A Domain-specific Language for The Control of Self-adaptive Component-based Architecture

Frederico Alvares, Eric Rutten, Lionel Seinturier

A Domain-specific Language for The Control of Self-adaptive Component-based Architecture

Frederico Alvares[a], Eric Rutten[b], Lionel Seinturier[c]

[a]*Mines-Nantes & INRIA Bretagne & LINA*
Nantes, France
frederico.alvares@inria.fr
[b]*INRIA Rhône-Alpes*
Grenoble, France
eric.rutten@inria.fr
[c]*Université Lille 1 & INRIA Lille*
Lille, France
lionel.seinturier@univ-lille1.fr

Abstract

Self-adaptive behaviors in the context of Component-based Architecture are generally designed based on past monitoring events, configurations (component assemblies) as well as behavioural programs defining the adaptation logics and invariant properties. Providing assurances on the navigation through the configuration space remains a challenge. That requires taking decisions on predictions on the possible futures of the system in order to avoid going into branches of the behavioural program leading to bad configurations. This article proposes the design of self-adaptive software components based on logical discrete control approaches, in which the self-adaptive behavioural models enriches component controllers with a knowledge not only on events, configurations and past history, but also with possible future configurations. We present Ctrl-F, a Domain-specific Language whose objective is to provide high-level support for describing these control policies. Ctrl-F is formally defined by translation to Finite State Automata, which allow for the exploration of behavioural programs by verification or Discrete Controller Synthesis, i.e., by automatically generating a controller to enforce correct self-adaptive behaviours. We integrate Ctrl-F with FraSCAti, a Service Component Architecture middleware platform and we illustrate the use of Ctrl-F by applying it to two case studies: a news web application and a mutual exclusive task workflow.

Keywords:
Component-based Architecture, Self-adaptation, Discrete Control
2000 MSC: , 10.040, 10.280, 10.090, 10.050, 20.120

1. Introduction

Architecting software-intensive systems has become a challenging task. From tiny applications embedded in house appliances or automobiles, passing through highly connected cyber physical systems, to huge and distributed services in the Cloud, nowadays software systems have to fulfill a number of requirements in terms of operational cost, safety and Quality of Service (QoS) while facing highly dynamic environments (e.g., varying workloads and changing user requirements) and platforms (e.g., software/hardware resource availability and failures). That level of dynamicity makes it difficult, often almost impossible, to manage such software systems in a manual way. It becomes thus imperative to engineer and architect with principles of self-adaptiveness in mind, i.e., to equip these software systems with capabilities to cope with environmental and contextual changes occurring at runtime.

The design of such complex computing systems has been improved by the help of structural organization support of component-based architecture, in which software components encapsulates functionalities that can be accessed through well-defined interfaces that do not depend on their particular implementations. In addition to the benefits of modularity and reuse [1] – which are consequences of this structural organization – adaptability and reconfigurability are key properties which are sought with this approach: one wants to be able to adapt the component assemblies (configurations) in order to cope with new requirements and new execution conditions occurring at runtime.

Architecture Description Languages (ADLs) captures the high-level structure of software systems by describing how they are organized by the means of a composition of components. In order to attain self-adaptation, ADLs are generally used to define initial configurations, whereas adaptive behaviours are achieved by programming fine-grained actions (e.g., to add, remove, connect elements), in either general-purpose languages within reflective component-based middleware platforms [2, 3, 4], or with the support of reconfiguration domain-specific languages (DSLs)[5, 6]. This low level of abstraction may turn the definition of transitions among configurations into a very tedious and costly task, which consequently

may lead to error-prone adaptive behaviours. In fact, it may be non-trivial, especially for large and complex architectures (e.g., web applications with hundreds/thousands of replicated components with specific tuning parameters and constraints), to conceive well-mastered self-adaptive behaviours, with assurances on the way the navigation through the configuration space is performed. Our previous work [7] tackles this problem by proposing a DSL that extends classic ADLs with high-level constructs to describe adaptation in software components by means of behavioural programs, i.e., in terms of order and/or conditions under which reconfigurations take place; and policies, i.e., constraints that have to be enforced all along the execution.

Adaptation decisions are taken at runtime by choosing the next configuration (e.g., a set of architectural elements) in function of not only the observed past history (monitoring events, states, configurations), but also on behavioural programs describing the adaptation logics and properties to be kept invariant all over the managed system's execution [8][9]. That form of decision must involve not only updating the current advancement in the behaviors, but also some predictions on the possible futures of the system. These guarantees can be achieved with the support of control theoretical approaches, where the use of behavioural models allows for predictive decisions. Control-based approaches for software systems have been investigated, mainly concerning quantitative aspects and using continuous control [10]. The use of discrete control is however more appropriated in the context of this work, since the purpose here is to choose configurations in a logical-basis. In particular, we address the design of such a *decision-maker* as a Discrete Controller Synthesis (DCS) problem [11], which consists in automatically generating a controller capable of controlling a set of input variables such that a given temporal property is satisfied.

We propose the design of software components based on Finite State Automata (FSA) behavioural models, which provide knowledge on events, states, past history as well as on possible futures, i.e., the space of reachable configurations. This way, we are be able to, by control, avoid going into behavioural program branches leading to wrong configurations. For that purpose, we formally define the semantics of Ctrl-F behavioural programs by the translation to a FSA model [12]. More precisely, we provide full translation from Ctrl-F to the reactive language Heptagon/BZR [13], which allows the compilation towards formal tools and thereby benefit from exploration by both DCS and verification.

Since the combinatorial complexity of that formal exploration can be exponential in the number of configurations and hence very costly to be performed at runtime, and given that adaptive behavioural programs can be known in advance, we advocate that the DCS should rather be performed off-line. That is to say that controllers are compiled away so as to provide correct solutions at runtime. Furthermore, this is done in a maximal permissive way, meaning that besides ensuring the correctness of decisions, the generated controller makes it optimal, in the sense that it keeps the maximum of possible configurations not violating the stated policies, and hence making the controlled system maximally flexible. The result of the compilation of a given behavioural program is an executable function which, at each decision step, takes the current state and current events, and returns a control value corresponding to the next configuration such that the stated policies are respected.

This article extends our previous work [7] and presents a detailed description of Ctrl-F language as well as its translation into Heptagon/BZR. In addition, we provide further details on its implementation and integration with FraSCAti [4], a middleware allowing for runtime reconfiguration of Service Component Architecture (SCA) software systems. We also illustrate our approach throughout two case studies: a news web application, with QoS and cost requirements [14]; and an application with mutual exclusion requirements. Finally, we provide a discussion on the applicability of our approach by executing these applications under different scenarios.

The remainder of this article is organized as follows. Section 2 introduces the main concepts necessary to understand our approach. Section 3 presents the Ctrl-F language. Its compilation to Heptagon/BZR is detailed in Section 4. Section 5 provides some details on the integration of Ctrl-F with a FraSCAti middleware platform. Section 6 presents some case studies and provides some discussion on the applicability of our approach. Related work is discussed in Section 7 and Section 8 concludes this article while pointing out some perspectives for future work.

2. Background

2.1. Component-based Software Architecture

2.1.1. Architecture Basics and Description Language

Software architectures define the high-level structure of software systems by describing how they are organized by the means of a composition of components [1]. Architecture description languages (ADLs) [15] are usually used to capture these architectures. Although the diversity of ADLs, the architectural elements proposed in almost all of them follow the same conceptual basis [16]. A *component* is defined as the most elementary unit of processing or data and it is usually decomposed into two parts: the implementation and the *interface*. The implementation describes the internal behaviour of the actual component, whereas the interfaces define how the component should interact with the environment. A component can be defined as simple or *composite* (i.e., composed of other components). A *connector* corresponds to interactions among components. Actually, it mediates an inter-component communication in diverse forms of interactions. A *configuration* corresponds to a directed graph of components and connectors describing the application's structure and/or a description on how the interactions among components evolve over the time. Other elements like attributes, constraints or architectural styles may also appear in ADLs [16].

2.1.2. Dynamic Reconfiguration

Dynamic reconfiguration denotes a reconfiguration (the passage from one configuration to another) in which it is not necessary to stop the system execution or to entirely redeploy it in

order for the modification to take effect. As a consequence, the number of interferences on the system execution is reduced and availability is increased.

Component-based architectures are very suitable for dynamic reconfigurations. Indeed, thanks to their native characteristics of modularity and reuse, it is possible to isolate the modifications so that the interference on the system execution is mitigated. In addition, with the advent of reflection, modern component models like Fractal [2] and OpenRec [3], among others, bring reflection capabilities to software architectures. They a meta level, in which components are equipped with control interfaces so as to allow for the introspection (observation on the architectural elements e.g., assemblies, interfaces, connectors, and so forth) and intercession (reconfiguration e.g., creation/suppression of components, connectors, etc.).

The Fractal Component Model (cf. Figure 1) includes the basic architectural concepts of hierarchical composition of components, required (client) and provided (server) interfaces, and bindings connecting components' interfaces. The model was also designed with *separation of concerns* design principle in mind. The main idea is to decouple a component implementation into two parts: *content* and *membrane*. The content manages the functional concerns and its operations are exposed by a set of *functional interfaces*. The *membrane* embodies a set of *controllers* that takes care of the non-functional concerns. *Control interfaces* are access points to membrane controllers, which in turn implement some introspection and/or intercession capabilities, making Fractal a distinguished Component Model concerning the support for dynamic reconfiguration. Examples of controllers are the *life-cycle*, which controls the component's behavioural phases (e.g., starting, started, stopping, stopped, etc.); and the *binding* controllers, to dynamically establish or break bindings between component's interface and its environment.

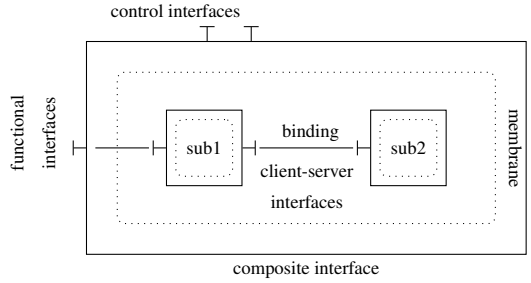


Figure 1: Fractal Architectural Concepts.

In short, component-based architectures, and especially the ones equipped with reflection capabilities like Fractal, have capabilities that are particularly interesting and applicable in the domain of self-adaptive software systems. This is generally achieved by first using ADLs, to define initial architectural configurations, from which, by relying on introspection and reconfiguration mechanisms [2] or languages [5], one can add or remove elements at runtime in response to environment changes, while mitigating the interferences on their execution.

Software components can be controlled according to monitored events, the current state, available configurations and invariant properties [17]. This reactive nature of software components makes reactive systems and languages a very suited to support design of such control behaviour.

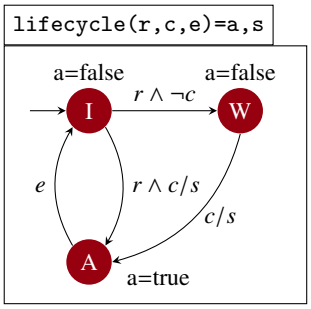


Figure 2: Graphical Representation of Component Lifecycle.

2.2. Reactive Systems and Languages

Reactive Languages have been proposed to describe systems that at each reaction perform a step taking input flows, computing transitions, updating states, triggering actions, emitting output flows [18]. Their definition is often based on Finite State Automata (FSA), which constitute the basic formalism for representing behaviours, as is the case of StateCharts [19] and of synchronous languages [20].

2.2.1. Heptagon

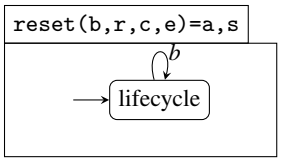
Heptagon/BZR [13] is an example of such languages. It allows the definition of reactive systems by means of generalized Moore machines, i.e., with mixed synchronous data-flow equations and automata [21]. An Heptagon program is modularly structured with a set of *nodes*. Each node corresponds to a reactive behaviour that takes as input and produces as output a set of stream values. The body of a node consists of a set of declarations that take the form of either automata or equations. The equations determine the values for each output, in terms of expressions on inputs' instantaneous values or other flows values.

Figure 2 and Listing 1 show an Heptagon program respectively in graphical and textual representations. The program describes the control of a component lifecycle that can be in either idle (I), waiting (W) or active (A) states. The program takes as input three boolean variables: `r`, which represents a request signal for the component; `c`, which represents an external condition (to be used later on as controllable variable); and `e`, to represent an end signal. It produces as output two boolean values, one that indicates whether the component is active (`a`) the another indicating a start action (`s`). When in the initial state, upon a request signal (i.e., when r is true), the automaton leads to either waiting or active states, depending whether the condition `c` holds. If it does not, it goes first to the waiting state and then to active when `c` becomes true. All the incoming transitions arriving at active state triggers the start action (`s`). From active state, it goes back to idle state upon an end signal.

Listing 1: Textual Representation of Component Lifecycle.

```
node lifecycle(r,c,e:bool) returns(a,s:bool)
let automaton
  state I do
   a=false;s=r & c
   until r & c then A | r & not c then W
  state W do
   a=false;s=c
   until c then A
  state A do
   a=true;s=false
   until e then I
 end;
tel
```

One important characteristic of Heptagon/BZR is the support for hierarchical and parallel automata composition. Figure 3 illustrates an example of hierarchical composition, in which a single state super-automaton embodies the *lifecycle* automaton of Figure 2. It has a self-transition that results in the resetting of the sub-automaton (i.e., *lifecycle*) at every occurrence of signal *b*. A stream of input/output values for this automaton can be seen in Table 1. In particular, we can see at step 9, the resetting of the sub-automaton, which brings it from state active back to idle (at step 10), without any explicit transition. Listing 2 illustrates the parallel composition of two instances of the *delayable* node (and the operator ';'). They run in parallel, in a synchronous way, meaning that one global step corresponds to one local step for every node.




```
node reset(b,r,c,e:bool)
     returns(a,s:bool)
let automaton
 state H do
  (a,s)=lifecycle(r,c,e)
  until b then H
 end;
tel
```

Figure 3: Example of Hierarchical Composition.

Table 1: Execution of the Hierarchical Composition.

step #	1	2	3	4	5	6	7	8	9	10	...
b	0	0	0	0	0	0	0	0	1	0	...
r	0	1	0	0	0	0	1	0	0	0	...
c	0	0	1	0	0	0	1	0	0	0	...
e	0	0	0	0	1	0	0	0	0	0	...
a	0	0	0	1	1	0	0	1	1	0	...
s	0	0	1	0	0	0	1	0	0	0	...

2.2.2. Contracts and Discrete Controller Synthesis

BZR is an extension of Heptagon with specific constructs for Discrete Controller Synthesis (DCS). That makes Heptagon/BZR distinguishable since its compilation may involve formal tools for DCS purposes. A DCS consists in automatically generating a controller capable of acting on the original program to control input variables such that a given temporal property is enforced. In Heptagon/BZR, DCS is achieved by associating a *contract* to a node. A contract is itself a program with two outputs: e_A, an assumption on the node environment; and e_G, a property to be enforced by the node. A set $\{c_1, c_2, \ldots, c_q\}$ of local controllable variables is used for ensuring this objective. Putting it differently, the contract means that the node will be controlled by giving values to $\{c_1, \ldots, c_q\}$ such that given any input flow satisfying assumption e_A, the output will always satisfy goal e_G. When a contract has no controllable variables specified, a verification that e_G is satisfied in the reachable state space is performed by model checking, even if no controller is generated.

Listing 2: Example of Contract in Heptagon/BZR.

```
1 node twocomponents(r1,r2,e1,e2:bool) returns
                    (a1,a2,s1,s2:bool)
2 contract
3 assume true
4 enforce not(a1 and a2)
5 with (c1,c2)
6 let
7  (a1,s1)=lifecycle(r1,c1,e1);
8  (a2,s2)=lifecycle(r2,c2,e2)
9 tel
```

Listing 2 shows an example of contract on a node enclosing a parallel composition of two instances of *lifecycle* (cf. Figure 2). It is composed of three blocks. The *assume* block (line 3), which in this case, states that there is no assumption on the environment (i.e., $e_A = true$). The *enforce* block (line 4) describes the control objective : $e_G = \neg(a1 \wedge a2)$, meaning that both components are mutually exclusive, i.e., they cannot be active at the same time. Lastly, the *with* block (line 5) defines two controllable variables that are used within the node (line 7). In practice they will be given values such that variables `a1` and `a2` are never both true at the same instant.

2.2.3. Compilation and code generation

The Heptagon/BZR compilation chain is as follows: from source code, the Heptagon/BZR compiler produces as output a sequential code in a general-purpose programming language (e.g., Java or C) implementing the control logic, in the form of a step function to be called at each decision in the autonomic loop. At the same time, if the code provided as input contains some contracts, the compiler will also generate a intermediary code that will be given as input to the model checker (e.g., Sigali or Reax), which will, in turn, perform the DCS and produce as output an Heptagon/BZR code corresponding to the generated controller. The latter is then compiled again so as to have an executable code also for the generated controller.

3. Ctrl-F Language

This section presents Ctrl-F, our domain specific language (DSL) for describing control policies. Section 3.1 introduces the Znn.com scenario that will be used throughout the article to illustre the concepts introduced with Ctrl-F. Section 3.2 presents the core concepts of Ctrl-F. Section 3.3 and 3.4 define the notions of behaviour and policy that are specific to Ctrl-F.

3.1. Example Application

Znn.com [14] is an experimental platform for self-adaptive applications, which mimics a news website. As in any web application, Znn.com follows a typical client-server n-tiers architecture, meaning that it relies on a load balancer to redirect requests from clients to a pool of replicated servers. The number of active servers can be regulated in order to maintain a good trade-off between response time and resource utilization. Hence, the objective of Znn.com is to provide news content to its clients/visitors within a reasonable response time, while keeping costs as low as possible and/or under control (i.e., constrained by a certain budget).

There might be times where only the pool of servers is not enough to provide the desired Quality of Service (QoS). For instance, in order to face workload spikes, Znn.com can be forced to degrade the content fidelity so as to require fewer resources to provide the same level of QoS. For this purpose, Znn.com servers are able to deliver news contents in three different ways: (i) with high quality images, (ii) with low quality images, and (iii) with only text. Hence, content fidelity can be seen as another criteria. In summary, the objectives are as follows:

- Keep the performance (in terms of response time) as high as possible;
- Keep content fidelity as high as possible or above a certain threshold;
- Keep the number of active servers as low as possible or under a certain threshold.

In order to achieve them, we may tune:

- The number of active servers and
- The content fidelity of each server.

3.2. Core Concepts

Ctrl-F is our proposal for a architecture description language with high-level constructs for describing reconfiguration behaviours and policies to be enforced all along the execution of the target system.

As depicted in Figure 4, the abstract syntax of Ctrl-F can be divided into two parts: a static one, which is related to the common architectural concepts (components, connections, configurations, etc.); and a dynamic one, which refers to reconfiguration behaviours and policies that must be enforced regardless of the configuration.

The static part of Ctrl-F provides the same core concepts as many existing ADLs (e.g., Fractal [2], Acme [16]). A *component* consists of a set of *interfaces*, a set of *event ports*, a set of *attributes* and a set of *configurations*. *Interfaces* define how a component can interact with other components. They are used to express a required functionality (*client interface*) that may be provided by another *component* and/or to express a provided functionally (*server interface*) that might be used by other *components*. *Event Ports* describe the events, of the given *Event Type*, a *component* is able to emit (*port out*) and/or listen to

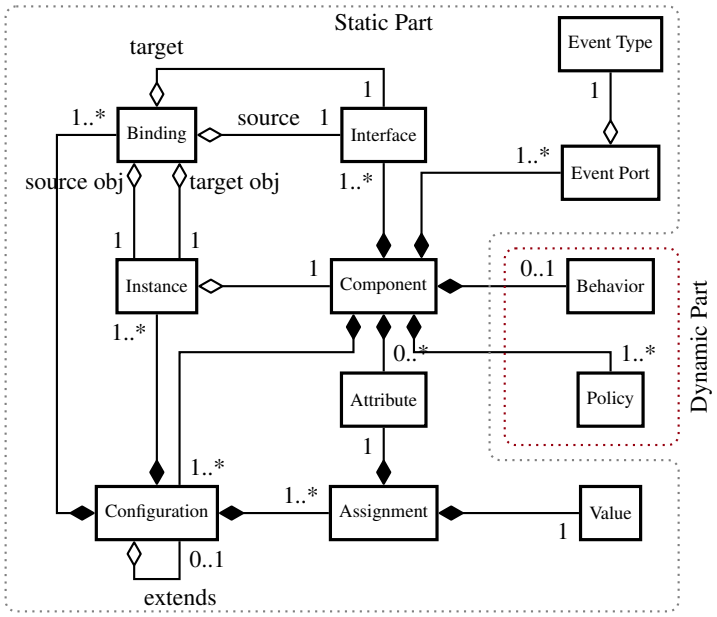


Figure 4: Language Abstract Syntax.

(*port in*). A *configuration* is defined as a set of *instances* of *components*, a set of *bindings* connecting *server* and *client interfaces* of those *instances* (i.e., an assembly), and/or a set of *attribute* assignments to *values*.

The dynamic part consists of a *behaviour* and a set of *policies* that can be defined for each component. A *behaviour* takes the form of orders and conditions (w.r.t. *events* and attribute *values*) under which transitions between configurations (reconfigurations) take place. The *policies* are high-level objectives/-constraints, which may imply the inhibition of some of those transitions.

The Znn.com example application of Section 3.1 can be modeled as a hierarchical composition of four components: *Main*, *Znn*, *LoadBalancer*, and *AppServer*. These components are instantiated according to execution conditions, the system current state (architectural composition), adaptation behaviours and policies defined within each component. Listing 3 shows the textual definition of such components with the static part of Ctrl-F. In addition, to better illustrate the Ctrl-F concepts, the graphical representation of those components is depicted in Figures 5, 6 and 7, where the vertical bars, squares, triangles, and upside-down triangles correspond to interfaces, attributes, port in and out, respectively.

The *Main* component (lines 1-14) encompasses two instances of *Znn*, namely *soccer* and *politics* within a single configuration (lines 7 and 8). The server interfaces of both instances (lines 9 and 10), which provides access to news services, are bound to the server interfaces of the *Main* component (lines 3 and 4) in order for them to be accessed from outside. A policy to be enforced is defined (line 13) and discussed in Section 3.4.

Component *Znn* (lines 16-33) consists of one provided interface (line 18) through which news can be requested. The com-

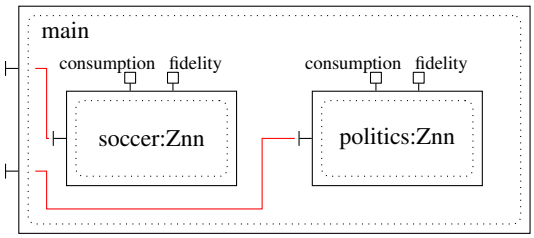


Figure 5: Graphical Representation of *Main* component.

ponent listens to events of types *oload* (overload) and *uload* (underload) (lines 20 and 21), which are emitted by other components. In addition, the component also defines two attributes: *consumption* (line 23), which is used to express the level of consumption (in terms of percentage of CPU) incurred by the component execution; and *fidelity* (line 24), which expresses the content fidelity level of the component.

Three configurations are defined for *Znn* component: *conf1*, *conf2* and *conf3*. *conf1* (lines 26-33) consists of one instance of each *LoadBalancer* and *AppServer* (lines 27 and 28); one binding to connect them (line 29), another binding to expose the server interface of the *LoadBalancer* component as a server interface of the *Znn* component (line 30), and the attribute assignments (lines 31 and 32). The attribute *fidelity* corresponds to the counterpart of instance *as1*, whereas for the *consumption* it corresponds to the sum of the consumptions of instances *as1* and *lb*. *conf2* (lines 34-39) *extends conf1* by adding one more instance of *AppServer*, binding it to the *LoadBalancer* and redefining the attribute values with respect to the just-added component instance (*as2*).

In that case, the attribute fidelity values the average of the counterparts of instances *as1* and *as2* (line 37), whereas for the consumption the same logics is applied so the consumption of the just-added instance is incorporated to the sum expression (line 38). The definition of configuration *conf3* follows the same idea: it extends *conf2* by adding a new instance of *AppServer*, binding it and redefining the attribute values.

Listing 3: Architectural Description of Components *Main*, *Znn*, *Load Balancer* and *AppServer* in Ctrl-F.

```
1  component Main {
2
3    server interface sis
4    server interface sip
5
6    configuration main {
7        soccer:Znn
8        politics:Znn
9        bind sis to soccer.si
10       bind sip to politics.si
11   }
12
13   policy {...}
14 }
15
16 component Znn {
17
18   server interface si
```

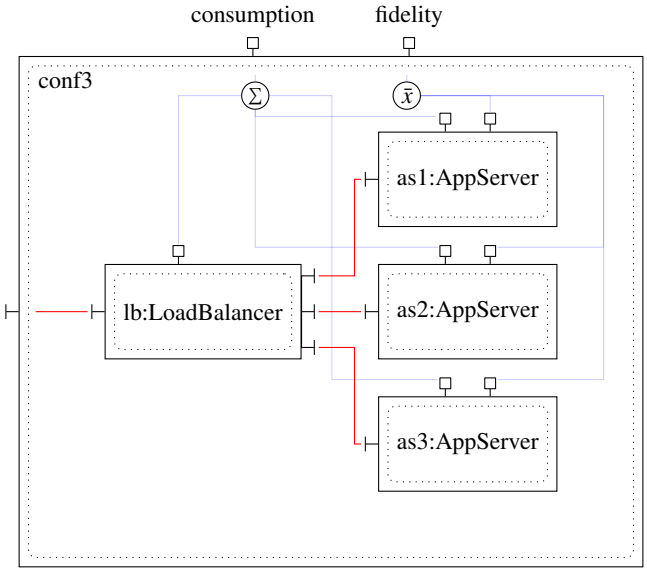


Figure 6: Graphical Representation of *Znn* component.

```
19
20   port in oload
21   port in uload
22
23   attribute consumption
24   attribute fidelity
25
26   configuration conf1 {
27      lb:LoadBalancer
28      as1:AppServer
29      bind lb.ci1 to as1.si
30      bind lb.si to si
31      set fidelity to as1.fidelity
32      set consumption to sum(as1.consumption,
           lb.consumption)
33   }
34   configuration conf2 extends conf1 {
35      as2:AppServer
36      bind lb.ci2 to as2.si
37      set fidelity to avg(as1.fidelity,as2.
           fidelity)
38      set consumption to sum(as1.consumption,
           as2.consumption,lb.consumption)
39   }
40
41   configuration conf3 extends conf2 {...}
42
43   behaviour {...}
44   policy {...}
45 }
46
47 component LoadBalancer {
48   server interface si
49   client interface ci1,ci2,c3
50
51   port out oload
52   port out uload
53
54   attribute consumption=0.2
55 }
```

```
56
57 component AppServer {
58   server interface si
59
60   port in oload
61   port in uload
62
63   attribute fidelity
64   attribute consumption
65
66   configuration text {
67     set fidelity to 0.25
68     set consumption to 0.2
69   }
70   configuration img-ld {
71     set fidelity to 0.5
72     set consumption to 0.6
73   }
74   configuration img-hd {...}
75
76   behaviour {...}
77   policy {...}
78 }
```

Component *LoadBalancer* (lines 47-55) consists of four interfaces: one provided (line 48), through which the news are provided; and the others required (line 49), through which the load balancer delegates each request for balancing purposes. We assume that this component is able to detect overload and underload situations (in terms of number of requests per second) and in order for this information to be useful for other components we define two event *ports* that are used to emit events of type *oload* and *uload* (lines 51 and 52). Like for component *Znn*, attribute *consumption* (line 54) specifies the level of consumption of the component (e.g., 0.2 to express 20% of CPU consumption). As there is no explicit definition of configurations, *LoadBalancer* is implicitly treated as a single-configuration component.

Lastly, the atomic component *AppServer* (lines 57-78) has only one interface (line 58) and listens to events of type *oload* and *uload* (lines 60 and 61). It has also two attributes: fidelity and consumption (lines 63 and 64), just like component *Znn*. Three configurations corresponding to each level of fidelity (lines 66-69, 70-73 and 74) are defined, and the attributes are valuated according to the configuration in question, i.e., the higher the fidelity the higher the consumption.

3.3. Behaviours

A particular characteristic of Ctrl-F is the capability to comprehensively describe behaviours in component-based applications. We mean by behaviour the process in which architectural elements are changed. More precisely, it refers to the order and conditions under which configurations within a component take place.

Behaviours in Ctrl-F are defined with the aid of a high-level imperative language. It consists of a set of behavioural statements (*sub-behaviours*) that can be composed together so as to provide more complex behaviours in terms of sequences of configurations. In this context, a *configuration* is considered as an atomic behaviour, i.e., a behaviour that cannot be decomposed into other *sub-behaviours*. A reconfiguration occurs when the current configuration is terminated and the next one is started. We assume that configurations do not have the capability to directly terminate or start themselves, meaning that they are explicitly requested or ended by behaviour *statements* according to the defined events and policies. Nevertheless, as components are capable to emit events, it would not be unreasonable to define components whose objective is to emit events in order to force a desired behaviour.

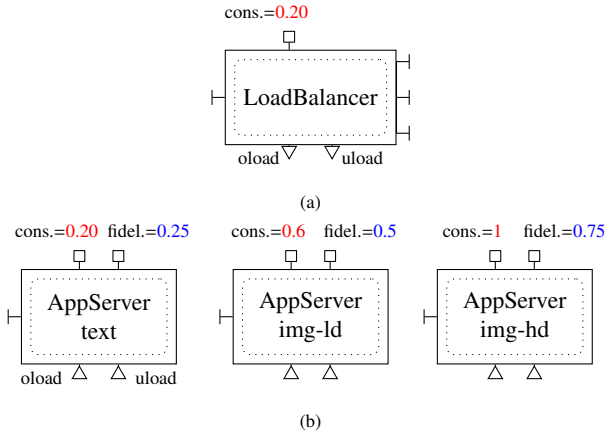


Figure 7: Graphical Representation of (a) *Load Balancer* and (b) *App Server* components.

Table 2: Summary of Behaviour Statements.

Statement	Description
B **when** e_1 **do** B_1, ... , e_n **do** B_n **end**	While executing B when e_i execute B_i
case c_1 **then** B_1, ... , c_n **then** B_n **else** B_e **end**	Execute B_i if c_i holds, otherwise execute B_e
$B_1 \mid B_2$	Execute either B_1 or B_2
$B_1 \parallel B_2$	Execute B_1 and B_2 in parallel
do B **every** e	Execute B and re-execute it at every occurrence of e

3.3.1. Statements

Table 2 summarizes the behaviour statements of the Ctrl-F behavioural language. During the execution of a given behaviour B, the *when-do* statement states that when a given event of type e_i occurs the configuration(s) that compose(s) B should be terminated and that (those) of the corresponding behaviour B_i are started.

The *case-then* statement is quite similar to *when-do*. The difference resides mainly in the fact that a given behaviour B_i is executed if the corresponding condition c_i holds (e.g., conditions on attribute values), which means that it does not wait for

a given event to occur. In addition, if none of the conditions holds ($c_1 \wedge ... \wedge c_n = 0$), a default behaviour ($B_e$) is executed, which forces the compiler to choose at least one behaviour. The *parallel* statement states that two behaviours are executed at the same time, i.e., at a certain point, there must be two independent branches of behaviour executing in parallel. This construct is also useful in the context of atomic components like *AppServer*, where we can, for instance, define configurations composed of orthogonal attributes like fidelity and font size/color (e.g., `text || font-huge`).

The *alternative* statement allows to describe choice points among configurations or among more elaborated sequential behaviour statements. They are left free in local specifications and will be resolved in upper level assemblies, in such a way as to satisfy the stated policies, by controlling these choice points appropriately. Finally, the *do-every* statement allows for execution of a behaviour *B* and re-execution of it at every occurrence of an event of type *e*. It is noteworthy that behaviour *B* is preempted every time an event of type *e* occurs. In other words, the configuration(s) currently activated in *B* is (are) terminated, and the very first one(s) in *B* is (are) started.

3.3.2. Example in Znn.com

We now illustrate the use of the statements we have introduced to express adaptation behaviours for components *AppServer* and *Znn* of the *Znn.com* case study. The expected behaviour for component *AppServer* is to pick one of its three configurations (*text*, *img-ld* or *img-hd*) at every occurrence of events of type *oload* or *uload*. To that end, as it can be seen in Listing 4, the behaviour can be decomposed in a *do-every* statement, which is, in turn, composed of an *alternative* one. It is important to mention that the decision on one or other configuration must be taken at runtime according to input variables (e.g., income events) and the stated policies, that is, there must be a control mechanism for reconfigurations that enforces those policies. We come back to this subject in Section 4.

Listing 4: AppServer's Behaviour.

```
1 component AppServer {  ...
2   behaviour {
3     do
4       text | img-ld | img-hd
5     every (oload or uload)
6   }
7 }
```

Regarding component *Znn*, the expected behaviour is to start with the minimum number of *AppServer* instances (configuration *conf1*) and add one more instance, i.e., leading to configuration *conf2*, upon an event of type (*oload*). From *conf2*, one more instance must be added, upon an event of type *oload* leading to configuration *conf3*. Alternatively, upon an event of type *uload*, one instance of *AppServer* must be removed, which will lead the application back to configuration *conf1*. Similarly, from configuration *conf3*, upon a *uload* event, another instance must be removed, which leads the application to *conf2*. It is notorious that this behaviour can be easily expressed by an automaton, with three states (one per configuration) and four transitions (triggered upon the occurrence of *oload* and *uload*). However, Ctrl-F is designed to tackle the adaptation control problem in a higher level, i.e., with process-like statements over configurations.

For these reasons, we describe the behaviour with two embedded *do-every* statements, which in turn comprise each a *when-do* statement, as shown in Listing 5 (lines 6-14 and 8-12). We also define two auxiliary configurations: *emitter1* (line 2) and *emitter2* (line 3), which extend respectively configurations *conf2* and *conf3*, with an instance of a pre-defined component *Emitter*. This component does nothing but emit a given event (e.g., *e1* and *e2*) so as to force a loop step and thus go back to the beginning of the *when-do* statements. The main *do-every* statement (lines 6-14) performs a *when-do* statement (lines 7-13) at every occurrence of an event of type *e1*. In practice, the firing of this event allows going back to *conf1* regardless of the current configuration being executed. *conf1* is executed until the occurrence of an event of type *oload* (line 7), then the innermost *do-every* statement is executed (lines 8-12), which in turn, just like the other one, executes another *when-do* statement (lines 9-11) and repeats it at every occurrence of an event of type *e2*. Again, this structure allows the application to go back to configuration *conf2*. Configuration *conf2* is executed until an event of type either *oload* or *uload* occurs. For the former case (line 9), another *when-do* statement takes place, whereas for the latter (line 10) configuration *emitter1* is the one that takes place. Essentially, at this point, an instance of component *Emitter* is deployed along with *conf2*, since *emitter1* extends *conf2*. As a consequence, this instance fires an event of type *e1*, which forces the application to go back to *conf1*. The innermost *when-do* statement (line 9) consists in executing *conf3* until an event of type *uload* occurs, then configuration *emitter2* takes place, which makes an event of type *e2* be fired in order to force going back to *conf2*.

It is important to notice that this kind of construction allows to achieve the desired behaviour while sticking to the language design principles, that is, high-level process-like constructs and configurations. It also should be remarked that while in Listing 5, we present an imperative approach to forcibly increase the number of *AppServer* instances upon *uload* and *oload* events, in Listing 4 we let the compiler choose the most suitable fidelity level according to the runtime events and conditions. Although there is no straightforward guideline, an imperative approach is clearly more suitable when the solution is more sequential and delimited, whereas as the architecture gets bigger, in terms of configurations, and less sequential, then a declarative definition becomes more interesting.

3.4. Policies

Policies are expressed with high-level constructs for constraints on configurations, either temporal or on attribute values. In general, they define a subset of all possible global configurations, where the system should remain invariant: this will be achieved by using the choice points in order to control the reconfigurations. An intuitive example is that two component instances in parallel branches might have each several possible configurations, and some of them have to be kept exclu-

Listing 5: Znn's Behaviour.

```
component Znn {...
  configuration emitter1 extends conf2 { e:Emitter }
  configuration emitter2 extends conf3 { e:Emitter }

  behaviour {
    do
      conf1 when oload do
        do
         conf2 when oload do (conf3 when uload do emitter2 end),
                      uload do emitter1
         end
        every e2
      end
    every e1
  }
}
```

sive. This exclusion can be enforced by choosing the appropriate configurations when starting the components.

3.4.1. Constraints/Optimization on Attributes

This kind of constraints are predicates and/or primitives of optimization objectives (i.e., maximize or minimize) on component attributes. Listing 6 illustrates some constraints and optimization on component attributes. The first two policies state that the overall fidelity for component instance *soccer* should be greater or equal to 0.75, whereas that of instance *politics* should be maximized. Putting it differently, instance *soccer* must never have its content fidelity degraded, which means that it will have always priority over *politics*. The third policy states that the overall consumption should not exceed 5, which can be interpreted as a constraint on the physical resource capacity, e.g., the number of available machines or processing units.

Listing 6: Example of Constraint and Optimization on Attributes.

```
component Main { ...
 policy { soccer.fidelity >= 0.75 }
 policy { maximize politics.fidelity }
 policy { (soccer.consumption +
           politics.consumption) <= 5 }
}
```

3.4.2. Temporal Constraints

Temporal constraints are high-level constructs that take the form of predicates on the order of configurations. These constructs might be very helpful when there are many possible reconfiguration paths (by either *parallel* or *alternative* composition, for instance), in which case the manual specification of such constrained behaviour may become a very difficult task.

To specify these constraints, Ctrl-F provides four constructs, as follows:

- $conf_1$ **precedes** $conf_2$: $conf_1$ must take place right before $conf_2$. It does not mean that it is the only one, but it should be among the configurations taking place right before $conf_2$.
- $conf_1$ **succeeds** $conf_2$: $conf_1$ must take place right after $conf_2$. Like in the precedes constraint, it does not mean that it is the only one to take place right after $conf_2$.
- $conf_1$ **during** $conf_2$: $conf_1$ must take place along with $conf_2$.
- $conf_1$ **between** $(conf_2, conf_3)$: once $conf_2$ is started, $conf_1$ cannot be started and $conf_3$, in turn, cannot be started before $conf_2$ terminates.

Listing 7 shows an example of how to apply temporal constraints, in which it is stated that configuration *img-ld* comes right after the termination of either configuration *text* or configuration *img-ld*. In this example, this policy avoids abrupt changes on the content fidelity, such as going directly from text to image high definition or the other way around. Again, it does not mean that no other configuration can take place along with *img-ld*, but the *alternative* statement in the behaviour described in Listing 4 leads us to conclude that only *img-ld* must take place right after either *text* or *img-hd* has been terminated.

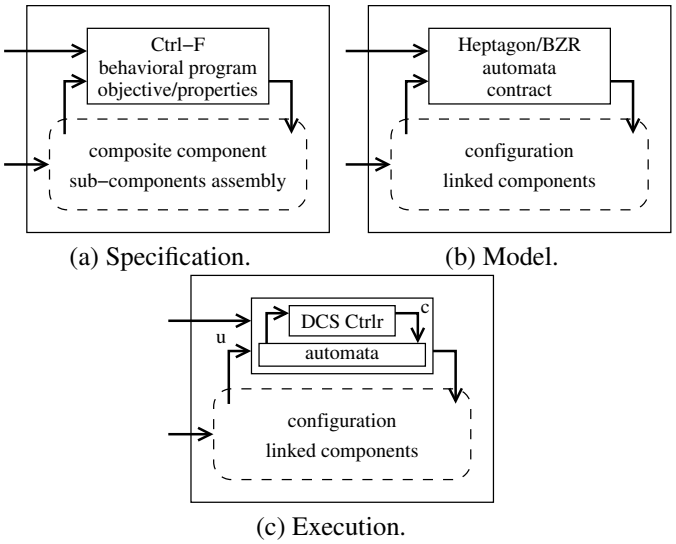
Listing 7: Example of Temporal Constraint.

```
component AppServer { ...
 policy { img-ld succeeds text }
 policy { img-ld succeeds img-hd }
}
```

4. Heptagon/BZR Model

4.1. Approach Overview

Our approach consists in seamlessly conceiving autonomic component-based applications by relying on a high-level behavioural description. The principle is to have an autonomic manager (AM) embodying a feedback control loop within each component. The manager takes decisions in response to occurred events, while taking into consideration the current/past configurations, a behavioural program, and determining as result which configurations have to be terminated and which ones



(a) Specification. (b) Model.

(c) Execution.

Figure 8: Approach Overview.

have to be started. We rely on Ctrl-F to specify: (i) behaviours in a process-like manner (in terms of sequences, alternative/-conditional/parallel branches and loops of configurations); and (ii) policies, which take the form of properties that have to be kept invariant regardless of the configuration, as depicted in Figure 8(a).

The behavioural program defined by Ctrl-F provides the AM an extra level of knowledge on the possible futures of the component configuration: that is, it enables the AM to explore the space of reachable configurations so as to avoid branches that may lead, in the future, to configurations violating the stated policies. To that end, we provide a set of translation schemes allowing for the automatic translation from a Ctrl-F description to the reactive language Heptagon/BZR and thereby benefiting from DCS. an Heptagon/BZR automaton and contract corresponding to the behavioural program and policies will be associated to each component under control, as can be seen in Figure 8(b). The Heptagon/BZR program, once equipped with contracts, allows us to either perform formal verification on the behavioural program with respect to the policies; and/or to obtain, via DCS, a correct-by-construction controller (cf. Figure 8 (c)). That is to say that the generated controller will be capable of controlling the automaton that models the component behaviour so as to prevent it going in branches leading to bad states (i.e., configurations that violate the policies). This process, from the Ctrl-F description to the Heptagon/BZR translation, is detailed next.

4.2. General FSA Model Structure

The component is the core of Ctrl-F description and can be modeled as an Heptagon/BZR node, as shown in Figure 9. The node takes as input external request (`r`) and end notification (`e`) signals, and a set of events $\{v_1, \ldots, v_k\}$, which corresponds to the event types the component in question (`comp`) listens to. As output, it produces a set of request (resp. end) signals $\{r_1, \ldots, r_m\}$ (resp. $\{e_1, \ldots, e_m\}$) for each configuration $conf_i$, for $i \in [1, m]$, defined within the concerned component. In addition, it also returns a set of weights $\{w_1, \ldots, w_l\}$, for the attribute valuation for each attribute in the component. The main node (`comp` in Figure 9) may contain a contract in which a set of controllable variables $\{c_1, \ldots, c_q\}$ (in the case there is any choice point such as a behaviour with an alternative statement) and the reference to the set of stated policies ($\{p_1, \ldots, p_t\}$) in order for them to be enforced by the controller resulting from the DCS. The details on how policies are translated are given in Section 4.4.

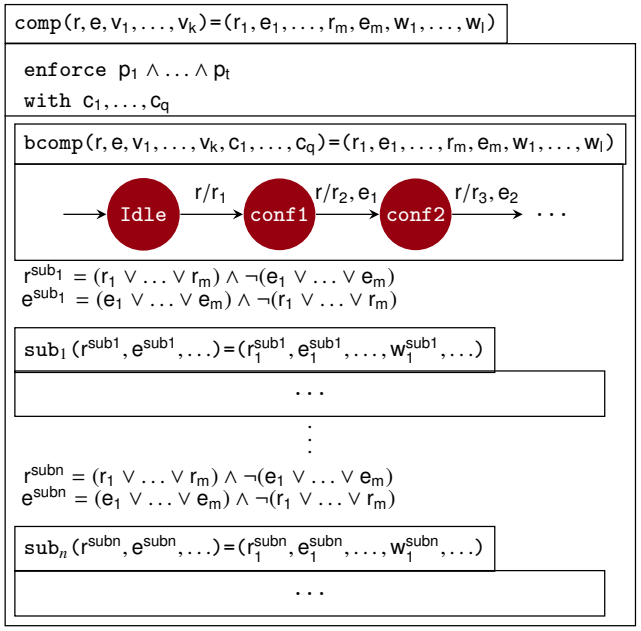


Figure 9: Translation Scheme Overview.

Component behaviours are modeled as a sub-node (`bcomp` in Figure 9), which consists of an automaton describing the order and conditions under which configurations take place. For this purpose, it gets as input the same request (`r`), end (`e`) and event ($\{v_1, \ldots, v_k\}$) signals of the main node. As a result of the reaction to those signals, it produces the same signals for requesting ($\{r_1, \ldots, r_m\}$) and ending ($\{e_1, \ldots, e_m\}$) configurations as the weights ($\{w_1, \ldots, w_l\}$) corresponding to the attributes valuation in the current state (configuration) of the behaviour. We provide further details on the translation of behavioural statements in Section 4.3. Lastly, there might also be some other sub-nodes ($\{sub_1, ..., sub_n\}$) referring to components instantiated within the concerned component, i.e., `comp`. They have interfaces and contents which are structurally identical to those of the main node. That is to say, that sub-nodes may have, in turn, a contract, a behaviour sub-node and a sub-node per component instance defined inside it. It is noteworthy that the request (r^{subi}) and end (e^{subi}) signals for a sub-component $sub_i \in \{sub_1, \ldots, sub_n\}$ are defined as equations of request and end signals. $\{r_1, \ldots, r_m\}$ and $\{e_1, \ldots, e_m\}$ are respectively the sets of request and end signals for the configurations $conf_1, \ldots, conf_m$ to which component sub_i belongs. That means that a sub-component sub_i will be requested if any configuration it belongs to is also requested ($r_1 \vee \ldots \vee r_m$) and none of them

is terminated $\neg(e_1 \vee \ldots \vee e_m)$, which avoids emitting a request signal for an already active component. The same applies for its termination.

Listing 8 shows an excerpt of Heptagon/BZR model for components *Znn* (lines 5-21) and *AppServer* (lines 1-3). For node *appserver*, besides the request and end signals, it gets as inputs the events of type `oload` and `uload` (line 1). As output (line 2), it produces request and end signals for configurations text (`r_text` and `e_text`), img-ld (`r_ld` and `e_ld`) and img-hd (`r_hd` and `e_hd`), apart from weights, i.e., attribute valuations (`fidelity` and `consumption`). Node `znn` has a very similar interface as *appserver*, except that it produces as output request and end signals for configurations *conf1* (`r_conf1` and `e_conf1`), *conf2* (`r_conf2` and `e_conf2`) and *conf3* (`r_conf3` and `e_conf3`). Regarding its body (lines 8-20), `znn` comprises one instance of the node that models the behaviour (`bznn`, line 15) and three instances of node `appserver` (lines 16-18). The request and end signals for these instances can be derived from the request and end signals for configurations (lines 8-13). At last, attributes are values based on the values of attributes of the instances of node `appserver` (line 19).

4.3. Behaviours

For each program in Ctrl-F, we need to construct a FSA model, in Heptagon/BZR, of all its possible behaviours. We translate each behaviour statement defined inside another behaviour as sub-automaton, hierarchically decomposing the whole behaviour into smaller pieces, down to a configuration.

4.3.1. The top-most behaviour

The top-most automaton i.e., the automaton modeling the whole behaviour consists of a two-state model, as depicted in Figure 10 (a). The automaton is in state *Idle* when the component does not take part in the current configuration. Upon a request signal (r), it goes to *Active* state, from where it can go back again to *Idle* state again upon an end signal (e). *Active* state accommodates a behaviour statement itself, which is itself modeled as a sub-automaton of state A.

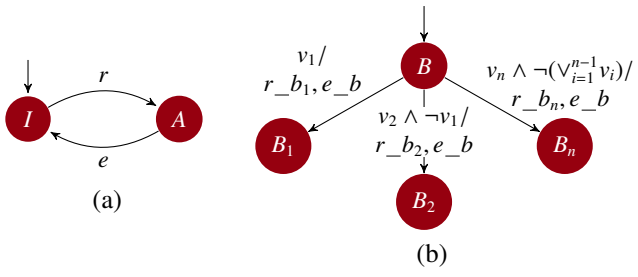


Figure 10: FSA Modeling: (a) Lifecycle; (b) *When-Do*.

4.3.2. Statements

The automaton that models the statement *when-do* (cf. Figure 10(b)) consists of an initial state B corresponding to the first behaviour statement to be executed. The automaton goes to state B_i (corresponding to the execution of the next behaviour) upon a signal (event) v_i while producing signals for requesting the initiation of to the next behaviour (r_b_i) and the termination (e_b) the current one (for $1 \leq i \leq n$). It is important to notice that upon two events at the same time, a priority is given according to the order behaviours are declared. For instance, if v_1 and v_2 triggers, respectively, behaviours B_1 and B_2, then B_1 will be triggered if declared before B_2.

Both behaviour statements *case* and *alternative* can be modeled by the automaton shown in Figure 11. As the sub-behaviour statements should be executed at the very first instant upon the request of the *case* or *alternative* statement, the automaton must be composed in parallel with the automaton modeling the main behaviour (inside node `bcomp`, in Figure 9). Hence, a *case* or an *alternative* statement is modeled as a simple state inside the (super) automaton in the hierarchy that models the main behaviour. Upon a request to those statements (signal r), the main automaton emits a request signal r' that will trigger a transition from state W to the next state (B_1 or B_2) according to variable c. Then it can go either to another behaviour, if another r' is emitted and c states so; or back to W if an end signal (e') is emitted.

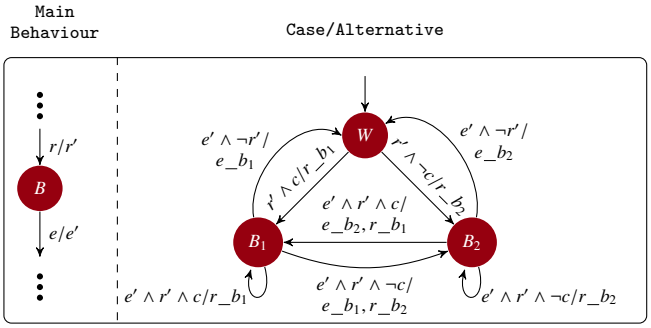


Figure 11: Parallel Composition of Automata Modeling the Main Behaviour and the *Case/Alternative* Statement.

There are two differences between the use of this automaton for a *case* or an *alternative* statement. First, for the *case* statement, several (i.e., more than two) branches are allowed, so there might be more states ($B_1, B_2, \ldots, B_n$) referring to each branch as well as their corresponding conditions $c_1, c_2, ..., c_n$, which was omitted here for readability reasons. Second, for the *alternative* statement, the conditions c_i will be considered as *controllable* variables in Heptagon/BZR. Thus, a DCS should be performed to guarantee that the stated policies are not violated.

The automaton model for the *do-every* statement is shown in Figure 12(a). It consists of a single-state automaton, which means that it starts by directly executing statement B. It has a self-transition at every occurrence of signal s, while emitting end (e_b) and request (r_b) signals, that is, statement B is re-executed at every occurrence of event s. Finally, Figure 12(b) presents the model for the *Parallel* statement: simply in the parallel composition of sub-automata.

Listing 8: Heptagon/BZR code for *Znn* and *AppServer*.

```
1  node appserver(r,e,oload,uload:bool) returns
2                (r_text,e_text,r_ld,e_ld,r_hd,e_hd:bool;fidelity,consumption:int)
3  let ... tel
4
5  node znn(r,e,oload,uload:bool) returns
6          (r_conf1,e_conf1,...,r_conf3,e_conf3:bool;fidelity,consumption:int)
7  let
8    r_as1 = r_conf1 or r_conf2 or r_conf3 and not(e_conf1 or e_conf2 or e_conf3);
9    r_as2 = r_conf2 or r_conf3 and not(e_conf2 or e_conf3);
10   r_as3 = r_conf3 and not(e_conf3);
11   e_as1 = e_conf1 or e_conf2 or e_conf3 and not(r_conf1 or r_conf2 or r_conf3);
12   e_as2 = e_conf2 or e_conf3 and not(r_conf2 or r_conf3);
13   e_as3 = e_conf3 and not(r_conf3);
14
15   (r_conf1,e_conf1,...) = bznn(r,e,oload,uload);
16   (r_text_as1,...,fid_as1,conso_as1) = appserver(r_as1,e_as2,oload,uload);
17   ...
18   (r_text_as3,...,fid_as3,conso_as3) = appserver(r_as3,e_as3,oload,uload);
19   consumption = conso_as1 + conso_as2 + conso_as3;
20   ...
21 tel
```

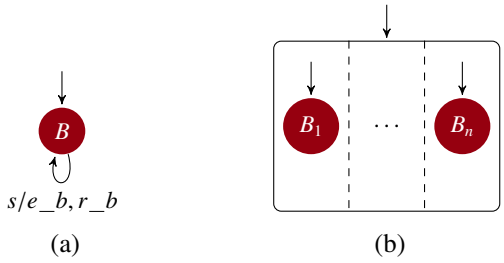


Figure 12: Automata Modeling: (a) the *Every* and (b) *Parallel* statements.

4.3.3. Znn.com Example

Figure 13 illustrates the translation for the *AppServer* component behaviour defined in Listing 4. It consists of a parallel composition of two automata: one to model the behaviour itself (on the left-hand side), and another to model the *alternative* sub-behaviour statement (on the right-hand side). The first automaton corresponds to the top-most automaton, as the one shown in Figure 10(a). The active state comprises a sub-automaton representing the *do-every* statement, which starts by state *B* and restarts it at every occurrence of events *oload* (overload) or *uload* (underload) while emitting at the same time request and end signals (r_b and e_b, respectively). The request signal (r_b) is used by the second automaton in order to enable transitions to states representing configurations (*txt*, *ld* and *hd*) according to the controllable variables c_1 and c_2, while emitting proper request signals (r_txt or r_ld) for the next configurations and end signals (e_txt or e_ld) for the current one. The end signal (e_b), on the other hand, is used to enable transitions to other or even the same configuration, in the presence of the request signal, or to the waiting state *W*, in the absence of the request signal. It should be mentioned that due to the lack of space, we omitted the outgoing and incoming transitions of state *hd* (configuration *img-hd*). In the generated executable

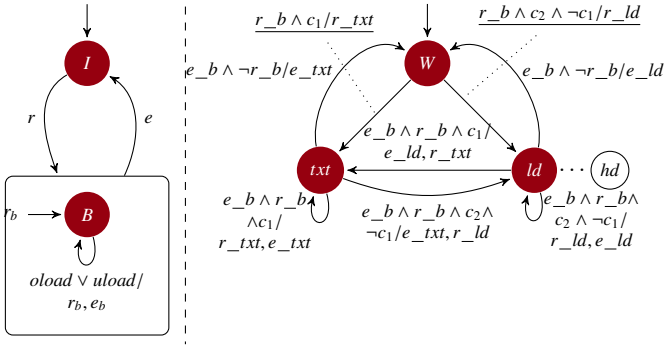


Figure 13: Translation of the component *AppServer* behaviour.

code, the output of those automata will be connected to pieces of code dedicated to trigger the actual reconfigurations. For instance, the presence of signals r_ld and e_txt will trigger the reconfiguration script that changes the content fidelity of given component from *text* to *img-ld* (cf. Section 5).

4.4. Policies

4.4.1. Constraints/Optimization on Attributes

For illustration, Listing 9 shows how the last policy of the *Main* component (Listing 6, line 4) is translated into Heptagon/BZR. This constraint is defined as an equation (line 8) that depends on the integer outputs `soccer_consumption` and `politics_consumption`, which are produced by the respective instances of node `znn` (lines 6 and 7). This equation is hence used in the enforce block of the contract (line 3). Although the declaration of optimization objectives are currently not supported by Heptagon/BZR, one may model a one-step optimization directly within the DCS tools Heptagon/BZR relies on [11] [22]. Please see [23] for more details.

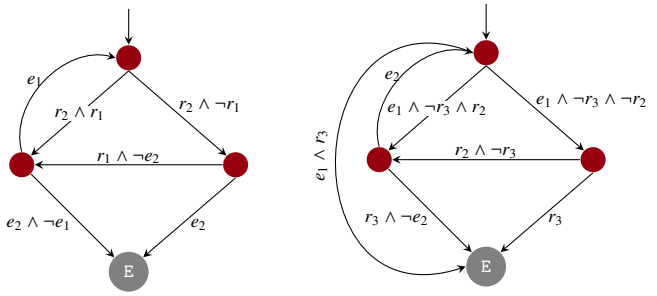
Listing 9: Example of Constraint on Attribute in Heptagon/BZR.

```
node main(r,e:bool;...) returns(...,p1:bool)
contract
enforce p1 and ...
with (...)
let ...
  (...,soccer_consumption)=znn(...);
  (...,politics_consumption)=znn(...);
  p1=(soccer_consumpton +
      politics_consumption) <= 5
... tel
```

4.4.2. Temporal Constraints

Temporal constraints refer to constraints on the logical order of configurations. They are modeled in Heptagon/BZR by a set of boolean equations of request (r) and end (e) signals that are emitted by automata modeling behaviours. For simple constraints like `conf1 succeeds conf2` (resp. `conf1 precedes conf2`), just a predicate like $e\_conf2 \Rightarrow r\_conf1$ (resp. $e\_conf1 \Rightarrow r\_conf2$) suffices. However, whenever there is a need for keeping track of the sequence of signals (to request and/or end configurations), the use of observer automata is needed. Observer automata are placed in parallel with the behavior automata, and generated in Heptagon/BZR as part of the contract. The principle is to have an automaton that observes the sequence of signals that leads to a policy violation and state that the state resulting from that sequence (an "error" state) should never be reached. Again, here we can rely on the enforce block of a Heptagon/BZR contract. The DCS objective is the invariance of the state set deprived of those where the variable error is true.

Figure 14(a) depicts an observer that models the policy during (`conf1 during conf2`), where r_1 and r_2 (resp. e_1 and e_2) correspond to the request (resp. end) signal for configurations `conf1` and `conf2`, respectively. The error state (E) is reached if `conf2` terminates before `conf1` ($e_2 \wedge \neg e_1$) or if `conf2` terminates before `conf1` has started. The observer that models the constraint between (`conf1 between (conf2,conf3)`) is depicted in Figure 14(b). Similarly, r_1, r_2 and r_3 (resp. e_1, e_2 and e_3) correspond to the request (resp. end) signal for configurations `conf1`,`conf2` and `conf3`, respectively. The automaton goes to the error state (E) whenever configuration `conf3` is started (r_3 is emitted) after configuration `conf2` (e_2), except when configuration `conf1` is started and terminated (r_1 and e_1) in the between.



(a) During (b) Between

Figure 14: Observer Automata for Temporal Constraints.

5. Implementation

5.1. FraSCAti and Service Component Architecture

Despite the fact that our contribution is technology-agnostic, for the sake of proof-of-concept, we rely on the Service Component Architecture (SCA) [1] as target component model. SCA is a component model for building applications based on the Service Oriented Architecture principles. SCA provides means for constructing, assembling and deploying software components regardless of the programming language or protocol used to implement and make them communicate. Figure 15 depicts the basic concepts of SCA model illustrated with the Znn.com example. A component can be defined as simple or composite, that is, composed of other components. A simple component is defined by an implementation, a set of services, references and properties. The implementation points to the actual implementation of the component (e.g., a Java Class). A service (resp. reference) refers to a business function provided (resp. required) by the component and is specified by an interface (e.g., via a Java Interface). Properties are attributes defined within components whose values can be set/got from outside the component. In order for services, references and properties be accessible from/or access outside the composite, they should be promoted to composite (or external) services/references/properties. Bindings define which methods and/or transports (e.g., HTTP, SOAP [2]) are allowed to access services or to be accessed by references. Lastly, service and reference are connected by wires.

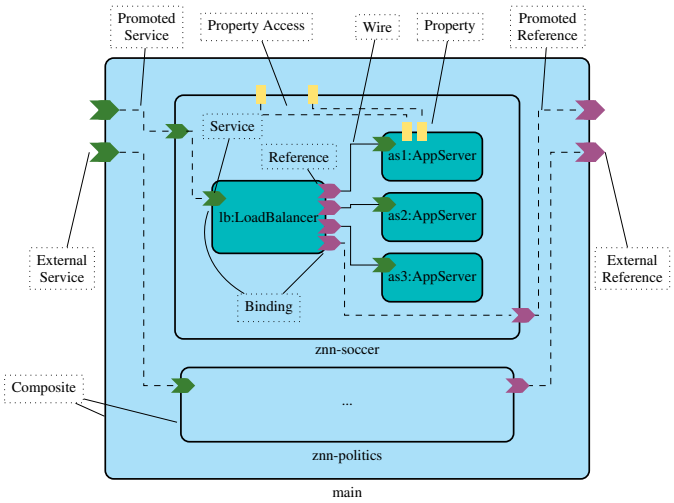


Figure 15: Service Component Architecture Concepts in Znn.com.

The objective is that a middleware platform implementing the SCA specifications takes care of component implementation, interoperability and communication details, so architects and developers can focus only on the architecture. In this work, we rely on the Java-based SCA middleware FraSCAti [4], since

[1] `http://www.oasis-opencsa.org/`
[2] `http://www.w3.org/TR/soap/`

it provides mechanisms for runtime reconfiguration of SCA application. The FraSCAti Runtime is itself conceived relying on the SCA model, that is, it consists of a set of SCA components that can be deployed *a la carte*, according to the user's needs. For instance, one can instantiate the *frascati-fscript* component, which provides services allowing for the execution of an SCA-variant of FPath/FScript [5], a domain-specific language for introspection and dynamic reconfiguration of Fractal components.

5.2. Compilation Tool-chain

As can be seen in Figure 16, the compilation process can be split into two parts: (i) the reconfiguration logics and (ii) the behaviour/policy control and verification. The reconfiguration logics is implemented by the *ctrlf2fscript* compiler, which takes as input a Ctrl-F definition and generates as output a FraSCAti FPath/FScript (1) containing a set procedures allowing going from one configuration to another. To that end, we rely on existing differencing/match algorithms for object-oriented models [24].

The script is later on used by a FraSCAti component (*Manager*) in charge of controlling the actual running component software system (2). Listing 10 shows an example of script describing a reconfiguration from configuration *conf1* to *conf2* of *Znn* component. In this example, the script just wires the reference of instance *lb:LoadBalancer* to the *as2:AppServer* instance's service, then it starts the instance *as2:AppServer*.

Listing 10: Reconfiguration Logics in FScript.

```
1 action conf1_conf2(znn){
2   stop($znn/scachild::lb);
3   r=$znn/scachild::loadbalancer/scareference
        ::as2;
4   s=$znn/scachild::as2/scareference::s;
5   addscawire($r,$s);
6   start($znn/scachild::as2);
7   start($znn/scachild::lb);
8 }
```

The behaviour control and verification is performed by the *ctrlf2ept* compiler, which takes as input a Ctrl-F definition and provides as output a synchronous reactive program in Heptagon/BZR (3). This code is given as input to the Heptagon/BZR compiler, which produces a code for the model checking and discrete controller synthesis tool (4). Heptagon/BZR is currently integrated with ReaX [22] tool. Thus, if the Heptagon/BZR code translated from the Ctrl-F description contains a contract with controllable variables, the tool generates a controller (if there is any) such that the stated properties are guaranteed (5). Conversely, if there is no need for controller synthesis, the corresponding tool simply verifies the correctness of the Heptagon/BZR program. The Heptagon/BZR program corresponding to the target system model (3) along with the generated controller (5), if that is the case, are compiled again with the Heptagon/BZR compiler.

The result of the compilation of an Heptagon/BZR code is a sequential code (6) in a general-purpose programming language (in our case Java) comprising two methods: `reset` and `step`.

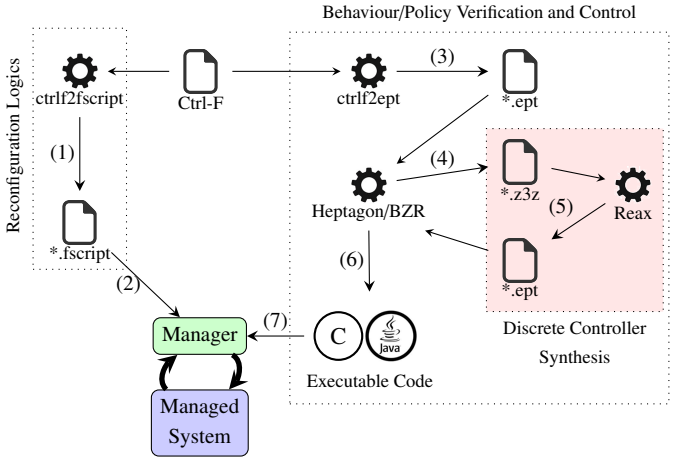


Figure 16: Ctrl-F Compilation Chain.

The former initializes the internal state of the program, whereas the latter is executed at each logical step to compute the output values based on a given vector of input values and the current state.

These methods are encapsulated by the FraSCAti component that controls the managed system (7) and they are typically used by first executing `reset` and then by enclosing `step` in an infinite loop, in which each iteration corresponds to a reaction to an event (e.g., *oload* or *uload*), as sketched in Listing 11. The step method returns a set of signals corresponding to the start or stop of configurations (line 4). From these signals, we can find the appropriate script that embodies the reconfiguration actions to be executed (lines 5 and 6).

Listing 11: Control Loop Sketch.

```
1 reset();
2 ...
3 on event oload or uload
4 <...,stop_conf1,start_conf2,...>=step(oload,
      uload);
5 reconfig_script=find_script(...,stop_conf1,
      start_conf2,...);
6 execute(reconfig_script);
```

5.3. Wrapping the Compilation Result into SCA Components

We wrap the control loop logics into three components, which are enclosed by a composite named *Manager*. Component *EventHandler* exposes a service allowing itself to be sent events (e.g., *oload* and *uload*). The method implementing this service is defined as non-blocking so the incoming events are stored in a First-In-First-Out queue. Upon the arrival of an event coming from the *Managed System* (e.g., Znn.com), component *EventHandler* invokes the step method, implemented by component *Architecture Analyzer*. The step method output is sent to component *Reconfigurator*, that encompasses a method to find the proper reconfiguration script to be executed.

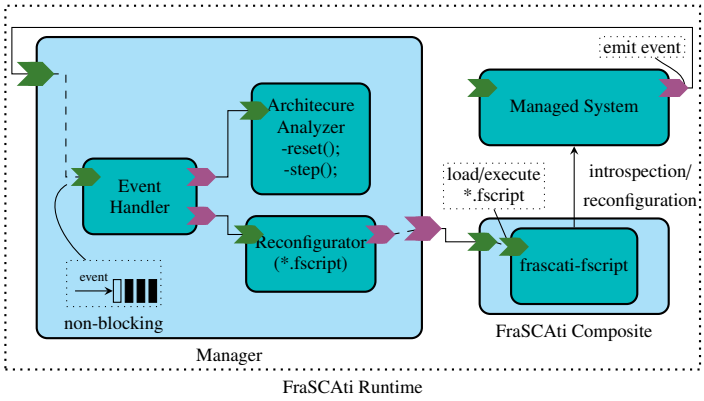


Figure 17: Manager Prototype Wrapping the Control Loop.

6. Case Studies

This section shows the application of Ctrl-F in two different situations. We first present an adaptive scenario by simulating the *Znn.com* case study, whose Ctrl-F model has been already detailed throughout the previous section. Then when provide a second case study, in which we apply Ctrl-F in order to control an application with a workflow of mutually exclusive tasks.

6.1. Case Study 1: Znn.com

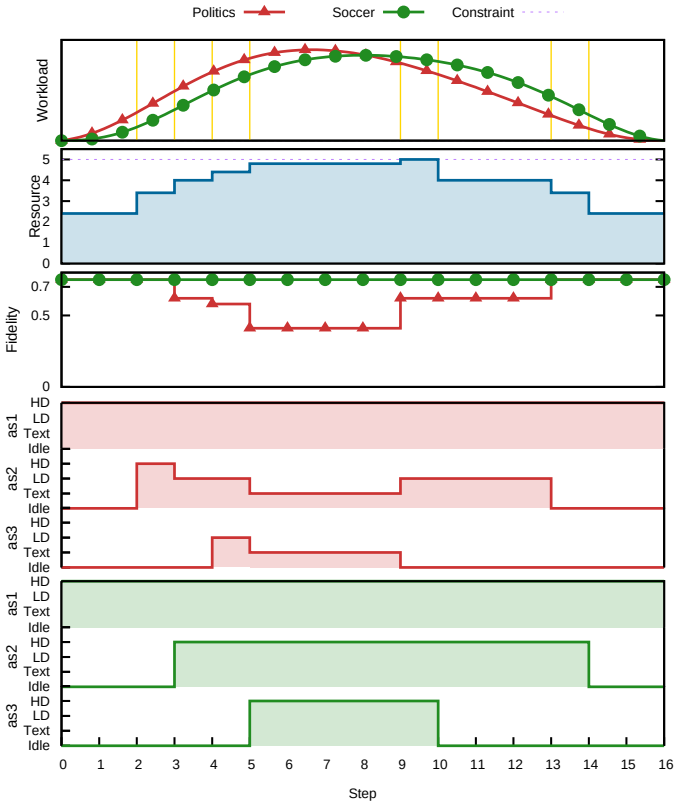


Figure 18: Execution of the *Znn.com* Adaptation Scenario.

We simulate the execution of the two instances of *Znn.com* application, namely *soccer* and *politics*, under the administration of the *Manager* presented in previous section, to observe the control of reconfigurations taking into account a sequence of input events. The behaviours of components *AppServer* and *Znn* are stated in Listings 4 and 5, respectively, while policies are defined in Listing 6 and 7.

As it can be observed in the first chart of Figure 18, we scheduled a set of overload (*oload*) and underload (*uload*) events (vertical dashed lines), which simulate an increase followed by a decrease of the income workload for both soccer and politics instances. The other charts correspond to the overall resource consumption, the overall fidelity, and the fidelity level (i.e., configurations *text*, *img-ld* or *img-hd*) of the three instances of component *AppServer* contained in both instances of component *Znn*.

As the workload of *politics* increases, an event of type *oload* occurs at step 2. That triggers the reconfiguration of that instance from *conf1* to *conf2*, that is, one more instance of *AppServer* is added within the *Znn*instance *politics*. We can observe also the progression in terms of resource consumption, as a consequence of this configuration. The same happens with *soccer* at step 3, and is repeated with *politics* and *soccer* again at steps 4 and 5. The difference, in this case, is that at step 4, the *politics* instance must reconfigure (to *conf3*) so as to cope with the current workload while keeping the overall consumption under control. In other words, it forces the *AppServer* instances *as2* and *as3* to degrade their fidelity level from *img-hd* to *img-ld*. It should be highlighted that although at least one of the *AppServer* instances (*as2* or *as3*) could be at that time at maximum fidelity level, the knowledge on the possible future configurations guarantees the maximum overall fidelity for instance *soccer* to the detriment of a degraded fidelity for instance *politics*, while respecting the temporal constraints expressed in Listing 7. Hence, at step 5, when the last *oload* event arrives, the fidelity level of *soccer* instance is preserved by gradually decreasing that of *politics*, that is, both instances *as2* and *as3* belonging to the *politics* instance are put in configuration *text*, but without jumping directly from from *img-hd*. At step 9, the first *uload* occurs as a consequence of the workload decrease. It triggers a reconfiguration in the *politics* instance as it goes from *conf3* to *conf2*, that is, it releases one instance of *AppServer* (*as3*). The same happens with *soccer* at step 10, which makes room on the resources and therefore allows *politics* to bring back the fidelity level of its *as2* to *img-ld*, and to the maximum level again at step 11. This is repeated at steps 13 and 14 for instances *politics* and *soccer* respectively, bringing their *consumptions* at the same levels as in the beginning.

6.2. Case Study 2: Mutual Exclusive Tasks

Figure 19 presents the example of a workflow, where boxes represent computing tasks to be executed, and where links are their execution dependencies from left to right. The application has parallel branches (indicated by a black dot), where all are executed starting at the same time. Alternative branches (indicated by a white dot) are also present: one and only of them will be executed, to be chosen by a controller according to the environing states and to a given global control objective coordinating parallel activities around constraints. These alternatives

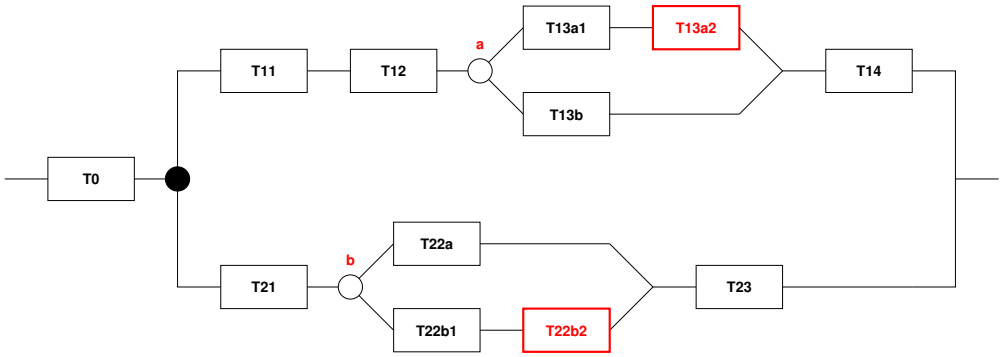


Figure 19: An Example Application Workflow, Showing Parallel and Alternative Branches.

correspond to the fact that, in a software component-based system, a given service or functionality can have different implementations, or can use different resources. Hence, each task corresponds to the execution of a components assembly, or possibly hierarchically to a sub-workflow. When the current situation is that task T_{ij}, the jth in branch i, is active, then upon reception of event e_{ij}, the reaction is to go to the next task in sequence in the flow (there can possibly be several tasks to start if there is a parallel branch). This is done by a reconfiguration from the current global configuration, towards a new configuration where the sub-assembly corresponding to the terminated task is replaced by the sub-assembly corresponding to the new ones.

6.2.1. Control Problem

In many cases, constraints forbidding the concurrent execution of some tasks e.g., because of their combined consumption of a resource exceeding available capacity, need to be defined. In this case the control must enforce their exclusivity. In this example, T_{13a2} and T_{22b2} must be kept exclusive. Therefore, a controller supervising the reconfigurations of the system should make the appropriate choices when choosing between alternative implementations of services. An alternative branch should be entered only if there is no risk that one of its tasks would have a conflict with another task in a parallel branch. This choice can be solved simply by having a semaphore-like mechanism, such that the first of the two tasks to be started would take the resource, and the other one, if started before the first one is terminated, must wait for the resource to be released again. However, this supposes that the starting of the tasks is controllable, which is not necessarily the case: it can be related to an event that is uncontrollable and must be answered immediately.

In our case, we consider that only the alternative branches choice points are controllable. Hence, when the progression in the workflow reaches a choice point in an alternative branching, for example **b** in Figure 19, then the controller must choose the branch by evaluating the global situation. If there is an exclusivity constraint then:

- if the other choice point **a** has already been passed then:
  - if the branch T_{13a} has been chosen then:
    - if the other task has not been passed yet there is a risk of a conflict, therefore the controller must choose at point **b** for the branch T_{22a}
    - else if is terminated then no conflict can occur anymore, and the choice is free at point **b** between T_{22a} and T_{22b},
  - else if the branch T_{13b} has been chosen (for some other reason) then the choice is free at point **b**,
- else the choice is open on both sides, and can be decided either randomly, or based on some other criterion like e.g., optimization related to costs of the alternative options. However, if the control for one alternative chooses to enter one of the branches featuring conflicting components, then this can constrain the choices allowed for the other alternative.

This decision clearly has to take into account not only the current state of active components/configurations and their resource consumption, but also the possible future evolutions which can be predicted according to the structure in branches of the application behavior.

6.2.2. Ctrl-F Model

The workflow example presented above is described in Ctrl-F as shown in Listing 12. For sake of simplicity and readability, we consider a single-component application containing two attributes (*r1* and *r2*), which corresponds to the consumption of two resources (lines 3 and 4). The component has several configurations (lines 8-21) in which, we specify the levels of consumption of each resource (cf. Table 3 for the complete configurations' specification). The events leading the system from one configuration to another are defined in line 6. The mutual exclusion is specified by a policy (line 26) stating that the sum of the consumptions of both resources must not exceed a certain capacity threshold. We also define another policy stating that the sum of the resource consumptions should be minimized (line 29).

It is noteworthy that exclusive components/configurations could be expressed in different ways. For instance, there could be two sub-components (a and b) , each with a boolean attribute (r) specifying whether the resource is used or not. Finally, a

policy at the composite level could make it explicit that *a* and *b*, regardless of their behaviour must not access the resource *r*, i.e., `! (a.r and b.r)`. For pedagogical purposes, we decided to express the mutual exclusion with numerical attributes and an explicit parallel behaviour, instead of two sub-components, which, when instantiated, behave implicitly in parallel (e.g., the *soccer* and *politics* Znn instance in the Znn Case Study).

Listing 12: Architectural Description of Components *Application* in Ctrl-F.

```
1  component Application {
2
3    attribute r1
4    attribute r2
5
6    port in e0,e11_12,e12_13...
7
8    configuration T0 {
9       set r1 to 0.5
10      set r2 to 0.2
11   }
12
13   configuration T11 {
14      set r1 to 0.4
15      set r2 to 0.0
16   }
17
18   configuration T12 { ... }
19   ...
20   configuration T14 { ... }
21   configuration T23 { ... }
22
23   behaviour { ... }
24
25   policy {
26        (r1 + rb) <= 1.0
27   }
28   policy {
29     minimize  (r1 + rb)
30   }
31
32 }
```

The Ctrl-F behavior for the workflow is defined inside the behaviour block (line 23), which is detailed in Listing 13. It consists of an outside *when-do* statement (lines 2-16), comprising a parallel statement, whose operator can be seen in line 10. Both branches of the parallel statement (lines 3-9 and 11-15) are composed of sequences of configurations with choice points. This sub-behaviours are modeled as a set of embedded *when-do*and *alternative* (lines 5 and 12) statements.

6.2.3. Execution

This section shows how the component-based software system modeled in Ctrl-F in the previous section reacts to different runtime scenarios, i.e., sequences of input events. More precisely, we want to exploit three scenarios where the policy could be violated, which implicitly means a violation on the mutual exclusion constraint between configurations *T13a2* and *T22b2*: (i) when the choice point *a* is evaluated before choice point *b*; (ii) when the choice point *b* is evaluated before choice point *a*; and when both choice points are evaluated at the same time. We consider the values for attributes *r1* and *r2* for each configuration as shown in Table 3.

Table 3: Workflow Configurations' Consumption Levels.

Configuration	**r1**	**r2**	**Configuration**	**r1**	**r2**
T0	0.5	0.2	T14	0.25	0.0
T11	0.4	0.0	T21	0.0	0.15
T12	0.2	0.0	T22a	0.0	0.3
T13a1	0.3	0.0	T22b1	0.0	0.2
T13a2	0.7	0.0	T22b2	0.0	0.6
T13b	0.35	0.0	T23	0.0	0.25

Tables 4, 5 and 6 show how the system evolves in terms of reconfiguration in response to a sequence of input events. The first row shows the incoming events at each logical time t_i. The second to the thirteenth rows indicate whether each configuration is active, whereas the row shows the sum of both attribute *r1* and *r2*.

For the first scenario (cf. Table 4) the sequence e_0, e_{11-12} and e_{12-13} leads the system the choice point *a* (at step t_4) before the choice point *b* , which make the controller pick the alternative that contains the configuration *T13a1* since it minimizes the sum of attributes *r1* and *r2*. Upon event e_{21-22}, the system is led to the choice point *b* (at step t_5) and the controller is forced to pick the alternative containing configuration *T22a* so as to respect the exclusion with *T13a2* and thus avoid an eventual policy violation ($r1 + r2 > 1.0$).

Table 4: Workflow Evolution Under Scenario 1.

	t_1	t_2	t_3	t_4	t_5	t_6
Events	e_0	e_{11-12}	e_{12-13}	$e_{13a1-13a2}$ e_{21-22}	e_{13-14} e_{22-23}	
T0	■					
T11		■				
T12			■			
T13a1				■		
T13a2					■	
T13b						
T14						■
T21		■	■			
T22a				■	■	
T22b1						
T22b2						
T23						■
r1+r2	0.7	0.6	0.55	0.6	1.0	0.5

Conversely, the sequence e_0 and e_{21-22}/e_{11-12} (cf. Table 5) takes the system first to the choice point *b* (at step t_3), which makes the controller choose the alternative containing *T22b1*, since it minimizes the resource consumption at next step. The next event (e_{12-13}), at step t_3, lead the other branch to choice point *a* (at step t_4). Since, at that point in time, configuration *T22b2* has not yet passed, the controller is forced to pick *T13b* in order to respect the mutual exclusion.

Finally, for the third scenario (cf. Table 6), upon events e_0,

Listing 13: Workflow Behaviour in Ctrl-F.

```
behaviour {
   T0 when e0 do
        T11 when e11_12 do
            T12 when e12_13 do
                ((T13a1 when e13a1_13ea2 do T13a2 end) | T13b) when e13_14 do
                    T14
                end
            end
        end
      ||
        T21 when e21_22 do
            (T22a | (T22b1 when e22b1_22b2 do T22b2 end)) when e22_23 do
                T23
            end
        end
   end
}
```

Table 5: Workflow Evolution Under Scenario 2.

	t_1	t_2	t_3	t_4	t_5	t_6
Events	e_0	e_{21-22} e_{11-12}	$e_{22b1-22b2}$ e_{12-13}	e_{22-23} e_{13-14}		
T0	■					
T11		■				
T12			■			
T13a1						
T13a2						
T13b				■		
T14					■	■
T21		■				
T22a						
T22b1			■			
T22b2				■		
T23					■	■
r1+r2	0.7	0.6	0.4	0.95	0.5	0.5

Table 6: Workflow Evolution Under Scenario 3.

	t_1	t_2	t_3	t_4	t_5	t_6
Events	e_0	e_{11-12}	e_{12-13} e_{21-22}	$e_{22b1-22b2}$	e_{13-14} e_{22-23}	
T0	■					
T11		■				
T12			■			
T13a1						
T13a2						
T13b				■	■	
T14						■
T21		■	■			
T22a						
T22b1				■		
T22b2					■	
T23						■
r1+r2	0.7	0.60	0.55	0.55	0.95	0.5

e_{11-12}, e_{12-13} / e_{21-22}, the system is lead to both choice points *a* and *b* at the same time, i.e., at step t_3. At that point, the decision taken by the controller is performed at the same time for both choice points so that the mutual exclusion is respected and the sum of resources is minimized. Thus, based on those criteria, for one branch, the configuration *T13b* is picked, whereas the alternative containing configuration *T22a1a* is chosen for the other branch.

Although, the scenarios above illustrate situations where either *T13a2* or *T22b2* are executed at the same time, there might be scenarios where the alternative statement of one branch is executed entirely before the one of the other branch. This means that the controller may allow the execution of both *T13a2* and *T22b2* as soon as the alternative behaviours are not executed at the same time.

6.3. Summary and Discussion

The control problem posed by software component-based systems as the ones illustrated in this section requires decision based upon observation of the current state and past events, but also prediction of possible futures from the current state. The manual design and writing of a control program doing this correctly would be very tedious and error-prone. Especially, obtaining a correct controller (which would require verification by model-checking anyway) is not the ultimate goal: it is important to have a maximally permissive controller, in order to keep the maximum flexibility in the execution and be minimally constraining by forbidding alternative branches only when necessary.

The case studies discussed in this section are very useful to understand how Ctrl-F and the *Manager* component that can be derived from are able to address those problems. First of all, from the language perspective, Ctrl-F is shown to be very useful to define self-adaptive component-based software systems in a

high-level and descriptive manner. From the control-theoretical point of view, the adaptation scenarios presented in this section shows in a pedagogical way, how controllers obtained by Ctrl-F compilation to Heptagon/BZR and DCS are capable to control reconfigurations by involving an exploration of future branches, from the current global advancement state, taking into account all possible event interleavings, and possible control choices in order to guarantee the stated policies.

7. Related Work

Our work can be compared to a body of work in the domains of Component-based Software Development, Model-Driven Development and Control.

In the domain of Component-based Software Development, runtime adaption is classically achieved by first relying on ADLs such as Acme [16] or Fractal [2] for an initial description of the software structure and architecture, then by specifying fine-grained reconfiguration actions with dedicated languages like Plastik [25] or FPath/FScript [5], or simply by defining Event-Condition-Actions (ECA) rules to lead the system to the desired state. A harmful consequence is that the space of reachable configuration states is only known as side effect of those reconfiguration actions, which makes it difficult to ensure correct adaptive behaviours. Moreover, a drawback of ECA rules is that, contrary to Ctrl-F, they cannot describe sequences of configurations. Even though, ECA rules can be expressed in Ctrl-F with a set of *when-do* (for the E part) and *case* (for the C and A parts) statements in parallel.

Rainbow [8] provides an autonomic framework for Acme components [16]. A DSL called Stitch is used to express autonomic behaviours (called strategies) in a tree-like manner. Branches in strategies are selected online by a utility-based algorithm according to runtime conditions. At the end (when it gets to a leaf in the tree), a strategy is evaluated as successful or failed and this information is used to improve the selection algorithm. A body of work [26][27][28][28][29][30][31] focus on how to plan a set of actions that safely lead component-based systems to a target configuration. These approaches are complementary to ours in the sense that our focus is on the choice of a new configuration and its control. Once a new configuration chosen, we rely on existing mechanisms to determine the plan of action actually leading the system from the current to the next configuration.

Kouchnarenko and Weber [9] propose the use of temporal logics to integrate temporal requirements to adaptation policies in the context of Fractal components [2]. The policies specify reflection or enforcement mechanisms, which refer respectively to corrective reconfigurations triggered by unwanted behaviours, and avoidance of reconfigurations leading to unwanted states. While in those approaches, enforcement (resp. decisions over strategies' branches) and reflection are performed at runtime, in our approach, the decisional part of the AM is obtained in an off-line manner, through the reactive language Heptagon/BZR and by performing DCS. This way, the exploration of behavioural programs is compiled away, producing as result a maximal permissive and correct-by-construction controller that enforces correct autonomic behaviours. That can be seen as a tremendous advantage, since the formal exploration can be very costly and exponential in the number of possible configurations to be performed online, which is even more complex when the control is required to be least restrictive. Conversely, due to model incompleteness and uncertainties inherent to unpredictable environments, assumptions taken at design time may no longer hold at runtime. One way to mitigate this limitation is to have a multi-tier control, as proposed by D'Ippolito et al. [32]. The idea is that one can define multiple models and controllers associated to different levels of assumptions (from the least to the most restrictive) and guaranteeable functionalities. The level of control is then determined according to the validity of assumptions at runtime.

In [33][34], feature models are used to express variability in software systems. At runtime, a resolution mechanism is used for determining which features should be present so as to constitute configuration. Those approaches rely on Model-Driven Engineering to ease the mapping between features and architectures as well as to automatically and dynamically generate the adaptation logics, i.e., the reconfiguration actions leading the target system from the current to the target configuration. In the same direction, Pascual et al. [35] propose an approach for optimal resolution of architectural variability specified in the Common Variability Language (CVL) [36]. A major drawback of those approaches is that in the adaptation logics specified with feature models or CVL, there is no way to define stateful adaptation behaviours, i.e., sequences of reconfigurations. In fact, the resolution is generally performed based on the current state and/or constraints on the feature model. On the contrary, in our approach, in the underlying reactive model based on FSA, decisions are taken also based on the history and possible futures of configurations which allows us to define more interesting and complex behaviours, while providing guarantees on them.

As in our approach, in [37], the authors also rely on Heptagon/BZR and DCS techniques to model autonomic behaviours in the context of Fractal components. An et al. [38] used Heptagon/BZR to conceive AMs in the context of partially reconfigurable FPGAs (Field Programmable Gate Arrays). Although those approaches provide us with interesting insights on how adaptive behaviours can be formalized, there is no general method allowing for the direct translation from a high-level description (e.g., ADL) to a synchronous reactive model. It means that for each new application, the formal model has to be recreated. Moreover, reconfigurations are controlled at the level of fine-grained reconfiguration actions (e.g., add/remove components and bindings), which can be considered time-consuming and difficult to scale, especially for large-scale architectures. In comparison, Ctrl-F proposes a set of high-level constructs to ease the description of adaptation behaviours and policies of component-based architectures. In addition, we propose an extensible AM that bridges Ctrl-F and a real component platform. Delaval et al. [39] propose the use of components to embody AMs conceived with Heptagon/BZR. The idea is to have modular controllers that can be coordinated so as to work together in a coherent manner. The approach is complementary to ours: on the one hand, it does not provide means to describe behavioural

programs for those managers, although the authors provide interesting intuitions on a methodology to do so. On the other hand, our approach does not provide means for the specification of the coordination among components' controllers. We do believe however that coordination is a major challenge that has to be tackled by any modular autonomic system. Hence, the integration of coordination aspects to Ctrl-F and its behavioural formalization must be considered in future work. Moreover, modularity seems to be an interesting perspective to mitigate the scalability issues due to state-space explorations.

8. Conclusion

8.1. Problem Statement Revisited

Reusability, modularity and reconfigurabily are enabling properties that make component-based architecture a major player in providing self-adaptive capabilities to software-intensive systems. High-level architecture description languages enable to define initial configurations, but programing adaptation behaviours is done with low-level fine-grained actions, which brings complexity, especially in large architectures. Another negative consequence is that the space of reachable configurations is only known as a side-effect of those fine-grained actions, which makes it hard to ensure correctness on the adaptive behaviours. In fact, such kind of adaptive behaviours requires decisions that are taken based not only on the past/current configurations and incoming events, but also on possible (reachable) futures configurations in a way to avoid branches, which, in the future, may lead the system to bad states.

8.2. Contributions

We tackled these problems by first proposing a high-level domain-specific language, named Ctrl-F, which allows for the description of adaptation behaviours and policies of component-based architectures and then by relying on discrete control to ensure correct adaptive behaviors. Concretely, we rely on Heptagon/BZR, a Finite State Automata-based reactive language, to formally define Ctrl-F's semantics. The compilation of Heptagon/BZR involves formal tools allowing for verification and Discrete Controller Synthesis (DCS), which makes it possible to ensure correct adaptive behaviours of component-based software systems defined in Ctrl-F. This article extends previous work [12, 7] by providing a detailed view on the Ctrl-F language itself, its compilation into Heptagon/BZR and its implementation and integration with FraSCAti, a Service Component Architecture middleware platform. Besides, we applied Ctrl-F to two cases studies and provided some discussion about its applicability.

8.3. Perspectives and Future Work

The intrinsic combinatorial complexity of Discrete Controller Synthesis raises some problems regarding the scalability of the approach. One perspective to be explored to cope with this issue is to decompose control problems specified in Ctrl-F in a systematical way, relying on the mechanisms of modular compilation and modular DCS [39]. That may require some effort on both the language redefinition and they way it is translated to Heptagon/BZR. Another important aspect that should be investigated in this domain is the possibility to make the adaption policies themselves subject to change, in order to react to changes in operation conditions imposing other properties than the ones specified at design time. Finally, in some contexts like in mobile computing, the presence of components in the system can change, by appearing or disappearing at any time. Therefore, an important perspective would be to work on a notion of adaptive control.

References

[1] I. Jacobson, M. Griss, P. Jonsson, Software reuse: architecture process and organization for business success, ACM Press books, ACM Press, 1997.

[2] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, J.-B. Stefani, An open component model and its support in java, in: Proceedings of the International Symposium on Component-based Software Engineering (CBSE'2003), Edinburgh, Scotland, 2004.

[3] I. Warren, J. Sun, S. Krishnamohan, T. Weerasinghe, An automated formal approach to managing dynamic reconfiguration, in: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, ASE '06, IEEE Computer Society, Washington, DC, USA, 2006, pp. 37–46.

[4] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, J.-B. Stefani, A component-based middleware platform for reconfigurable service-oriented architectures, Software: Practice and Experience 42 (5) (2012) 559–583. doi:10.1002/spe.1077.

[5] P.-C. David, T. Ledoux, M. Léger, T. Coupaye, FPath & FScript: Language support for navigation and reliable reconfiguration of Fractal architectures, Annals of Telecommunications: Special Issue on Software Components – The Fractal Initiative.

[6] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, P. Steenkiste, Rainbow: Architecture-based self adaptation with reusable infrastructure, IEEE Computer 37 (10).

[7] F. Alvares, E. Rutten, L. Seinturier, High-level Language Support for Reconfiguration Control in Component-based Architectures, in: Proc. 9th European Conf. on Software Architecture (ECSA), Dubrovnik, Croatia, 2015.

[8] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, P. Steenkiste, Rainbow: Architecture-based self-adaptation with reusable infrastructure, Computer 37 (10) (2004) 46–54.

[9] O. Kouchnarenko, J.-F. Weber, Adapting component-based systems at runtime via policies with temporal patterns, in: J. L. Fiadeiro, Z. Liu, J. Xue (Eds.), FACS 2013, 10th Int. Symposium on Formal Aspects of Component Software, Revised Selected Papers, Vol. 8348 of LNCS, Springer, Nanchang, China, 2014, pp. 234–253, revised Selected Papers.

[10] T. Abdelzaher, Y. Diao, J. Hellerstein, C. Lu, X. Zhu, Introduction to control theory and its application to computing systems, in: Z. Liu, C. Xia (Eds.), Performance Modeling and Engineering, Springer US, 2008, pp. 185–215.

[11] H. Marchand, P. Bournai, M. Le Borgne, P. Le Guernic, Synthesis of discrete-event controllers based on the signal environment, Discrete Event Dynamic System: Theory and Applications 10 (4) (2000) 325–346.

[12] F. Alvares, E. Rutten, L. Seinturier, Behavioural Model-based Control for Autonomic Software Components, in: Proc. 12th Int. Conf. Autonomic Computing (ICAC'15), Grenoble, France., 2015.

[13] G. Delaval, H. Marchand, E. Rutten, Contracts for modular discrete controller synthesis, in: ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2010), Stockholm, Sweden, 2010.

[14] S.-W. Cheng, D. Garlan, B. Schmerl, Evaluating the effectiveness of the rainbow self-adaptive system, in: Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS '09. ICSE Workshop on, 2009, pp. 132–141.

[15] N. Medvidovic, R. N. Taylor, A classification and comparison framework for software architecture description languages, IEEE Trans. Softw. Eng. 26 (1) (2000) 70–93.

[16] D. Garlan, R. T. Monroe, D. Wile, Acme: Architectural description of component-based systems, in: G. T. Leavens, M. Sitaraman (Eds.), Foundations of Component-Based Systems, Cambridge University Press, 2000, pp. 47–68.

[17] J. Kramer, J. Magee, Self-managed systems: An architectural challenge, in: 2007 Future of Software Engineering, FOSE '07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 259–268. doi:10.1109/FOSE.2007.19.
URL http://dx.doi.org/10.1109/FOSE.2007.19

[18] D. Harel, A. Pnueli, Logics and models of concurrent systems, Springer-Verlag New York, Inc., New York, NY, USA, 1985, Ch. On the Development of Reactive Systems, pp. 477–498.

[19] D. Harel, Statecharts: A visual formalism for complex systems, Sci. Comput. Program. 8 (3) (1987) 231–274. doi:10.1016/0167-6423(87)90035-9.

[20] N. Halbwachs, Synchronous Programming of Reactive Systems, Springer-Verlag, Berlin, Heidelberg, 2010.

[21] J.-L. Colaço, B. Pagano, M. Pouzet, A conservative extension of synchronous data-flow with state machines, in: Proceedings of the 5th ACM International Conference on Embedded Software, EMSOFT '05, ACM, New York, NY, USA, 2005, pp. 173–182.

[22] N. Berthier, H. Marchand, Discrete controller synthesis for infinite state systems with reax, in: IEEE International Workshop on Discrete Event Systems, Cachan, France, 2014, pp. 46–53.

[23] E. Dumitrescu, A. Girault, H. Marchand, É. Rutten, Multicriteria optimal reconfiguration of fault-tolerant real-time tasks, in: Workshop on Discrete Event Systems, WODES'10, IFAC, Berlin, Germany, 2010, pp. 366–373.
URL https://hal.inria.fr/inria-00510019

[24] Z. Xing, E. Stroulia, Umldiff: An algorithm for object-oriented design differencing, in: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE '05, ACM, New York, NY, USA, 2005, pp. 54–65. doi:10.1145/1101908.1101919.
URL http://doi.acm.org/10.1145/1101908.1101919

[25] T. Batista, A. Joolia, G. Coulson, Managing dynamic reconfiguration in component-based systems, in: Proceedings of the 2Nd European Conference on Software Architecture, EWSA'05, Springer-Verlag, Berlin, Heidelberg, 2005, pp. 1–17.

[26] N. Arshad, D. Heimbigner, A Comparison of Planning Based Models for Component Reconfiguration, Research Report CU-CS-995-05, U. Colorado (Sep. 2005).

[27] C. E. da Silva, R. de Lemos, Dynamic plans for integration testing of self-adaptive software systems, in: Proc. 6th Int. Symp. on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '11, 2011, pp. 148–157. doi:10.1145/1988008.1988029.

[28] M. Tichy, B. Klöpper, Planning self-adaption with graph transformations, in: Proc. 4th Int. Conf. on Applications of Graph Transformations with Industrial Relevance, AGTIVE'11, 2012, pp. 137–152. doi:10.1007/978-3-642-34176-2_13.

[29] F. Boyer, O. Gruber, D. Pous, Robust reconfigurations of component assemblies, in: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 13–22.
URL http://dl.acm.org/citation.cfm?id=2486788.2486791

[30] M. Luckey, B. Nagel, C. Gerth, G. Engels, Adapt cases: Extending use cases for adaptive systems, in: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '11, ACM, New York, NY, USA, 2011, pp. 30–39. doi:10.1145/1988008.1988014.
URL http://doi.acm.org/10.1145/1988008.1988014

[31] S. Becker, S. Dziwok, C. Gerking, C. Heinzemann, W. Schäfer, M. Meyer, U. Pohlmann, The mechatronicuml method: Model-driven software engineering of self-adaptive mechatronic systems, in: Companion Proceedings of the 36th International Conference on Software Engineering, ICSE Companion 2014, ACM, New York, NY, USA, 2014, pp. 614–615. doi:10.1145/2591062.2591142.
URL http://doi.acm.org/10.1145/2591062.2591142

[32] N. D'Ippolito, V. Braberman, J. Kramer, J. Magee, D. Sykes, S. Uchitel, Hope for the best, prepare for the worst: Multi-tier control for adaptive systems, in: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, ACM, New York, NY, USA, 2014, pp. 688–699.

[33] B. Morin, O. Barais, G. Nain, J.-M. Jezequel, Taming dynamically adaptive systems using models and aspects, in: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 122–132.

[34] C. Parra, X. Blanc, L. Duchien, Context awareness for dynamic service-oriented product lines, in: Proceedings of the 13th International Software Product Line Conference, SPLC '09, Carnegie Mellon University, Pittsburgh, PA, USA, 2009, pp. 131–140.

[35] G. G. Pascual, M. Pinto, L. Fuentes, Run-time support to manage architectural variability specified with cvl, in: Proceedings of the 7th European Conference on Software Architecture, ECSA'13, Springer-Verlag, Berlin, Heidelberg, 2013, pp. 282–298.

[36] O. Haugen, A. Wasowski, K. Czarnecki, Cvl: Common variability language, in: Proceedings of the 17th International Software Product Line Conference, SPLC '13, ACM, New York, NY, USA, 2013, pp. 277–277.

[37] T. Bouhadiba, Q. Sabah, G. Delaval, E. Rutten, Synchronous control of reconfiguration in fractal component-based systems: A case study, in: Proceedings of the Ninth ACM International Conference on Embedded Software, EMSOFT '11, ACM, New York, NY, USA, 2011, pp. 309–318.

[38] X. An, E. Rutten, J.-P. Diguet, N. Le Griguer, A. Gamatié, Autonomic Management of Dynamically Partially Reconfigurable FPGA Architectures Using Discrete Control, in: In Proc. of the 10th International Conference on Autonomic Computing (ICAC'13), SAN JOSE, CA, United States, 2013.

[39] G. Delaval, S. M.-K. Gueye, E. Rutten, N. De Palma, Modular coordination of multiple autonomic managers, in: Proceedings of the 17th International ACM Sigsoft Symposium on Component-based Software Engineering, CBSE '14, ACM, New York, NY, USA, 2014, pp. 3–12.