

chical Fractal architectures using an xml ADL, and the implementation of the primitive components using the C langage. By automaticaly generating the controllers related glue-code, the toolchain allow the developer to focus on the implementation of the primitives' content operations i.e. the *functional interfaces*' methods in figure 2. From an application definition and its implementation, the toolchain generate a standalone executable or a set of independent component binaries. The Comete middleware described in the next section relies on these independent bricks to dynamically load and bind instances of components to build arbitrary architectures.

2.2 Comete

Comete [2] is a minimal middleware and run-time layer engineered by STMicroelectronics to dynamicaly deploy, run and reconfigure Cecilia components over distributed platforms. Comete is providing a *distributed event-driven architecture*. Applications deployed using Comete can take advantage of the high-level abstraction of this platform to communicate *asynchronous messages* between components. The middleware models a distributed platform as a set of *processing elements* and handles communication between them so that application developpers don't have to know about the underlying communication channels and protocols.

The first step of any application deployment using Comete lies in the instantiation of a *loader* component (Figure 1(a)) bound to the Comete API interface (this interface is further detailed in Listing 4). This mandatory component is in charge of deploying the application by invoking Comete operations such as instantiations and bindings to build the initial configuration (Figure 1(b)). Once the deployment process done, the loader can remotely manipulate *LifeCycleController (lcc)* interface of any instantiated component through the *start/stop* methods to initiate the execution. Then, the application is free to diverge from its initial architecture by successive reconfigurations. The initial deployment is like a reconfiguration from a single primitive component to a potentially more complex architecture.

Each processing element executes message handlers related to components it embodies. The runtime layer consists of a task queue associated with a FIFO scheduler (Figure 3). The scheduler is non preemptive. Every message directed to a given component will be handled as task on the respective processing element. The execution model guaranties that: a) At any given time only one method is executing on a processing element. b) Components deployed on different processing element may execute methods concurrently.

At its lowest level, the runtime layer is dealing with platform heterogeneity in terms of operating systems, hardware platforms and communication protocols. This heterogeneity is then abstracted by the middleware layer. Internally, a remote binding is handled by a couple of *stub/skeleton* components loaded respectively on the client's and server's processing element. On the client side, the stub transparently intercepts and serializes emitted messages. Data are then transmitted through a platform

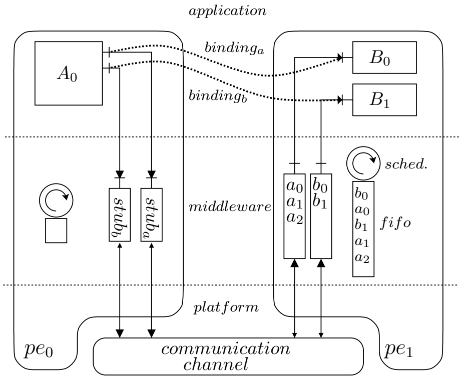


Figure 3: A simple application spanning over two processing elements. Internal implementation of applicative bindings (stroked) involves dedicated communication components in the middleware layer to support platform channels.

channel to the server side. The skeleton reads serialized message data, operates the inverse process and pushes a handler task in the local runtime queue. The message arguments are stored until the associated task is scheduled. Thanks to a generic, scalable, component-based architecture, Comete is targeting a wide range of platforms, from embedded System-On-Chip to distributed computers. The middleware is built over an extensible library of components providing support for various processing elements and communication channels. Noticeable specializations of Comete are: the STm8010(Traviata) board including three ST200 cores, the xStream many-cores streaming prototype, posix-compliant operating systems easing deployment over multi-threaded hardware, computers distributed over a TCP/IP network.

2.3 Heptagon and BZR

2.3.1 Heptagon language

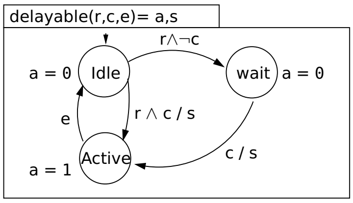
In this work, we use the language Heptagon/BZR [13][1]. The Heptagon language allows to describe reactive systems by means of generalized Moore machines, i.e., mixed synchronous dataflow equations and automata [11], with parallel and hierarchical composition. The basic behavior is that at each reaction step, values in the input flows are used in order to compute the values in the output flows for that step. Inside the nodes, this is expressed as a set of declarations, which takes the form of equations defining, for each output, the values that the flow takes, in terms of an expression on other flows' instantaneous values, possibly using values computed in preceding steps (also known as state values).

Figure 4 shows a small program in this language. It describes the control of a task, which can either be idle or active. When it is idle, i.e., in the initial

[1] Available at bzr.inria.fr

Idle state, then the occurrence of the input `r` *requests* the launch of the task. Another input `c` (which will be controlled further by the synthesized controller) can either allow the activation, or temporary block the request and make the automaton go to a waiting state. When active, the task can be ended with the input `e`. This `delayable` node has two outputs, `a` featuring the instantaneous activity of the task, and `s` being emitted on the instant when it becomes active:

step #	1	2	3	4	5	6	7	...
r	0	1	0	0	0	0	0	...
c	0	0	1	0	0	0	0	...
e	0	0	0	0	0	1	0	...
a	0	0	0	1	1	1	0	...
s	0	0	1	0	0	0	0	...

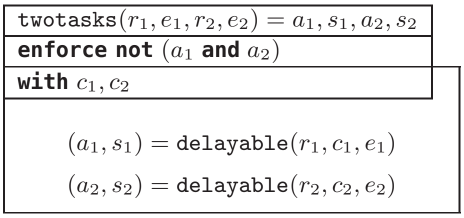


```
node delayable(r,c,e: bool) returns (a,s: bool)
  let
    automaton
      state Idle
        do a = false ; s = r and c
        until r and c then Active
            | r and not c then Wait
      state Wait
        do a = false ; s = c
        until c then Active
      state Active
        do a = true ; s = false
        until e then Idle
    end
  tel
```

Figure 4: Delayable task (graphical/textual syntax).

2.3.2 BZR and controller synthesis

BZR is an extension of Heptagon, allowing its compilation to involve *discrete controller synthesis* (DCS) using the DCS tool Sigali [17]. DCS allows to compute automatically a controller, i.e., a function which will act on the initial program so as to enforce a given temporal property. Concretely, the BZR language allows the declaration of *controllable variables*, which are not defined by the programmer. These free variables can be used in the program so as to let some choices undecided (e.g., choice between several transitions). The controller, computed by DCS, is then able to avoid undesired states of the application by setting and updating appropriate values for these variables at runtime.



twotasks$(r_1, e_1, r_2, e_2) = a_1, s_1, a_2, s_2$

enforce not $(a_1$ **and** $a_2)$

with c_1, c_2

$(a_1, s_1) =$ delayable(r_1, c_1, e_1)

$(a_2, s_2) =$ delayable(r_2, c_2, e_2)

Figure 5: Mutual exclusion in BZR.

Figure 5 shows an example of use of these controllable variables. It consists in two instances of the `delayable` node, as in Figure 4. They run in parallel, defined by synchronous composition: one global step corresponds to one local step for every equation, i.e., here, for every instance of the automaton in the `delayable` node. Then, the `twotasks` node so defined is given a *contract* composed of two parts: the **`with`** part allowing the declaration of controllable variables (c_1 and c_2), and the **`enforce`** part allowing the programmer to assert the property to be enforced by DCS, using the controllable variables. Here, we want to ensure that the two tasks running in parallel won't be both active at the same time. Thus, c_1 and c_2 will be used by the computed controller to block some requests, leading automata of tasks to the waiting state whenever the other task is active.

2.3.3 Heptagon/BZR compilation

The compilation of a Heptagon/BZR program produces sequential code in a target general programming language (C, Java or Caml). This code takes the form of two functions (or methods), named *reset* and *step*. *reset* initializes the internal state of the program. The *step* function is evaluated at each logical instant to compute output values from an input vector and the current state, possibly updated. A typical way of using these functions is to enclose this *step* call in an infinite loop:

```
current_state ← reset() // state initialization
for each step do
   inputs ← gather current events
   outputs, next_state ← step(inputs, current_state)
   handle outputs
   current_state ← next_state
```

Eventually, such infinite loop will not be as clearly stated, but hidden within, e.g., events managers, threads, interrupts, depending on the application context. In our context, we will use the C generated code. The API, for a node `f` with inputs $\mathtt{x}_1, \ldots, \mathtt{x}_n$ (typed $\mathtt{t}_i$) and outputs $\mathtt{y}_1, \ldots, \mathtt{y}_p$ (typed $\mathtt{t}'_i$) is given below. The additional input `mem` is a pointer towards the internal state, to be used and updated. The result of the `f_step` call is a structure in which are placed the outputs values.

```
void f_reset(f_mem* mem);
f_res f_step(t_1 x_1, ...,t_n x_n, f_mem* mem);
```

2.4 System/manager Interaction

Our approach applies to systems supporting dynamic reconfiguration and providing some events describing their observable states. It relies on the modeling of the system (and possibly the environment) by means of Heptagon automata and describing the control objectives. The BZR compilation tool-chain compiles the automata and the control objectives into a synchronous program which will be referred to as manager in the sequel. The manager encapsulates a model of the system and a controller to enforce the objectives.

Figure 6 illustrates the use of a manager (i.e., synchronous program) to manage the reconfiguration of a system. The manager receives some input events from the system and the environment. According to these inputs and the current state of the model, the controller may update the state of the model. Internally, this reduces to fill the controllable variables with the appropriate computed values. The computation of the new state of the model corresponds to a step of the manager and may fire some commands. The commands are meant to reconfigure the managed system in order for this latter to be coherent with its model.

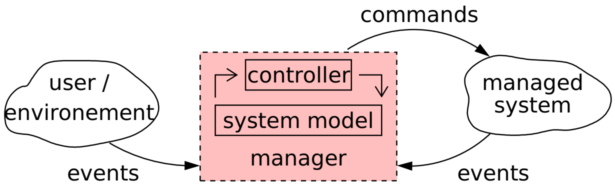


Figure 6: Application control by a manager

3 The Comanche Http Server

This section describes an example of a reconfigurable component-based software application written in Fractal together with the execution platform on which the application will be run. The complete system will be used as a case-study in order to introduce our approach for the management of reconfigurable systems in Section 4.

3.1 Components architecture

Figure 7 describes the architecture of a HTTP server written in Fractal. It is a variant of the Comanche server used as an example in tutorials[2]. Incoming requests are received by the `frontend` component, which transmits them

[2] http://fractal.ow2.org/tutorial

to the `analyzer` component. The latter forward well-formed requests to the `dispatcher` which queries `fileserver`$_1$ or `fileserver`$_2$ to solve the requests. The `analyzer` component can also send requests to the `logger` to keep track of HTTP requests.

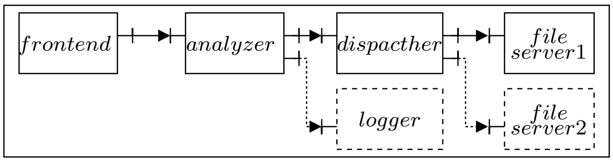


Figure 7: Comanche architecture.

The required components for this application to work are `frontend`, `analyzer`, `dispatcher` and `fileserver`$_1$. A first available degree of dynamical reconfiguration lies in `fileserver`$_2$ and `logger`. As illustrated by the dashed lines, `fileserver`$_2$ and `logger` may be activated (resp., deactivated) and connected to (resp., disconnected from) the rest of the components.

3.2 Execution architecture

A second degree of reconfiguration concerns component mapping over processing elements. For our experiments, we used a server equipped with two Intel Xeon dual-core processors, each one running at 1.86GHz clock frequency. The four available cores enable the execution of four tasks in parallel. In the sequel, we will refer to each core as a *processing element* (pe$_0$, ..., pe$_3$).

Figure 8 describes the initial deployment of Comanche components using Comete middleware (Section 2.2) on the execution platform. Each component is associated with a processing element, on which it will execute in order to handle messages in its associated FIFO. Component bindings are implemented by the Comete middleware as asynchronous communication channels. For the sake of clarity of the figure, the bindings are made implicit. Notice that the manager and Comete loader also execute on the same platform.

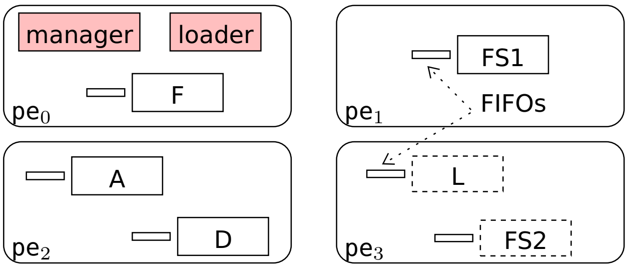


Figure 8: Architecture of the controlled application

3.3 Renconfiguration policy

The control objectives we want to enforce are related to distinct aspects of the Comanche application and the execution platform. We are able to design a manager to fire reconfiguration commands in order to enforce the objectives. Comete together with Fractal API provide means to reflect manager commands on the managed system. Those means we are interested in are *component migration* provided by Comete API and *lifecycle* (resp., *binding*) control interface of the Fractal API in order to start/stop (resp., bind/unbind) components. In the following, we list the control objectives:

Processing element availability A processing element is a shared resource that may be unavailable for some reasons (e.g., energy saving, higher priority task, fault, etc.). We want that no component is running on an unavailable processing element (*component migration*).

Workload balancing There is a maximum workload, a processing element should not exceed. Migrate components in order to decrease the workload.

Quality-of-Service When the $\texttt{fileserver}_1$ is overloaded, start the $\texttt{fileserver}_2$ in order to keep the average response time low (*lifecycle and binding control*).

Exclusiveness The `logger` should be started upon the request of the user, and then $\texttt{fileserver}_2$ should not be running (*lifecycle and binding control*).

In order to ensure the objectives we described above, a manager is implemented and put in a closed-loop with the software application as described in section 2.4. A simple scenario is as follows. The manager receives an event stating that the $\texttt{fileserver}_1$ is overloaded. It performs some computation and emits the command to start the $\texttt{fileserver}_2$ and bind it to the rest of the components. We measure the workload of $\texttt{fileserver}_1$ by comparing the size of the FIFO associated to it with a specific threshold. Next, the controller receives from the environment that the processing element $\texttt{pe}_3$ is no longer available. The controller reacts by emitting the command to migrate the components running on $\texttt{pe}_3$ to other processing elements, balancing the workload as described by the related property.

4 Designing the manager

We follow a modeling method for the design of a manager to enforce the properties we described above; we will elaborate on it by improving the treament of reconfiguration actions in Section 6. The use of DCS ensures that the manager is correct with respect to the properties given as objectives. It mainly consists of the following steps: (1) Providing a synchronous model of the behavior of the reconfigurable components and the processing elements, giving the state space of the control problem; (2) Specifying the control objectives and identifying the controllable variables; (3) Compiling: controller synthesis and code generation.

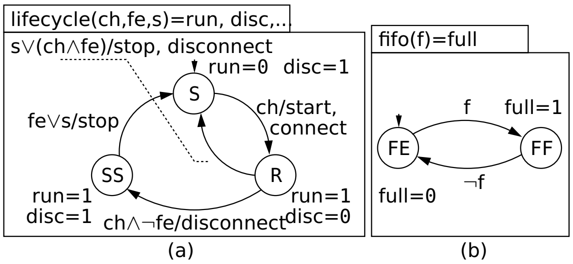
4.1 Modeling Components With Heptagon

We adopt a modular modeling approach to enable reusing models. Moreover, we distinguish between the application models (i.e., software) and the execution platform (e.g., hardware) models. Instances of these models will be associated in the main synchronous program (see Section 4.3) to obtain the global model. Hence, the approach facilitates replacing hardware models without modifying software ones in case the application is ported to other execution platforms.

4.1.1 Modeling Software Components

Component Lifecycle Figure 9-(a) shows the automaton modeling a Fractal component lifecycle. A component may be in one of three possible states. A `Running (R)` component is connected to the other components and may handle incoming messages in its associated FIFO. A `Stopped (S)` component is disconnected from the others and can not execute to handle messages in its input FIFO. Finally, a component may be put in a `Safe Stopping (SS)` in order to handle the remaining messages in its FIFO before completely stopping.

The automaton has three inputs (ch, fe and s for change, FIFOs empty and stop respectively), and two outputs run and disc. Initially, the component is stopped (i.e., state S). It goes to R (i.e., running) upon receiving ch. From this state, it goes to state SS upon receiving ch. At state SS, it goes to state S upon receiving fe which means that the input FIFOs are empty. At any state, receiving s forces the automaton to go to the state S. The two outputs of the automaton run and disc (for running and disconnect) take their values as described by the figure. These outputs tell whether a component is running (resp., disconnected) or not.



(a) (b)

Figure 9: Model of: (a) Lifcecyle, (b)FIFO state

As described by the transitions of the automaton, some reconfiguration commands (start, stop, connect, disconnect), represented as additional outputs, are fired upon changing state. These commands are meant to reconfigure the related component in order to be coherent with its lifecycle model. For instance the transition from S to R, outputs the commands start and connect in order to start the component and connect it to the rest of the components.

FIFO State With each server interface of a component is associated a FIFO. The FIFO stores the input events before they are handled by the component. Figure 9-(b) models the state of such FIFOs. It is a two state automaton with one input and one output (f and full respectively). Initially, the FIFO is empty. It goes to the state FF (resp., FE) upon receiving f (resp., $\neg$f). The value output full indicates the state of the FIFO: true (resp., false) means that the FIFO size is above (resp., below) a given threshold.

4.1.2 Modeling Hardware Components

Component mapping on Processing Elements Figure 10 models the mapping of a component on the available processing elements. It is a four state automaton. Each state represents mapping on one processing element, on which a component may run. It may change from one state to another depending on the Boolean inputs a and b. That is, each state has three outgoing transitions (one to each remaining state). The transition to take depends on the value of a and b (two Boolean inputs to encode four possibilities). For the sake of clarity, not all of the transitions are present in the figure. The output of the automaton is of enumerated type; it takes its values in $\{pe_0, pe_1, pe_2, pe_3\}$ depending on the state of the automaton.

The transitions of the automaton are associated with outputs (mig_0, mig_1, mig_2, mig_3). They are associated to migration commands, in order to migrate the corresponding component to its new processing element.

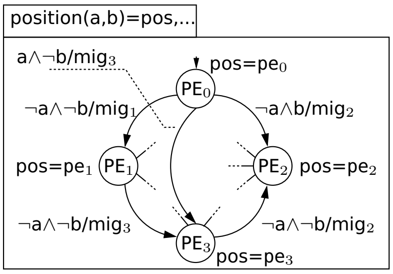


Figure 10: Partial model of a component mapping.

Processing Element Availability We need a model of the availability of a processing element in order to know whether a component may run on it or not. Figure 11 describes such a model. It is a two states automaton with one input dis (for disable) and one output on which tells on the availability of the processing element. Initially, a processing element is available (i.e., at state ON). It goes to the state OFF (resp., ON) upon receiving dis (resp., $\neg$dis). The output on takes its value depending on the state of the automaton.

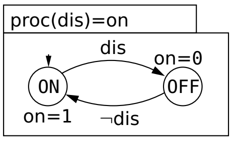


Figure 11: Model of processing element availability

Processing Element Workload The workload of a $\mathtt{pe}_i$ depends on the components running on it and their bindings to the other components. Two components C and S (client and server), bound through their respective interfaces I_c and I_s induce some workload on their associated processing elements $\mathtt{pe}_c$ and $\mathtt{pe}_s$. Based on some benchmarks, we estimate the workload related to $\mathtt{pe}_c$ and $\mathtt{pe}_s$ as described by the following equations:

a) $load(pe, s) = f \times c$. b) $load(pe, c) = f \times distance$.

The workload of the processing element running the server component S depends on the cost c of the executed function, and the function call frequency f. The processing element running the client component is charged with the communication effort. That is, the processing element workload depends on the call frequency and a parameter $distance$ which depends on the position of the two communicating processing elements. In our case-study, the processing elements are processor cores. The communication uses cache memories of distinct levels. The values we associate to $distance$ represent the fact that the communication cost differs depending on the type of cache used between the processing elements. These values are: $distance = 200$ in case $\mathtt{pe}_c$ and $\mathtt{pe}_s$ refer to the same core, $distance = 500$ in case the processing elements are distinct cores of the same processor, $distance = 1000$ in case the two processing elements refer to distinct cores of distinct processors.

In Listing 1, we give the interface of the node cost implementing workload equations. For a given binding between two components, the node takes as input the position of the client and the server component, the frequency f of the calls between them and the cost of the function executed by the server component. The outputs are integer values associating workloads induced by the two components with the available processing elements. The involved processing elements will be associated with positive integer values, the others will take the value zero.

```
node cost(posC, posS: peid; f, c: int)
returns (cp0, cp1, cp2, cp3: int)
```

Listing 1: The interface of the node cost

4.2 Complete system model

The main program (partially described in listing of Listing 2) consists of the synchronous composition of the automata modeling each component (hardware and software). The inputs of the main node consist of the events stating on: (pes1) the availability of the processing element $\mathtt{pe}_1$; (add_L) the request of the `logger`; (f_S1, f_S2, f_L) the load of the FIFO associated with `fileserver`$_1$, `fileserver`$_2$ and `logger` respectively.

The outputs of the main program correspond to the reconfiguration/migration commands fired by the automata. For instance mig_L_p0 is fired by the automaton modeling the position of the component `logger` (line 22). When mig_L_p0 takes the value true, the component `logger` should be migrated to `pe`$_0$. The output start_S2 is fired by the automaton modeling `fileserver`$_2$ lifecycle (line 17). When it is true, it states that the component should be started.

The main program is composed of: An instance of node fifo for each server interface of the components (lines 11-13). An instance of node lifecycle for each component (lines 15-17). An instance of processing element availability for each processing element (lines 19-20). An instance of node position for each component (lines 22-24). An instance of node cost for each binding between components (lines 26-28). Finally, the cost imposed on each processing element corresponds to the sum of all the cost induced by the bindings (lines 29-30).

```
node main(pes1, add_L, f_S1, f_S2, f_L: bool)
     returns (mig_L_p0, ..., mig_L_p3:int;...
        mig_F_p0, ..., mig_F_p3:int;
        start_S2, stop_S2, conn_S2, disc_S2:bool;
        start_L, stop_L, conn_L, disc_L:bool)
contract
  enforce(pe_av and wl_ba and qos and exc);
  with(....)

let
  full_S1 = fifo(f_S1);
  full_S2 = fifo(f_S2);
  full_L  = fifo(f_L);
  ...
  (run_L,disc_L,start_L,stop_L,conn_L,disc_L)
         = lifecycle(ch_L, full_L, s_L);
  (run_S2,disc_S2...)=lifecycle(ch_S2,full_S2,s_S2);
  ...
  pe1 = proc(pes1);
  pe2 = proc(false);
  ...
  (pos_L,mig_L_p0,mig_L_p1,mig_L_p2,mig_L_p3)
        = position(a_L, b_L);
  (pos_S1,...)= position(a_S1, b_S1);
  ...
  (c11,c21,c31,c41)=cost(pos_D, pos_S1, f_1, c_1);
  (c12,c22,c32,c42)=cost(pos_D, pos_S1, f_2, c_2);
  ...
  wl1 = c11 + c12 + ... + c18;
  ...
tel
```

Listing 2: The main program of the model

4.3 Describing Control Objectives with BZR

We now have a complete model of the possible behaviors of the system, in the absence of control. We want to obtain a controller that will enforce the policy given informally in Section 3.3. We do this in the form of a contract for each of the points of the policy as follows.

Processing element availability We want no component running on an unavailable processing element: this can be achieved by *component migration*.

The corresponding expression in the contract to be enforced i.e., to be controlled for invariance, is:

$$\mathtt{pe\_av} = \bigwedge_{i \in PE} \left(\, \mathtt{pe}_i \; \vee \; \mathbf{not} \left(\bigvee_{j \in comp} pos\_j = \mathtt{pe}_i \right) \right)$$

The involved controllable variables defined in the **with** part will be, for each component: $\mathtt{a}\_j$, $\mathtt{b}\_j$, $j \in comp$. The effect of the control will be that, when some processing element becomes unavailable, appropriate migrations will be fired. More precisely: upon reception of input $\mathtt{pes}_i$ at value **true**, the availability model makes a transition as shown in Figure 11. In the global model, a transition will be taken to a next state where the above expression is **true**. For components on the unavailable PE, the automata shown in Figure 10 can not stay in the current state without violating the property, therefore controllables will take a value such that a local transition occurs, hence firing a migration action towards another available PE.

Workload balancing Workload on each PEi is bounded by Max_i: this is achieved by *component migration*.The expression is:

$\texttt{wl\_ba} = \bigwedge_{i \in PE}(\ \texttt{wl}_i \leq Max_i\ )$.

The involved controllables are: $\texttt{a\_}j$, $\texttt{b\_}j$, $j \in comp$. The effect of the control will be that, if some migration or starting of a component happens on a PE, the choice encoded by controllables will be between PEs for which this addition would not violate the bound.

Quality-of-Service When the $\texttt{fileserver}_1$ is overloaded, start the $\texttt{fileserver}_2$: this is achieved by *lifecycle and binding control*. The expressions are:

$\texttt{qos=}((\textbf{not}\ \texttt{full\_S1}) \Rightarrow (\texttt{disc\_S2}))$
$\texttt{and}((\textbf{not}\ \texttt{run\_L}\ \textbf{and}\ \texttt{full\_S1}) \Rightarrow (\texttt{run\_S2}\ \textbf{and}\ \textbf{not}\ \texttt{disc\_S2}))$.

The involved controllables are: `s_S2`, `ch_S2`. The effect of the control will be that, if the FIFO of $\texttt{fileserver}_1$ reaches its threshold, the $\texttt{fileserver}_2$ is started unless the Logger runs; and when the FIFO goes back under the threshold, it is disconnected, and eventually stopped.

Exclusiveness When the `logger` runs, $\texttt{fileserver}_2$ doesn't: this is achieved by *lifecycle and binding control*. The corresponding expression is:

$$\texttt{exc} = (\texttt{run\_L}\ \textbf{and}\ \textbf{not}\ \texttt{disc\_L}) \Rightarrow \texttt{disc\_S2}$$

The involved controllable is: `s_S2`. The effect of the control is that, when the Logger is started, the $\texttt{fileserver}_2$ must be stopping or idle.

5 Manager Integration

The C code generated by the BZR compilation tool-chain consists of two functions *step(...)* and *reset(...)*. Listing 3 is a sketch of a program using these functions in order to manage the reconfiguration of a system. At line 2, *reset()* is called once to initialize the memory of the program. The piece of code ranging from line 5 to line 10 is made sensitive to incoming events; i.e., this part of the code is executed each time new events are received. The function *prepare_events()* prepares the events and provides the inputs for the function *step()*. The signature of the function *step()* corresponds to the one of the main program of Listing 2. A call to this function corresponds to a step of the synchronous program. The outputs of *step()* are given to the function *generate_commands()* in order to translate the outputs into the consequent reconfiguration commands.

```
1 // program initialization
2 reset(&mem);
3 ...
4 // reading input events
5 <pes1, add_L, f_S1, f_S2, f_L> = prepare_events();
6 // calling the step function
```

```
<outputs>=step(pes1, add_L, f_S1, f_S2, f_L, &mem);
// translate outputs to reconfiguration commands
generate_commands(<outputs>);
...
```

Listing 3: Principle of manager integration

5.1 Wrapping the manager into a component

The functions listed in Listing 3 are wrapped into a Fractal component in order to be connected to the rest of application and middleware components (see Figure 12). This component provides one server interface *Events* and one client interface *Commands*. The signature of the interface *Events* is described in Listing 4. The method *setEvent* provided by this interface enables components to register event values. Indeed, the manager hides a buffer containing the last value registered of each event. The function *prepare_events()* uses this buffer to provide the inputs for the function *step()*.

The function *generate_commands()* translates the outputs of the function *step()* into method calls through the *Commands* interface. Indeed, the signature of the required interface *Commands* is that of *ComponentManager* provided by Comete API (see Listing 4). It provides methods for component migration, reconfiguration and lifecycle management.

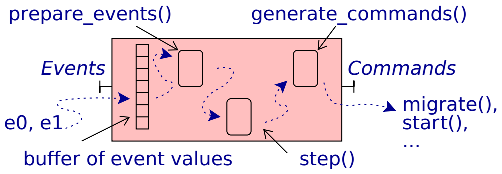


Figure 12: The manager as a Fractal component

```
interface SynchronousManager.api.Events{
  setEvent(in event_name, in event_value)
}

interface Comete.generic.api.ComponentManager {
  instantiate(in component_binary_name,
              in target_pe_id, out instance_id)
  destroy(in instance_id)
  bind(in client_id, in client_interface_name,
       in server_id, in server_interface_id,
       in binding_type)
  unbind(in client_id, in client_interface_name)
  start(in instance_id)
  stop(in instance_id)
  migrate(in instance_id, in target_pe_id)
}
```

Listing 4: *Events* and *Commands* interface signature

5.2 Concrete integration of the manager

Figure 13 describes the concrete integration of the manager together with the application and Comete middleware components. The component user is connected to the *Events* interface of the manager. It intercepts user requests (add logger for instance) and registers them as events. The FIFO components (part of Comete) are made implicit in the figure. As illustrated by the dotted lines, the FIFOs associated with `fileserver`$_1$, `fileserver`$_2$ and `logger` are connected to the *Events* interface of the manager in order to register events related to their size.

Upon receiving events, the manager performs some computation and fires some reconfiguration commands in the form of method calls. These method calls are handled by the *loader* of Comete. Indeed, as illustrated by the dashed lines, the *loader* is connected to the control interface of the application components in order to start/stop bind/unbind them. Moreover, the loader knows how migrating components should be achieved regarding the execution platform.

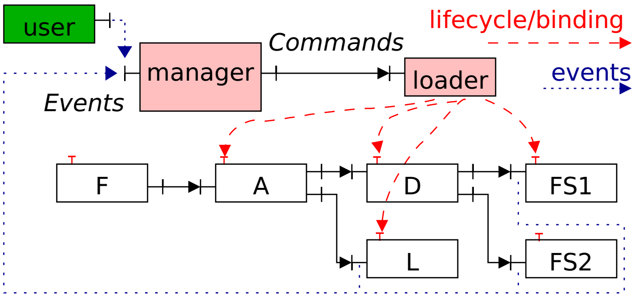


Figure 13: Integration of the manager in the system

Comete middleware hides complex mechanisms and provides a simple API to manage component lifecycle, migration, and architecture transparently, whatever the execution platform is. For that purpose, we adopted a centralized version of the manager emitting reconfiguration commands to Comete instead of application components. We could make the manager communicate directly with the components (modulo re-engineering Comete) to enable a distributed implementation of the manager. This means that parts of the manager would be integrated into Fractal component membranes.

In this section we presented a simple version of the use of a synchronous manager for the reconfiguration of a parallel and asynchronous application. The reconfiguration commands were considered to be short enough to complete before the next reaction of the manager. However, this is not always the case. Indeed, some reconfiguration commands may take non negligible time to complete. During this time interval, the system model inside the manager does not reflect the actual state of the system. For instance, when the command to start a component C_x is fired, the model of C_x is at state `Running (R)` but the actual component is not yet started. The manager reaction to other incoming events during reconfiguration progress may lead the managed system to undesirable states. Next section proposes an extension to our modeling approach in order to solve issues related to asynchronous commands.

6 Asynchronous commands

We propose a controller architecture together with some guidelines for writing synchronous models in order to overcome the issues related to the transient incoherence of the model with respect to the state of the system. Our approach relies on the explicit representation of the states where some commands are being processed but not completed, together with some synchronization mechanisms.

The purpose of identifying the states where some reconfiguration commands are in progress is to make it possible for the programmer to decide what should happen during reconfigurations. The programmer should rewrite objectives properties taking into account these situations.

6.1 Modeling command execution

6.1.1 Behavioral models

Some synchronous languages (e.g., Esterel) provide built-in constructs in order to perform asynchronous calls within a synchronous program [7, 18]. We follow the same principle of such constructs and model the asynchronous[3] aspect of reconfiguration commands by means of an automaton.

Such an automaton is illustrated by Figure 14. At state P (for pending), the command is emitted but not yet completed. At state D, the command has finished. The automaton has two inputs do and done. The output pending tells whether the command is pending or not. From the initial state, upon receiving do which corresponds to the firing of the command, the automaton goes to state P. The automaton stay at state P until receiving the input done. The input done signals that the command has been completed.

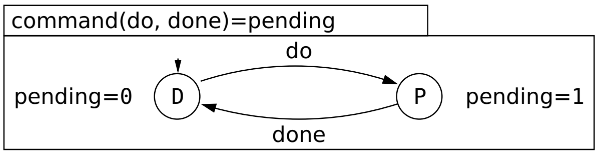


Figure 14: Automaton for an asynchronous action.

We associate an instance of such an automaton with each asynchcronous command. Hence the modification w.r.t. model of Section 4 is systematic and modular.

6.1.2 Objectives and contracts

Being aware of the state where a command is in progress but not finished, one can add some properties to be enforced and modify the previous ones in order to decide what should happen during reconfigurations. For illustration purpose, we consider the same program as in Section 4.3 with the

[3] by asynchronous we mean that a command takes more than one reaction step to complete.

command start_S2 as the only command taking non-negligible time to complete. In the following fragment of program, we associate an instance of the node command to start_S2 (line 5) and modify the contract:

```
1 ...
2 enforce (pend and  pe_av and wl_ba and
3          (not pen_sS2  ⟹  (qos and exc)))
4 ...
5 pen_sS2 = command(start_S2, done);
6 pend = not (pen_sS2 and disc_S2) ; ...
```

The contract changes as follows:

- The properties qos and exc are now dependent to the pending of the command. **not** pen_sS2 ⟹ (qos **and** exc) tells that qos and exc may not be enforced when the command start_S2 is in progress. But, they must be enforced once the command completes.
- pend (defined at line 6) is a new property. It forbids the controller to modify the state of the fileserver$_2$ when the command to start it is in progress. This property is explained below.

In order to understand the property pend, consider the automata modeling fileserver$_2$ lifecycle and start_S2 command execution in Figure 15. Adding the property pend = **not** (pen_sS2 **and** disc_S2) forces the controller to avoid the states where the command is pending (i.e., at state p) and any state where disc_S2 is true (i.e., at states s and ss). That is, to avoid the states (ss,p) and (s,p) in the product of the two automata.

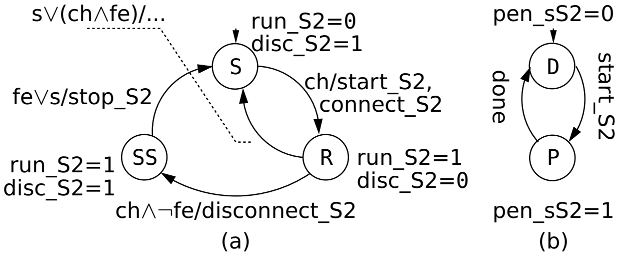


Figure 15: Instances of lifecycle(a) and command(b)

At the beginning the automata are at state (s,d). Upon receiving ch, automaton (a) changes states and fires the command start_S2 which makes automaton (b) change state at the same instant. That is, the global state is (r,p). Unless receiving done, which is uncontrollable, the controller will not make automaton (a) change state in order not to violate the property pend.

6.2 Controller Architecture

The actual implementation of the manager is component-based. Figure 16 describes the internals of such a component. It consists of three subcomponents: evt, ctrl, and cmd. The functions *prepare_events()*, *reset()*, *step()*, and *generate_commands()* are spread over the subcomponents as follows: evt encapsulates *prepare_events()*. It prepares the inputs to ctrl that encapsulates *reset()* and *step()* generated by the BZR compilation. The component `cmd` translates the outputs of `ctrl` into reconfiguration methods calls.

The component `evt` is in charge of implementing the method provided by the interface *Events* from which components and the environment may register events. The component `cmd` is in charge of calling reconfiguration methods through *Commands* interface. Moreover, each time a reconfiguration command is completed, `cmd` notices `done` events related to this command. That is why `cmd` is bound to `evt`.

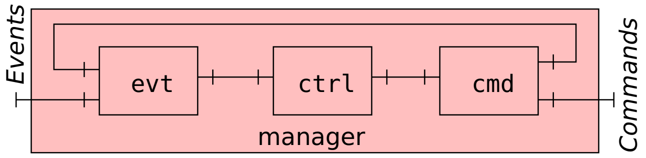


Figure 16: Internal view of the controller

7 Related work

A lot of work has been devoted to dynamic reconfiguration of component-based software systems. In the case of Fractal, one can refer to the tools for component introspection and the languages for specifying reconfigurations [12] or integration in parallel frameworks [10].

Our work fits in the context of applying formal methods for dynamic reconfiguration [6, 16, 15, 19] in order to ensure properties related to components and reconfigurations. In [15] the authors specify reconfiguration properties by means of temporal logic in order to apply model checking or runtime monitoring of system reconfigurations. In comparison, our approach benefits from the discrete controller synthesis which provides a correct by construction manager ensuring properties on components and reconfigurations. In [14], the authors give a general software engineering framework, with some indications of integration, but no complete implementation. In comparison, we perform a concrete integration of synchronous managers with Fractal, in the form of its Cecilia programming environment, and the Comete middleware. In addition, we consider and treat the case of asynchronous reconfiguration actions.

Other related work concern the concrete integration of synchronous programs for the management of asynchronous systems. In [8], the authors provides a synchronous controller for configuring device drivers aiming at global power management of embedded systems. Apart from their not using DCS techniques, the difference with our approach lies in the call of reconfiguration functions. Indeed, in [8], function calls are performed inside the reaction of the controller. This has the benefit of always keeping a model reflecting the exact state of the system. In our case we use to follow the principle of asynchronous function calls as in [7, 18].

8 Conclusion

Our contribution is on the one hand a synchronous model of the behavior of the reconfigurable components, in the form of the state space of the control problem, and the specification of the control objectives (Section 4). On the other hand, we contribute a component-based architecture for the implementation of the resulting controller with the Fractal/Cecilia programming environment, taking advantage of the Comete middleware. (Sections 5 and 6)

In this work, we apply formal techniques issued from academic research to an open, stable and available framework supported by an independent industry consortium [5]. The Fractal model is currently being implemented in the MIND project [4], a sequel to the Cecilia framework. The MIND framework is a collaborative initiative to spread component-based software engineering to an even larger community of academic institutions and industries. The present contribution keeps its legitimacy in the MIND context as most of the concepts and semantics introduced with Cecilia and Comete remain. Therefore, applying our contribution to MIND should be limited to a straightforward adaptation.

We have ongoing work on another interesting case study: an H264 video processing application, implemented on a multicore architecture using Cecilia and Comete. It could be handled following the very same methodology we propose, because it follows essentially the same structure: degrees of reconfiguration concern migrations, and adding and removal of components performing video effects on the stream.

Perspectives are in the line of generalizing our proposal at the language-level, by extending the Fractal ADL with a way to incorporate automata notation for reconfiguration description, as well as the behavioral contracts, and integrating the application of BZR in a global Fractal compilation flow. Also, the integration of synchronous controllers in a Fractal components architecture should be facilitated by identifying general programming guidelines, providing end-users with informal rules such that components are controllable, and the synchronous instant and step (and the states in the automata) fits well with the event granularity.

A Complete Example and Integration

In what follows, we describe the complete integration process of the code produced by the bzr compilation for the control of the *comanche* application software. Writing the synchronous model of the manager has been discussed above. Here we give more details providing simulation results of the manager and the process of integration in the Fractal application.

B Simulation of the Synchronous Program

The interface of the synchronous program is described in Listing 5. The input *disable* states on the availability of the processing element PE_3. This input takes the value *false* as long as PE_3 is available (*false otherwise*). *fifoH1F, fifoH2F, fifoL2F* are Booleans stating on the size of the FIFOs associated with FileServer1, FileServer2 and Logger respectively. They take the value *false* as long as the size of the associated FIFOs is under their specific threshold. The input *addlog* tells whether the logger is required or not. The input *c_startH2_done* notifies that the command for starting FileServer2 has been actually completed.

The outputs of the main program are associated with the possible commands provided by the reconfigurable system. An output associated with a command takes the value *false* until the command has to be fired. Then, it takes the value *true* during one step.

```
node main(disable , fifoH1F , fifoH2F , fifoL2F , addlog , c_startH2_done:bool)
  returns(
    c_startH2 , c_stopH2 , c_connectH2 , c_diconnectH2:bool;
    c_startL , c_stopL , c_connectL , c_diconnectL:bool;
    c_mig1f1  , c_mig2f1  , c_mig3f1  , c_mig4f1 ,
    c_mig1f2  , c_mig2f2  , c_mig3f2  , c_mig4f2  ,
    c_mig1d  , c_mig2d  , c_mig3d  , c_mig4d:bool;
    ...)
```

Listing 5: The interface of the main program

Figure 17 illustrates part of the simulation results. The inputs (in blue color) are given (by the user) in order to simulate the events received from the application. The outputs (in red color) corresponding to reconfiguration commands take the value *true* each time a command has to be fired.

The interaction scenario as illustrated by the figure is as follows:

- Step 1 is for initializing the system (not relevant).
- at Step 4 the input fifoH1F changes value to *true*. The program reacts and assigns the value *true* to the outputs *c_startH2* and *c_connectH2* to true. These are the commands to be fired at this step. This situation corresponds to the activation of `fileserver`$_2$ when `fileserver`$_1$ is overloaded.
- at Step 8, the program receives the notification that *c_startH2* has been completed.

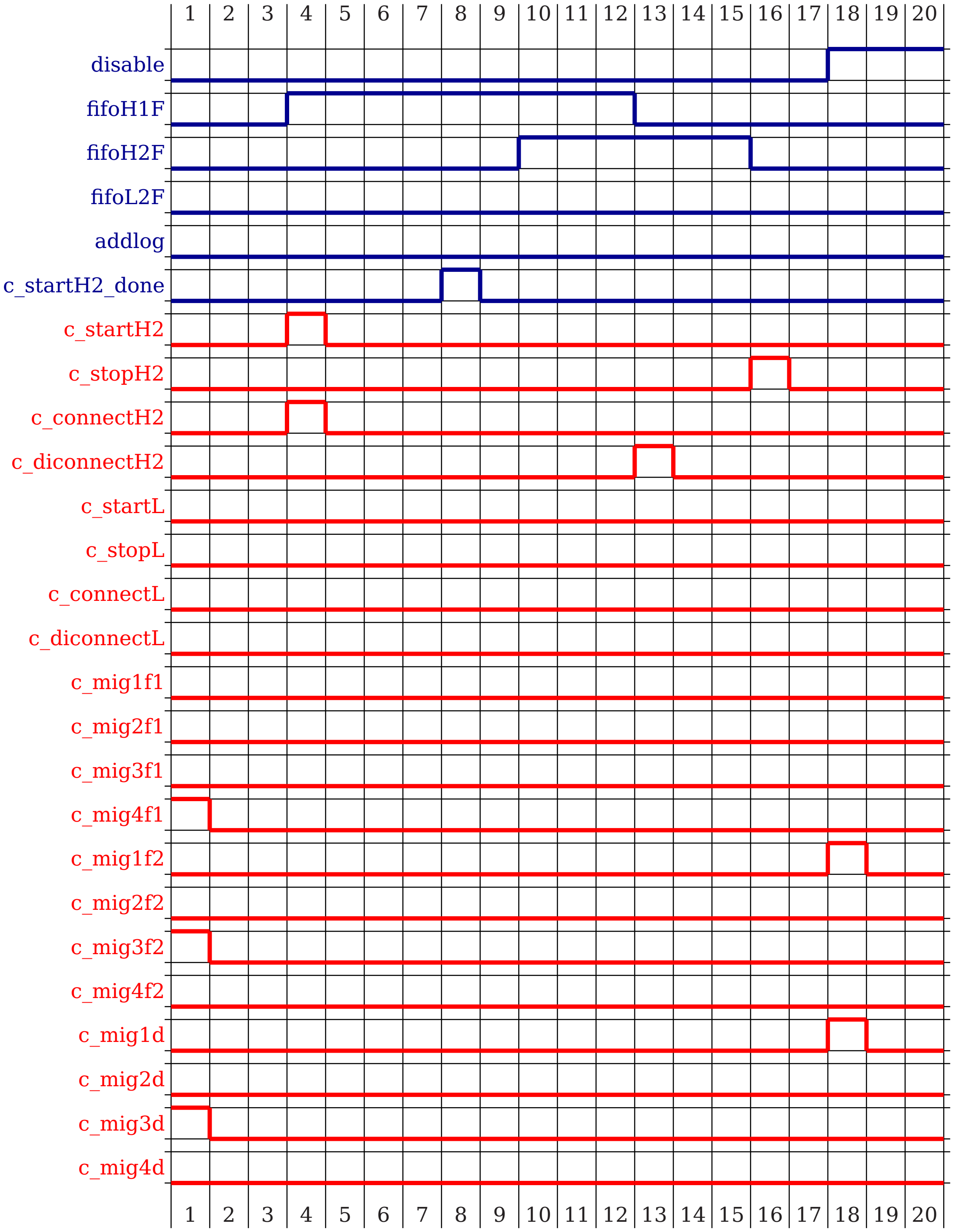

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
disable
fifoH1F
fifoH2F
fifoL2F
addlog
c_startH2_done
c_startH2
c_stopH2
c_connectH2
c_diconnectH2
c_startL
c_stopL
c_connectL
c_diconnectL
c_mig1f1
c_mig2f1
c_mig3f1
c_mig4f1
c_mig1f2
c_mig2f2
c_mig3f2
c_mig4f2
c_mig1d
c_mig2d
c_mig3d
c_mig4d

Figure 17:

- at Step 10 the input fifoH2F changes value to *true*. The program reacts but no command is fired.
- at Step 13 the input fifoH1F changes value to *false*. The program reacts and assigns the value *true* to the output *c_disconnectH2*. This corresponds to the situation where the FileServer2 is no longer required. It should be disconnected and left executing in order to terminate the remaining inputs in its FIFO.
- at Step 16, the input fifoH2F changes value to *false*. The program reacts and assigns the value *true* to *c_stopH2*. Now that the FIFO associated with the FileServer2 is empty, the component can be completely stopped.
- at Step 18, the input disable changes to *true*. The program reacts and assigns the value true to *c_mig1f2* and *c_mig1d*. This corresponds to the migration of the components running on PE_0 to other PE_s. The output *c_mig1f2* (resp., *c_mig1d*) corresponds to the migration of the `fileserver`$_2$ (resp., `dispatcher`) to PE_1.

B.1 BZR Program Compilation into C Code

The BZR compilation tool chain compiles the synchronous program into C code. The generated functions of our interest are the reset() and step() functions of the main program. reset() initializes the memory associated with the synchronous program. The function step() computes the outputs of the program depending on the inputs. It also updates the internal memory.

```

main_res main_step(int disable, int fifoH1F, int fifoH2F,
                   int fifoL2F, int addlog,
                   int c_startH2_done, main_mem* self);
```

Listing 6: Prototype of the function step

The prototype of the function step() is presented in Listing 5. It has the same input parameters as the main function of the synchronous program in Listing 5. The parameter `self` refer to the memory of the complete program. The returned value of the function step() is of type `main_res` it is a structure. Each field of this structure corresponds to an output of the main program.

C Integration of the step() function

The integration process of a synchronous manager for the control of reconfiguration lies in the integration of the step() function generated by the synchronous compilation. In order to achieve this, we have to make some choices. In particular, we need to decide when the function should be called, and how to apply reconfiguration commands:

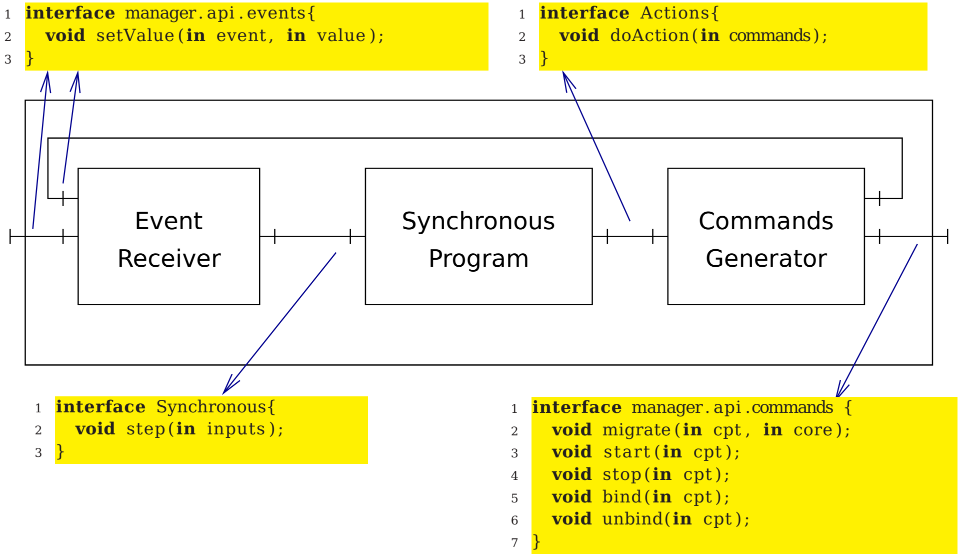
- **Calling Step():** Two approaches are possible: Time-triggered or Event-triggered. The first solution consists of calling the step periodically each `t time units`. The second one consists of making the step function sensitive to incoming events; i.e., each time a new event is notified, step is called.

- **Firing commands:** The outputs of the step function correspond to commands to be fired. One way to apply these commands is to call the reconfiguration commands before the step is finished. The advantage of doing so is to have a model always reflecting the state of the system. The other approach is to complete the step and deal with the commands to be fired in an asynchronous way.

Our choice for this case study is to consider event-triggered calls of the step together with asynchronous firing of commands. The choice may be discussed. But this is out of the scope of the paper.

C.1 Wrapping The Generated Code Into a Fractal Component

The C generated code is wrapped into a Fractal component in order to receive application events and fire reconfiguration commands. Figure 18 describes the internals of such a component and the type of the provided (resp. required) interface of the components. Notice that two interfaces may be bound if an only if they are of the same type.



```
interface manager.api.events{
  void setValue(in event, in value);
}
```

```
interface Actions{
  void doAction(in commands);
}
```

```
interface Synchronous{
  void step(in inputs);
}
```

```
interface manager.api.commands {
  void migrate(in cpt, in core);
  void start(in cpt);
  void stop(in cpt);
  void bind(in cpt);
  void unbind(in cpt);
}
```

Figure 18:

C.1.1 The Component EventReceiver

Listing 7 illustrates part of the implementation of the method *setValue* of the `EventReceiver` component. This method is called by other components in order to notify events.

The component manages an array of Boolean values (DATA.step_inputs). Each cell of the array corresponds to one input event. Each time an event is notified through this method, its value is updated, then the method step is called. This method is implemented by the component `SynchronousProgram`. The parameter of the call is the array of event values.

```
void METHOD(events, setValue)(void *_this, int event , int value) {
  ...
  DATA.step_inputs[event]=value;
  ...
  CALL(REQUIRED.synchprog, step, DATA.step_inputs);
}
```

Listing 7: code of the setValue method

C.1.2 The Component SynchronousProgram

Listing 8 illustrates the code associated with the method *step* of the component `SynchronousProgram`. This method contains a call to the *step* method actually generated by the BZR compilation (at line 7). The result of the synchronous step (i.e., `DATA.res`) are forwarded to the `CommandsGenerator` in order to translate them into the consequent reconfiguration commands.

```
1  void METHOD(step, step)(void *_this, jboolean *inputs) {
2    int input_disable = inputs[0];
3    int input_fifoH1F = inputs[1];
4    ...
5
6    //Call to the generated method step()
7    DATA.res = Migration_main_step(input_disable ,input_fifoH1F ,
8                                   input_fifoH2F , input_fifoL2F,
9                                   input_addlog ,input_c_startH2_done,
10                                  &DATA.mem);
11
12   CALL(REQUIRED.actionManager, doAction, (void *) &DATA.res);
13   }
```

Listing 8: Code of the step method

C.1.3 The Component CommandsGenerator

Listing 9 illustrates the implementation of the method *doAction* of the component `CommandsGenerator`. It receives a structure containing the Boolean values associated with the possible commands. This method parses the received values and fire a command each time the associated output variable has the value `true`.

```
void METHOD(actionManager, doAction)(void *_this, void * sync_output) {
  Migration_main_res *commands;
  commands = (Migration_main_res *)sync_output;

  if(commands->Migration_c_startH2 )
    CALL(REQUIRED.commands, start, FILES2);

  if(commands->Migration_c_stopH2)
    CALL(REQUIRED.commands, stop , FILES2 );
....
```

Listing 9: Code of the doAction method

D Retrieving Application/Environment Events

As explained above, the component `EventReceiver` of the manager offers a method for event notification. The components in charge of notifying these events are connected to this interface. Among the application components, only the FIFOs associated with $\texttt{fileserver}_1$, $\texttt{fileserver}_2$ and `logger` are supposed to send events to the manager. Moreover, there is a component modeling the user and the environment. The component is in charge of notifying the events related to adding the logger and the availability of a processing element. These components call the method *setValue()* of the component `EventReceiver` whenever a particular condition holds. Listing 10 is a sample piece of program for notifying events. The condition depends on the nature of the event to be notified. In case of the FIFOs, the condition concerns the size of the FIFO regarding to its associated threshold.

```
  ...
  if(condition )
 CALL(REQUIRED.events , setValue , EVENT_ID, EVENT_VALUE);
  ...
```

Listing 10: Event notification

References

[1] Cecilia on the ow2 website. http://fractal.ow2.org/cecilia-site/current.

[2] Comete on the ow2 website. http://fractal.ow2.org/minus-site/current/comete.

[3] Fractal on the ow2 website. http://fractal.ow2.org/.

[4] Mind on the ow2 website. http://mind.ow2.org.

[5] The ow2 consortium website. http://www.ow2.org.

[6] T. Barros, R. Ameur-Boulifa, A. Cansado, L. Henrio, and E. Madelaine. Behavioural models for distributed fractal components. *Annals of Telecommunications*, 64, 2009.

[7] G. Berry, S. Ramesh, and R. K. Shyamasundar. Communicating Reactive Processes. In *ACM Symposium on Principles of Programming Languages*, Charleston, South Carolina, 1993.

[8] N. Berthier, F. Maraninchi, and L. Mounier. Synchronous programming of device drivers for global resource control in embedded operating

systems. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES)*, Chicago, IL, USA, apr 2011.

[9] E. Bruneton, T. Coupaye, and J. B. Stefani. Recursive and dynamic software composition with sharing. In *European Conference on Object Oriented Programming*, 2002.

[10] 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.

[11] J.-L. Colaço, B. Pagano, and M. Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. In *International Conference on Embedded Software*, 2005.

[12] 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*, 64:45–63, 2009.

[13] G. Delaval, H. Marchand, and E. Rutten. Contracts for modular discrete controller synthesis. In *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems*, 2010.

[14] G. Delaval and E. Rutten. Reactive model-based control of reconfiguration in the fractal component-based model. In *Proc. of the 13th Int. Symp. on Component Based Software Engineering (CBSE),* Prague, 23-25 June, 2010.

[15] J. Dormoy, O. Kouchnarenko, and A. Lanoix. Using Temporal Logic for Dynamic Reconfigurations of Components. In *7th International Workshop on Formal Aspects of Component Software - FACS'2010*.

[16] M. Léger, T. Ledoux, and T. Coupaye. Reliable dynamic reconfigurations in a reflective component model. In *Proc. of the 13th Int. Symp. on Component Based Software Engineering (CBSE),* Prague, 23-25 June. 2010.

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

[18] J.-P. Paris. *Exécution de tâches asynchrones depuis Esterel*. PhD thesis, University of Nice, 1992.

[19] M. Poulhiès, J. Pulou, and J. Sifakis. Buzz: analyzable embedded component-based software. In *Workshop on Component Models for Embedded Systems (COMES)*, 2008.

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

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399