HAL
open science

High-level Language Support for the Control of Reconfiguration in Component-based Architectures

Frederico Alvares de Oliveira Jr., Eric Rutten, Lionel Seinturier

To cite this version:

Frederico Alvares de Oliveira Jr., Eric Rutten, Lionel Seinturier. High-level Language Support for the Control of Reconfiguration in Component-based Architectures. [Research Report] RR-8669, INRIA Grenoble - Rhône-Alpes; INRIA Lille - Nord Europe; Laboratoire d'Informatique Fondamentale de Lille; INRIA. 2015. hal-01103548

HAL Id: hal-01103548

https://inria.hal.science/hal-01103548v1

Submitted on 14 Jan 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

High-level Language Support for the Control of Reconfiguration in Component-based Architectures

Frederico Alvares de Oliveira Jr., Eric Rutten, Lionel Seinturier

RESEARCH REPORT
N° 8669
January 2015
Project-Teams Ctrl-a and Spirals

ISSN 0249-6399 ISRN INRIA/RR--8669--FR+ENG

High-level Language Support for the Control of Reconfiguration in Component-based Architectures

Frederico Alvares de Oliveira Jr.*, Eric Rutten, Lionel Seinturier†

Project-Teams Ctrl-a and Spirals

Research Report n° 8669 — January 2015 — 34 pages

Abstract:
Architecting in the context of variability has become a real need in nowadays software development. Modern software systems and their architecture must adapt dynamically to events coming from the environment (e.g., workload requested by users, changes in functionality) and the execution platform (e.g., resources availability). Component-based architectures have shown to be very suited for self-adaptation purposes, not only because of their intrinsic characteristics like reusability and modularity, but also as virtue of their dynamical reconfiguration capabilities. The issue, nevertheless, remains that adaptation behaviors are generally conceived by means of fine-grained reconfiguration actions from the very initial configurations. This way, besides the complexity in managing large-sized architectures, the space of reachable configurations is not know in advance, which prevents ensuring well-mastered adaptive behaviours. This paper presents Ctrl-F, a domain-specific language whose objective is to provide high-level support for describing adaptation behaviours and policies in component-based architectures. The proposed language lies on synchronous reactive programming, which means that it benefits of an entire environment and formal tooling allowing for the verification and control of reconfigurations. We show the applicability of Ctrl-F by first integrating it to FraSCAti, a Service Component Architecture middleware platform, and then by applying it to Znn.com, a well known self-adaptive case study.

Key-words: Component-based Architectures, Self-adaptation, Control Theory

* INRIA Grenoble - Rhône-Alpes
† INRIA Lille Nord Europe

RESEARCH CENTRE GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Support langage de haut niveau pour le contrôle des reconfigurations dans les architectures à composants

Résumé : La conception des architectures logicielles dans le contexte de la variabilité est devenue un vrai besoin du développement logiciel de nos jours. Les systèmes logiciels modernes et leur architectures doivent s'adapter dynamiquement aux événements provenant de l'environnement (par exemple, la charge de travail demandée par les utilisateurs, les modifications apportées aux fonctionnalités) et la plate-forme d'exécution (par exemple, la disponibilité de ressources). Les architectures à base de composants ont montré être très appropriées à l'auto-adaptation, en raison non seulement de leurs caractéristiques intrinsèques, comme la réutilisation et la modularité, mais aussi grÃ¢ce à leurs capacités de reconfiguration dynamiques. Néanmoins, la question demeure que les comportements d'adaptation sont généralement conçus au moyen d'actions de reconfiguration à grain fin à partir des configurations initiales. De cette façon, en plus de la complexité de gestion, notamment dans le cadre des grandes architectures, l'espace de configurations accessibles n'est pas explicitement connu à l'avance, ce qui empêche d'assurer des comportements adaptatifs bien maîtrisés. Cet article présente Ctrl-F, un langage dédié dont l'objectif est de fournir un support de haut niveau pour décrire des comportements et politiques d'adaptation pour les architectures à base de composants. Ce langage s'appuie sur la programmation réactive synchrone, ce qui signifie qu'il bénéficie de tout un environnement et outillage formels permettant la vérification et le contrôle de reconfigurations par la voie de la synthèse de contrôleurs discrets. Nous montrons l'applicabilité de Ctrl-F d'abord par l'intégration à FraSCAti, une plate-forme middleware pour la Service Component Architecture, puis en l'appliquant à Znn.com, une application auto-adaptative utilisée comme cas d'étude.

Mots-clés : Architecture à composants, auto-adaptation, théorie du contrôle

1 Introduction

From tiny applications embedded in house appliances or automobiles to huge Cloud services, nowadays software-intensive systems have to fulfill a number of requirements in terms safety and/or Quality of Service (QoS) while facing highly dynamic environments (e.g., varying workloads and changing user requirements) and platforms (e.g., resource availability). Thus, it becomes a real necessity to engineer such software systems with principles of self-adaptiveness in mind, i.e., to equip software systems with capabilities to dynamically change themselves in order to cope with such a level of variability.

In this direction, software architectures and more specifically software components have played a very important role. Besides the usual benefits of modularity and reuse, adaptability and reconfigurability are key properties which are sought with this approach: one wants to be able to adapt the component assemblies in order to cope with new requirements and new execution conditions occurring at run-time. However, in the current state of the art, there is no explicit, first-class language-level support for the specification of dynamic reconfigurations. In general, initial assemblies (or configurations) are defined with the help of Architectural Description Languages (ADLs) and reconfigurations are achieved by programming fine-grained actions in either general-purpose languages within component-based frameworks [23], or with the support of reconfiguration domain-specific languages (DSLs)[8]. Hence, the space of configurations that can be reached as well as the possible switchings between them are only considered through side effects. As a further matter, dealing with reconfigurations at such a low level may easily become a very exhaustive, tedious and error-prone task, especially for large-scale architectures.

We claim that there is a need for language support not only for the declaration of configurations in the form of assemblies, but also for the explicit specification of policies driving reconfigurations with respect to when or under which conditions reconfigurations shall be triggered or even which configurations in the reconfiguration history should be avoided. Furthermore, designing well-mastered behaviors of reconfigurable systems, with assurances on the way they navigate in the configuration space while respecting policies defined separately, requires tool-supported design methods.

The control of these reconfigurations has to be managed in reaction to firing events and conditions from observations on the component-based architecture and its environment, to take decisions depending on the current state of the architecture as well as its reconfiguration history and its constraints. The result is a set of actions to be performed for going to the next configuration. This forms a feedback control loop, as in autonomic computing [17], whose design can benefit from a whole methodology, programming environments and tools that have been developed for the design of reactive systems, and hence be empowered with specification, verification, validation, implementation and automatic generation of executable code capabilities [2]. More precisely, it is possible to have controllers conceived with synchronous languages [10], which in turn, combine, among other things, techniques of Discrete Controller Synthesis (DCS) to generate correct controllers, from a Finite State Automata (FSA) based model of the controlled system, and a given strategy expressed as a safety property. The advantage with respect to manual programming of a controller are: automation of the design, correctness of the synthesized controller, and optimality in the sense of maximal permissiveness.

This paper presents Ctrl-F, a language that extends classic ADLs with high-level constructs intended to express the dynamicity of component-based architectures. In addition to the usual description of assemblies (configurations), existing in other ADLs, Ctrl-F also comprises a set of constructs that are dedicated for the description of: (i) behavioural aspects, that is, the order and/or conditions under which configurations should take place; and (ii) policies that have to be enforced all along the execution. We provide full translation from Ctrl-F to the

synchronous reactive language Heptagon/BZR [10], which allows us to benefit from tools for: (i) formal exploitation of programs, for verification and controller synthesis purposes; and (ii) the compilation towards executable code in general purpose languages (e.g., Java or C). We validate the applicability of our language through *Znn.com* [6], a self-adaptive news website that is commonly used as case study.

The remainder of this paper is organized as follows: Section 2 introduces the main concepts and tools necessary for the good comprehension of our approach. Sections 3 presents the case studied that is used all along the paper. Sections 4 and 5 present the Ctrl-F language itself and how it is translated into Heptagon/BZR. Section 6 provides some details on the integration of Ctrl-F with a real component platform. The related work is discussed in Section 7 and Section 8 concludes this paper.

2 Background

This section aims at introducing the main concepts and tools used to develop this research work. First, we provide an overview on Component-based Architecture along with its suitability for self-adaptiveness. Then we survey some concepts regarding Reactive Programming, which is the formal paradigm we rely on in order to ensure correct reconfiguration behaviours.

2.1 Component-based Architecture

2.1.1 Basic Concepts

A software architecture defines the high-level structure of a software system, by describing how it is organized as a means of a composition of components [16]. Architecture description languages (ADL) [20] are usually used to describe the architecture of a system. Although the diversity of ADLs, the architectural elements proposed in almost all of them follow the same conceptual basis [14].

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 defines how the component should interact with the environment. A component can be defined as simple or *composite* (composed of other components). A *connector* corresponds to interactions among components. Roughly, 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 often appear in ADLs [14]. For sake of brevity we omit further description on this regards.

Nowadays software applications require the possibility to continuous change so as to adapt to execution conditions or cope with new user requirements. Thus, it becomes imperative that an architecture/configurations once defined may be evolved while leading to new configurations that are more appropriated to the current context.

2.1.2 Dynamic Reconfiguration

Dynamic reconfiguration is denoted 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 interferences on the system execution is reduced and the 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 reflexion, modern component models like Fractal [5] and OpenRec [24], among others, bring reflection capabilities to software architectures, by providing 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.).

At first glance, as can be seen in Figure 1, the Fractal Component Model seems pretty much like any other component model, allowing for hierarchical composition of components, specification of required (client) and provided (server) interfaces and bindings connecting components' interfaces. However, Fractal was rather conceived with *separation of concerns* design principle in mind, which separates 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* and the *binding* controllers. The former controls the component's behavioural phases (e.g., starting, started, stopping, stopped, etc.), while the latter provides means for dynamically establishing or breaking bindings between component's interface and its environment.

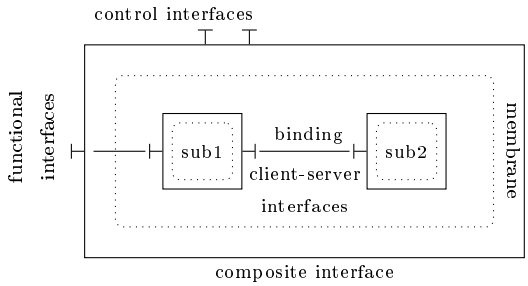


Figure 1: Fractal Architectural Concepts.

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, in which software systems should adapt to context changes while mitigating the interferences on their execution.

2.1.3 Service Component Architectures

While Component Models usually focus on the modularity, reuse and adaptability, they do not fulfill a number of requirements regarding interoperability, platform and protocol diversity in distributed environments over the Internet. Service-Oriented Architecture (SOA), on the other hand, is an architectural style in which applications are conceived as composition of loosely coupled, coarse grained, technology agnostic, remotely accessed, and interoperable (i.e. accessed via standardized protocols) services. In the other sense, SOA requires appropriate infrastructure to conceive, deliver and manage distributed applications based on the SOA principles [23]. Therefore, Component-based Architecture and SOA can be seen as complementary approaches.

In this context, Service Component Architecture (SCA) [1] is a component model for building applications based on the SOA principles. SCA provides means for constructing, assembling and deploying software components regardless the programming language or protocol used to implement and make them communicate. For that purpose, the SCA specification is decomposed into a number of specifications, including the component implementation specification, whose objective is specify how a component can actually be implemented in a specific programming language (e.g., Java, C, C++, etc.); and the binding specification, in which one can define how services are exposed and/or how dependencies on other components are satisfied by describing which access methods and/or transport are allowed.

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. Although this work's contribution is technology-agnostic, for proof-of-concept and implementation purposes, we rely on SCA and more precisely on the SCA middleware platform FraSCAti [23]. The reason for that resides in that fact that FraSCAti is developed on top of Fractal Component Model and thus inherits Fractal's reflection capabilities and therefore allows for dynamic reconfiguration of SCA application.

2.2 Reactive Programming

In this work we rely on the synchronous reactive language Heptagon/BZR [10] to formally model Ctrl-F and thus benefit of an entire ecosystem including specification language environments, verification, control and executable code generation tools.

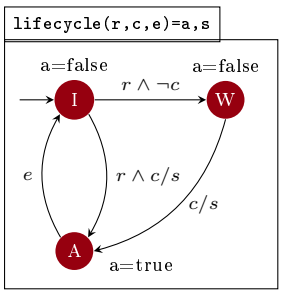
2.2.1 Heptagon

Synchronous Reactive Languages based on Finite State Automata (FSA) constitute the basic formalism for Discrete Control Theory. The general execution scheme of a reactive program is that at each reaction, a step is performed taking input flows, computing transitions, updating states, triggering actions, emitting output flows.

Heptagon is a language allowing for the definition of reactive systems by means of generalized Moore machines, that is, with mixed synchronous data-flow equations and automata [7]. An Heptagon program is modularly structured with as 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 determines the values for each output, in terms of expressions on inputs' instantaneous values or other flows values.

Figure 2.2.1 shows an Heptagon program in both graphical and textual representations. The program describes the control of a component's life-cycle 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 a end signal. It produces as output three boolean values, one that indicates whether the component is active (`a`) the another indicating a start actions (`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 (i.e., when `e` is true).

[1] `http://www.oasis-opencsa.org/`




```
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
```

Figure 2: Graphical and Textual Representation of Component Life-cycle.

One important characteristic of Heptagon/BZR is the support for hierarchical and parallel automata composition. Listing 1 illustrates an example of hierarchical composition, in which a single-stated super-automaton embodies the *lifecycle* automaton. It has a self-transition that results in the resetting of the containing automata (i.e., *lifecycle*) at at every occurrence of signal *b*. A stream of input/output values for this automaton can be seen in Table 2.2.1. 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. Next section provides an example on the parallel composition.

Listing 1: Example of Hierarchical Composition.

```
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
```

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 a distinguished reactive language, since its compilation may involve

formal tools such as Sigali [19] and Reax [3] 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 resulting in e_A, the output will always result in 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, although no controller is generated.

Listing 2 shows an example of contract. The example presents a node enclosing a parallel composition of two instances of node *lifecycle* (c.f. Figure 2.2.1). As can be seen, the contract 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 property to be enforced by the node, which in this case is $e_G = \neg(a1 \wedge a2)$, meaning that both components are mutual 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 (lines 7 and 8). In practice the controllable variables (`c1` and `c2`) will be given values in a way variables `a1` and `a2` are never both true at the same instant.

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
```

2.2.3 Compilation and code generation

The Heptagon/BZR compilation chain is depicted in Figure 2.2.3. From a source code (1), the Heptagon/BZR compiler produces as output an sequential code in a general-purpose programming language (e.g., Java or C) corresponding to the system model (2). At the same time, if the code provided as input contains any *contract*, the compiler will also generate a intermediary code that will be given as input to the model checker (2, in this case either Sigali or Reax), which will, in turn, perform the DCS and produce as output an Heptagon/BZR code corresponding to the generated controller (3). The latter is then compiled again so as to have an executable code also for the generated controller (4).

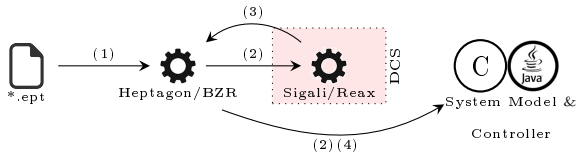


Figure 3: Heptagon/BZR Compilation Chain.

The executable code comprises two function/methods: `reset` and `step`. 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 functions/methods are typically used by first executing `reset` and then by enclosing `step` in an infinite loop, in which each interaction correspond to a reaction, i.e., a step to be performed.

3 Example Application

This section provides an overview of a motivating scenario that is used throughout the following sections to explain several aspects and concepts of our proposed Ctrl-F language. The scenario is an extension of the well-known Znn.com [6] use case.

3.1 Overview of Znn.com

Znn.com [6] is an experimental platform for self-adaptive applications, which mimics a news website. As in any web application, Znn follows a typical client-server n-tiers architecture, as illustrated in Figure 4. Znn.com 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).

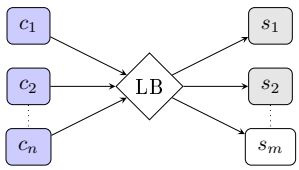


Figure 4: Znn.com Architecture.

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 could be forced to degrade the content fidelity so as to require fewer resource 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;
- The content fidelity of each server.

3.2 Znn.com Instantiation

We extend Znn.com by enabling its replication in presence of difference content providers. Figure 5 illustrates the case with two different content providers: one specialized in soccer and another one specialized in politics. These two instances of Znn.com will be running/sharing on the same physical infrastructure (machines $m_1, \ldots, m_k$). Depending on the contract signed between the service provider and his/her clients that establishes the terms of use of the service, Znn.com Service Provider can give more or less priority to a certain client. For instance, during the World Cup the content provider specialized in soccer will always have priority over the other one. Conversely, during the elections, the politics-specialized content provider is the one that has the priority.

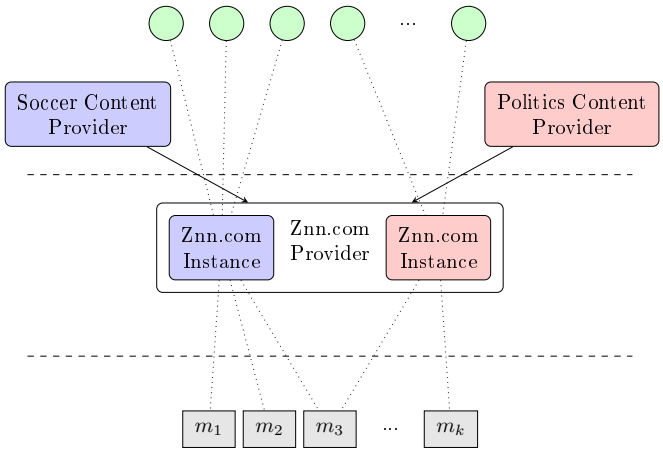


Figure 5: Znn.com Instances.

4 Ctrl-F Language

Ctrl-F is our proposal for a domain-specific language that extends the common concepts of classic ADLs by providing high-level constructs for describing reconfigurations' behaviour and policies to be enforced all along the execution of the target system.

4.1 Overview and Common Concepts

Figure 6 presents a meta-model of the language's abstract syntax. It 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 shares the same concepts of many existing ADLs (e.g., Fractal [5], Acme [14]). 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. So 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 given *Event Type* a *component* is able to emit (*port out*) or listens to (*port in*).

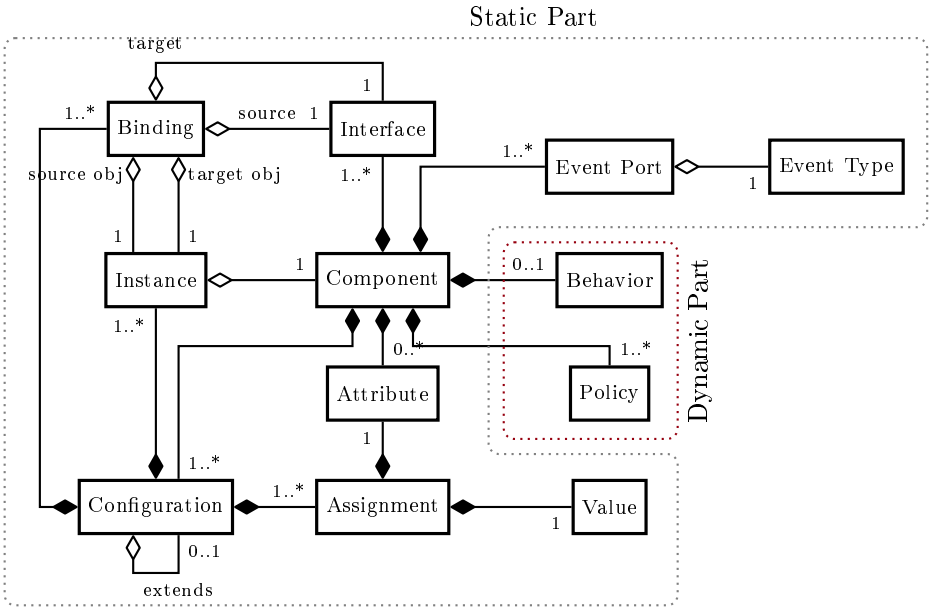


Figure 6: Language Abstract Syntax.

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 *attributes* assignment to *values*. That is to say that a *configuration* is a snapshot of the *attributes* valuation, the current (sub) components *instances* within the concerned (super) component and the *bindings* connecting *interfaces* of these *instances*.

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

The Znn.com example application of Section 3 can be modeled as a hierarchical composition of four components: (i) *Main*; (ii) *Znn*; (iii) *LoadBalancer*; and (iv) *AppServer*. These components are instantiated according to execution conditions, the system's current state (architectural composition), adaptation behaviours and policies defined within each component. The following paragraphs details the definition of such components with the static part of Ctrl-F.

The Ctrl-F description for the *Main* component is shown in Listing 3. This is the top-most component and it encompasses two instances of component *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. Here we also define a some policies to be enforced (line 13), but these aspects are discussed in more details in the coming sections.

Listing 3: Main Component in Ctrl-F.

```
1 component Main {
2
```

```
3   configuration main {
4       soccer:Znn
5       politics:Znn
6   }
7
8   policy {...}
9 }
```

Listing 4 shows the definition of component *Znn*. It consists of one provided interfaces (line 2) through which news can be requested. The component listens to events of types *oload* (overload) and *uload* (underload) (lines 5 and 6), which are emitted by other components. In addition, the component also defines two attributes: *consumption* (line 9), which is used to express the level of consumption (in terms of percentage of CPU) incurred by the component execution; and *fidelity* (line 8), which expresses the content fidelity level of the component in question.

Three configurations are defined for *Znn* component: *conf1*, *conf2* and *conf3*. *conf1* (lines 11-18) consists of one instance of each *LoadBalancer* and *AppServer* (lines 12-13); one binding to connect them (line 15), another binding to expose the server interface of the *LoadBalancer* component as a server interface of the *Znn* component (line 14), and the attribute assignments (lines 16 and 17). 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 19-24) *extends conf1* by adding one more instance of *LoadBalancer*, 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 22), whereas for the consumption the same logics is applied so the consumption of the just-added instance is incorporated to the sum expression (line 23). Due to lack of space we omit the definition of configuration *conf3*. Nevertheless, it follows the same idea, that is, it extends *conf2* by adding a new instance of *LoadBalancer*, binding it and redefining the attribute values.

Listing 4: Znn Component in Ctrl-F.

```
1 component Znn {
2
3   server interface si
4
5   port in oload
6   port in uload
7
8   attribute fidelity
9   attribute consumption
10
11  configuration conf1 {
12     lb:LoadBalancer
13     as1:AppServer
14     bind lb.ci1 to as1.si
15     bind lb.si to si
16     set fidelity to as1.fidelity
17     set consumption to sum(as1.consumption,lb.consumption)
18  }
19  configuration conf2 extends conf1 {
20     as2:AppServer
21     bind lb.ci2 to as2.si
22     set fidelity to avg(as1.fidelity,as2.fidelity)
23     set consumption to sum(as1.consumption,as2.consumption,lb.
          consumption)
```

```
24   }
25
26   configuration conf3 extends conf2 {...}
27
28   behaviour {...}
29   policy {...}
30 }
```

Listing 5 shows the description of component *LoadBalancer*. It consists of four interfaces: one provided (line 2), through which the news are provided; and the others required (line 3), 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 5 and 6). Like for component *Znn*, attribute *consumption* (line 8) specifies the level of consumption of the component. As there is no explicit definition of configurations, *LoadBalancer* is implicitly treated as a single-configuration component.

Listing 5: Load Balancer Component in Ctrl-F.

```
1 component LoadBalancer {
2   server interface si
3   client interface ci1,ci2,c3
4
5   port out oload
6   port out uload
7
8   attribute consumption=0.2
9 }
```

Lastly, component *AppServer* (Listing 6) has only one interface (line 2) and listens to events of type *oload* and *uload* (lines 4-5). The component has also two attributes: consumption and fidelity (lines 7 and 8), just like component *Znn*. Three configurations corresponding to each level of fidelity (lines 10-13, 14-17 and 18) are defined within the component, and its attributes are valuated according to the configuration in question, that is, the higher the fidelity the higher the consumption.

Listing 6: AppServer Component in Ctrl-F.

```
1 component AppServer {
2   server interface si
3
4   port in oload
5   port in uload
6
7   attribute fidelity
8   attribute consumption
9
10  configuration text {
11    set fidelity to 0.25
12    set consumption to 0.2
13  }
14  configuration img-ld {
15    set fidelity to 0.5
16    set consumption to 0.6
17  }
```

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 **end**	Execute B_i if c_i holds
$B_1 \mid B_2$	Execute either B_1 or B_2
$B_1 \mid\mid B_2$	Execute B_1 and B_2 in parallel
do B **every** e	Execute B and re-executes at every occurrence of e

```
18   configuration img-hd {...}
19
20   behaviour {...}
21   policy {...}
22 }
```

4.2 Behaviours

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

Behaviours in Ctrl-F are defined with the aid of a high-level imperative language, which consists of a set of behavioural statements (*sub-behaviours*) that can be composed together so as provide more complex behaviours in terms of sequences of reconfiguration. In this context, a *configuration* is considered as an atomic behaviour, i.e., a behaviour that cannot be decomposed into other *sub-behaviours*.

We assume that configurations do not have the capability to terminate or start themselves, meaning that they are explicitly requested or ended by behaviour *statements*. Hence, a reconfiguration occurs when the current configuration is terminated and the next one is started.

The next sections discuss the behaviour statements while illustrating their usage in the *Znn.com* example application.

4.2.1 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 event type e_i arrives the configuration(s) that composes 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. The *parallel*

statement states that two behaviours can be executed at the same time. That is to say that at a certain point there must be two independent branches of behaviour executing in parallel. The *alternative* statement allows to describe choice points between configurations or modes in atomic components, or between more elaborated sequential behavior 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 at every occurrence of e. In other words, the configuration(s) currently activated in B is (are) terminated, and the very first one(s) in B is (are) started.

4.2.2 Example in Znn.com

This section illustrates the use of the statements introduced in the last section to express a behaviour for components *AppServer* and *Znn* (c.f. Section 4.1).

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 7, the behaviour can be decomposed in a *do-every* (lines 4-6) statement, which is, in turn, is composed of an *alternative* one (line 5). 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 5.

Listing 7: Behaviour of AppServer Component.

```
1 component AppServer {
2   ...
3   behaviour {
4     do
5       text | img-ld | img-hd
6     every (oload or uload)
7   }
8 }
```

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 an event of type *uload*, another instance must be removed, which leads the application to configuration *conf2*.

As shown in Listing 8, that behaviour can be achieved with a main *do-every* statement (lines 4-12), which executes a *when-do* statement (lines 5-11) at every occurrence of an event of type *e1*. In practice, the firing of this event allows to go back to the beginning of the *when-do* statement, that is, the configuration *conf1* regardless of the current configuration being executed. According to the *when-do* statement, *conf1* is executed until the occurrence of an event of type *oload* (line 5), then another *do-every* statement is executed (lines 6-10), which in turn, just like the other one, executes another *when-do* statement (lines 7-9) and repeats it at every occurrence of an event of type *e2*. Again, that structure allows the application go back to the beginning of the *when-do* statement, that is, the configuration *conf2*. Configuration *conf2* is executed until an event of type either *oload* or *uload* occurs. For the former case (line 7), another *when-do* statement takes place, whereas for the latter (line 8) a configuration named *emitter1* is executed. The *when-do*

statement (line 7) consists in executing configuration *conf3* until an event of type *uload* is fired, then a configuration named *emitter2* takes place.

It is noteworthy that configurations *emitter1* and *emitter2* are special configurations that contain, each one, an instance of the pre-defined component *Emitter* (omitted here due to space constraints). The purpose of this component is to emit events such as the ones of type *e1* and *e2*. This allows the application to trigger the inner-statements within the *do-every* statements (lines 1-12 and 6-10) so as to be able to to back to configurations *conf1* and *conf2*, from configurations *conf2* and *conf3*, respectively.

Listing 8: Behaviour of Znn Component.

```
1  component Znn {
2    ...
3    behaviour {
4      do
5        conf1 when oload do
6          do
7           conf2 when oload do (conf3 when uload do emitter2 end),
8                       uload do emitter1
9           end
10         every e2
11       end
12     every e1
13   }
14 }
```

4.3 Policies

Policies are expressed with high-level constructs that enable the definition of constraints on configurations in a declarative way. They can be grouped into temporal constraints and constraints on attributes. In general, they define a subset of all possible global configurations, where the system should remain: this will be achieved by using the choice points in order to control the reconfigurations. An intuitive example is that two components in parallel branches might have each several possible modes, and some of them to be kept exclusive. This exclusion is a policy which can be enforced by choosing the appropriate modes when starting the components.

4.3.1 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 9 illustrates examples of 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 0.7, whereas that for 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 6, which could be interpreted as a constraint on the physical resource capacity, that is, the number of available machines ($k = 6$).

Listing 9: Example of Constraint and Optimization on Attributes.

```
component Main {
  ...
  policy {
     soccer.fidelity >= 0.7
```

```
  }
  policy {
     maximize politics.fidelity
  }
  policy {
     (soccer.consumption + politics.consumption) < 6
  }
}
```

4.3.2 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.

In order to ease the specification of such kind of 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 $conf_2$ has been terminated.

Listing 10 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 could take place along with *img-ld*, but the *alternative* statement in the behaviour described in Listing 7 leads us to conclude that only *img-ld* must take place right after either *text* or *img-hd* has been terminated.

Listing 10: Example of Temporal Constraint.

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

5 Modeling Ctrl-F in Heptagon/BZR

The constructs presented in Section 4 allow for the high-level description of reconfigurations and policies on configuration orders and/or on architectural elements. For simple examples like

Znn.com, the control of reconfigurations in such a way that policies are enforced can be easily achieved in any general purpose language. However, as applications get larger and more complex, the implementation of such a control becomes harder and error-prone. For this reason, we rely on a synchronous reactive language, namely Heptagon/BZR, to model component-based applications described in Ctrl-F in a way to benefit from verification, discrete control and executable code generation tools. This section presents a set of translation schemes that enable us to model Ctrl-F-described applications into Heptagon/BZR.

5.1 General Model: The Component

A component can be modeled as an Heptagon/BZR node, as shown in Figure 7. It receives 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 $\{w_1, \ldots, w_l\}$, which correspond to attribute valuation for each attribute defined in the component.

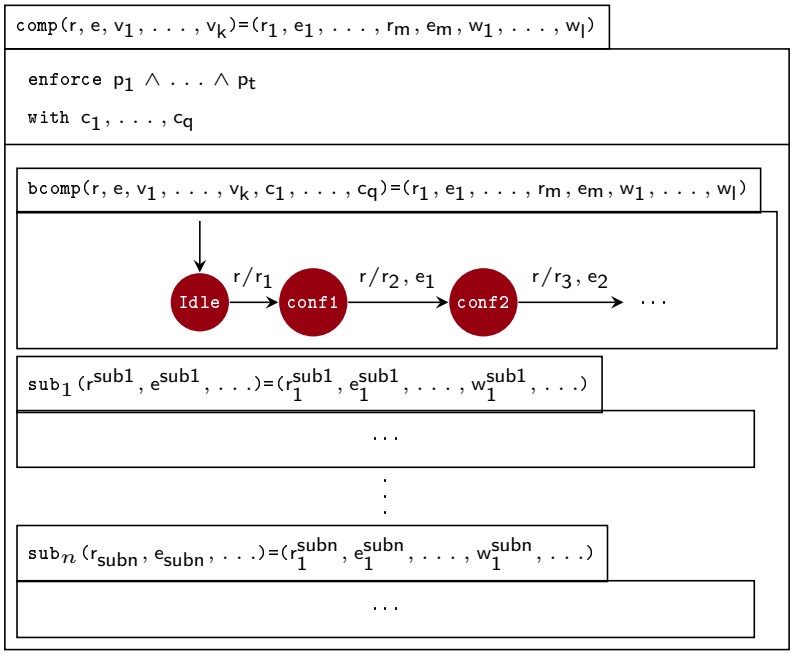


Figure 7: Translation Scheme Overview.

The main node (comp in Figure 7) 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 discrete controller synthesis. The details on how policies are translated are given in Section 5.3.

Component behaviours are modeled as a sub-node (bcomp in Figure 7), which consists of an automaton describing the orders and conditions 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 5.2.

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 ($\mathsf{r}^{\mathsf{subi}}$) and end ($\mathsf{e}^{\mathsf{subi}}$) signals for a sub-component $sub_i \in \{sub_1, \ldots, sub_n\}$ are defined as equations of request and end signals for the configurations the concerned component belongs to (cf. Equation 1).

$$r^{sub_i} = (r_1 \vee \ldots \vee r_m) \wedge \neg(e_1 \vee \ldots \vee e_m) \tag{1}$$

$$e^{sub_i} = (e_1 \vee \ldots \vee e_m) \wedge \neg(r_1 \vee \ldots \vee r_m) \tag{2}$$

Where $\{r_1, \ldots, r_m\}$ and $\{e_1, \ldots, e_m\}$ are respectively the sets of request and end signals for the configurations $conf_1, \ldots,$ $conf_m$ component sub_i belongs to. 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 a component that is already active. The same applies for its termination.

Listing 11 shows an excerpt of Heptagon/BZR model for components *Znn* (lines 6-23) and *AppServer* (lines 1-4). For node *appserver*, besides the request and end signals, it gets as inputs the events of type `oload` and `uload` (line 1). As output (lines 2 and 3), it produces request and end signals for configurations text (`r_text` and `e_text`), image-ld (`r_ld` and `e_ld`) and image-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 10-22), `znn` comprises one instance of the node that models the behaviour (`bznn`, line 17) and three instances of node `appserver` (lines 18-20). The request and end signals for these instances can be derived from the request and end signals for configurations (lines 10-15). At last, attributes are values based on the values of attributes of the instances of node `appserver` (line 21).

Listing 11: 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;
3                  fidelity,consumption:int)
4  let ... tel
5
6  node znn(r,e,oload,uload:bool) returns
7          (r_conf1,e_conf1,...,r_conf3,e_conf3:bool;
8           fidelity,consumption:int)
9  let
10 r_as1 = r_conf1 or r_conf2 or r_conf3 and not(e_conf1 or e_conf2 or
     e_conf3);
11 r_as2 = r_conf2 or r_conf3 and                          not(e_conf2 or
     e_conf3);
12 r_as3 = r_conf3 and not(e_conf3);
13 e_as1 = e_conf1 or e_conf2 or e_conf3 and not(r_conf1 or r_conf2 or
     r_conf3);
14 e_as2 = e_conf2 or e_conf3 and                          not(r_conf2 or
     r_conf3);
```

```
e_as3 = e_conf3 and not(r_conf3);

(r_conf1,e_conf1,...)=bznn(r,e,oload,uload);
(r_text_as1,...,fid_as1,conso_as1)=appserver(r_as1,e_as2,oload,uload);
...
(r_text_as3,...,fid_as3,conso_as3)=appserver(r_as3,e_as3,oload,uload);
consumption=conso_as1+conso_as2+conso_as3;
...
tel
```

5.2 Modeling Behaviours

The behavioural statements presented in Section 4 are represented in the form of automata when translated into Heptagon/BZR. In fact, as Heptagon/BZR allows for hierarchical composition of automata, we translate each behaviour statement defined inside another behaviour as sub-automaton. That way, we can hierarchically decompose the whole behaviour into smaller pieces until we get to an indivisible behaviour, that is, a configuration.

The top-most automaton i.e., the automaton modeling the whole behaviour consists as a two-state model, as depicted in Figure 8. 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.

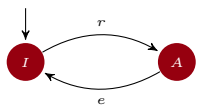


Figure 8: Top-most Automaton for Behaviours.

5.2.1 When-Do

The automaton that models the statement *when-do* (cf. Figure 9) consists of a 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 it has been declared before B_2.

5.2.2 Case and Alternative

Both behaviour statements *case* and *alternative* can be modeled by the automaton shown in Figure 10. As the sub-behaviour statements should be executed at the very first instant upon the request of the *case* or *alternative* statement, the automata should be composed in parallel with the automata modeling the behaviour (inside node `bcomp`, in Figure 7) , otherwise it would be needed two steps to go through state W and decide which branch to take, before actually

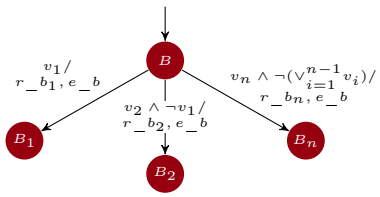


Figure 9: Automaton for *When-Do* Statement.

executing it (i.e., the sub-behaviour corresponding to the chosen branch). In fact, a *case* or an *alternative* statement is modeled as simple state inside the automata hierarchically composed that model the behaviour. Upon a request to those statements, 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. The difference between the use of this automaton for a *case* or an *alternative* statement is that for the latter the conditions c_i will be considered as a *controllable* variables in Heptagon/BZR. Thus, a discrete controller synthesis (DCS) should be performed to guarantee a behaviour such that the stated policies are not violated.

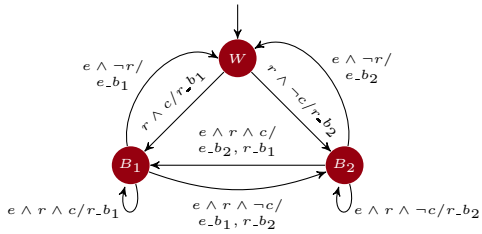


Figure 10: Automaton for *Case* and *Alternative* Statements to Be Composed in Parallel.

5.2.3 Every

The automaton model for the *do-every* statement is straightforward, as it is shown in Figure 11. 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.

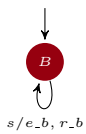


Figure 11: Automaton Modeling the *Every* statement.

5.2.4 Parallel

Finally, Figure 12 presents the model for the *Parallel* statement. It consists simply in the parallel composition of two or more sub-automata.

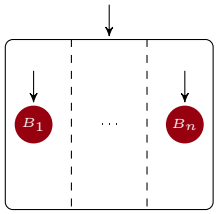


Figure 12: Automaton for *Parallel* Statement.

5.2.5 Znn.com Example

In order to ease the understanding of the translation from behaviour statements defined in Ctrl-F to Heptagon/BZR automata, Figure 13 illustrates how the translation can be applied to the behaviour defined for component *AppServer* (cf. Listing 7) of the *Znn.com* example. 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 8. 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_b_1 or r_b_2) for the next configurations and end signals (e_b_1 or e_b_2) 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*.

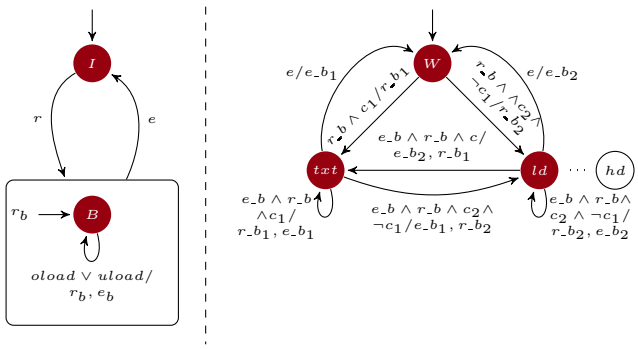


Figure 13: Translation of the component *AppServer* behaviour.

In the generated executable 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 6).

5.3 Modeling Policies

As previously stated in Section 4, Ctrl-F policies can be categorized into constraints/optimization objectives on attributes or temporal constraints (i.e., constraints on the order) over configurations.

5.3.1 Constraints/Optimization on Attributes

Constraints on attribute values can be translated into Heptagon/BZR in a very straightforward way. Indeed, they correspond to a set of boolean equations defined within the nodes that model components where the policies are stated. Then, the references to the equations are used inside the *enforce* block of a contract in order to state that they always hold by the control on the values of controllable variables.

Listing 12 shows an example of how a policy is translated into Heptagon/BZR. It refers to a policy on the attributes consumptions defined inside the *main* component of *Znn.com* example (cf. Listing 9, line 10). Basically, it states an equation (line 9) that depends on the integer outputs `soccer_consumption` and `politics_consumption`, which are produced by the respective instances of node `znn` (lines 7 and 8). Finally, as shown, this equation is used in the enforce block of the contract (line 3).

Listing 12: Example of Constraint on Attribute in Heptagon/BZR.

```
1  node main(r,e:bool;...) returns(...,p1:bool)
2  contract
3  enforce p1 and ...
4  with (...)
5  let
6     ...
7     (...,soccer_consumption)=znn(...);
8     (...,politics_consumption)=znn(...);
9     p1=(soccer_consumpton + politics_consumption) < 3
10    ...
11 tel
```

The declaration of optimization objectives are currently not supported by Heptagon/BZR. However, there are means to model an one-step optimization directly within the Discrete Controller Synthesis (DCS) tool (e.g., Sigali [19] or Reax [3]) Heptagon/BZR relies on.

5.3.2 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`), a 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 becomes necessary. 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 synthesis tool will interpret this as a new synthesis objective: the invariance of the states 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 $conf_1$ and $conf_2$, respectively. The error state (E) is reached if $conf_2$ terminates before $conf_1$ ($e_2 \wedge \neg e_1$) or if $conf_2$ terminates before $conf_1$ have started.

The observer that models the constraint between (`conf1 between (conf1,conf2)`) 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 $conf_1, conf_2$ and $conf_3$, respectively. The automaton goes to the error state (E) whenever configuration $conf_3$ is started (r_3 is emitted) after configuration $conf_2$ (e_2), except when configuration $conf_1$ is started and terminated (r_1 and e_1) in the between.

(a) During (b) Between

Figure 14: Observer Automata for Temporal Constraints.

6 Implementation

This section presents how Ctrl-F language is implemented and integrated into a target component platform. First, we give an overview on the component model and platform middleware we rely on. Then we present the Ctrl-F tool-chain as well as how the code resulting from the compilation is wrapped into a management software that actually enacts adaptation on the target system. Finally, we show the applicability of our language through an adaptation scenario over the *Znn.com* application.

6.1 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) [2] 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 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,

[2] `http://www.oasis-opencsa.org/`

they should be promoted to composite (or external) services/references/properties. Bindings define which methods and/or transports (e.g., HTTP, SOAP [3]) are allowed to access services or to be accessed by references. Lastly, service and reference are connected through wires.

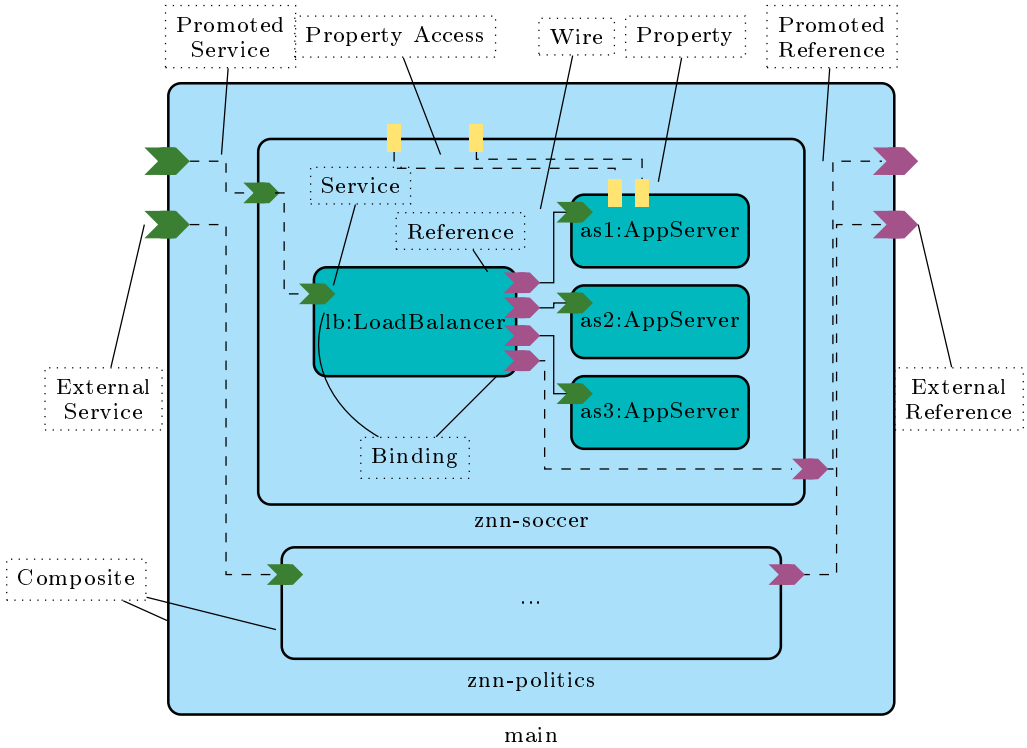


Figure 15: Service Component Architecture Concepts in Znn.com.

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 [23], since 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 [8], a domain-specific language for introspection and dynamic reconfiguration of Fractal components.

6.2 Compilation Tool-chain and Manage Prototype

The Ctrl-F compilation tool-chain is depicted in Figure 16. As can be seen, 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 also as input a Ctrl-F definition and generates as output an FPath/FScript script containing a set procedures allowing going from/to all configurations. Listing 13 shows an example of script describing a reconfiguration from configuration *conf1* to *conf2* of *Znn* component. In

[3] `http://www.w3.org/TR/soap/`

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 13: Reconfiguration Logics in FScript.

```
action conf1_conf2(znn){
  stop($znn/scachild::lb);
  r=$znn/scachild::loadbalancer/scareference::as2;
  s=$znn/scachild::as2/scareference::s;
  addscawire($r,$r);
  start($znn/scachild::as2);
  start($znn/scachild::loadbalancer);
}
```

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

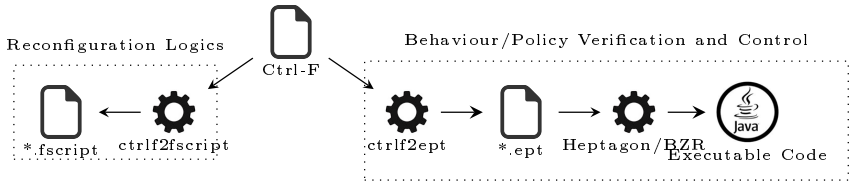


Figure 16: Ctrl-F Compilation Chain and Manager Wrapper

These methods are typically used by first executing `reset` and then by enclosing `step` in an infinite loop, in which each interaction correspond to a reaction to an event (e.g., *oload* or *uload*), as sketched in Listing 14. 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 FPath/FScript script that embodies the reconfiguration actions to be executed (lines 5 and 6).

Listing 14: 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);
```

As illustrate in Figure 6.2, we wrap the control loop logics into three SCA components, which are, in turn, 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 is defined in SCA as *one-way*, menaing that the calls are non-blocking so the incoming events are stored in a First-In-First-Out queue. Upon the arrival of an event provenient from the *Managed System* (e.g., Znn.com), component *EventHandler* invokes the step method, which is implemented by component *Architecture Analyzer*. The step method outptut is sent to component *Reconfigurator*, that encompasses a method to find the proper reconfiguration script to

be executed by the FraSCAti middleware component *frascati-fscript*. The *frascati-fscript* component relies on other components integrating the middleware, inside the FraSCAti Composite, to perform introspection and runtime reconfiguration on the managed system's components.

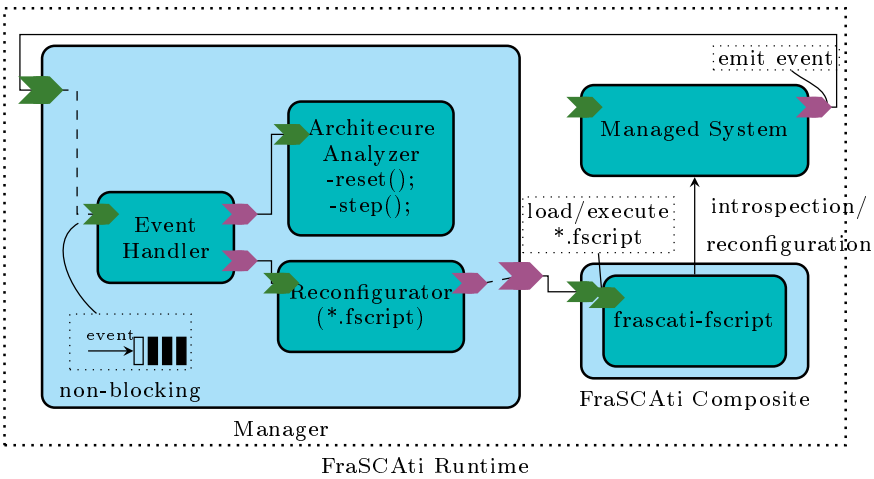


Figure 17: Manager Prototype Wrapping the Control Loop.

6.3 Znn.com Adaptation Scenario

We simulated the execution of the two instances of *Znn.com* application, namely *soccer* and *politics*, under the administration the *Manager* presented in the last section. The objective was to observe how reconfigurations are controlled, taking into account a sequence of input events as well as the stated behaviours and properties. The behaviours for components *AppServer* and *Znn* were as stated in Listings 7 and 8, respectively, whereas the properties are as defined in Listing 9.

As it can be observed in the first chart of Figure 6.3, we scheduled a set of overload (*oload*) and underload (*uload*) events (vertical dashed lines), which simulates an increase followed by a decrease of the income workload for both soccer and politics instances.

The other charts depicted in Figure 6.3 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*, namely *politics* and *soccer*, respectively.

As the workload of *politics* increases, an event of type *oload* is fired 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 component *Znn*. 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 when the last *oload* event arrives, the *soccer* instance must not only to reconfigure to *conf3* so as to cope with the current workload, but to keep the its fidelity level at the maximum (i.e., *img-hd* for all the instances of *AppServer*). That forces *politics* instance to degrade the fidelity level of two of its instances of *AppServer* (*as2* and *as3*). At step 9, the first *uload* is fired as a consequence of the workload decrease. It triggers a reconfiguration in *politics* instance as it goes from *conf3* to *conf2*, that is, it releases one instance of *AppServer* (*as3*, which makes room on

the resources and therefore allows it to bring back its fidelity level to the maximum level again. This is followed by a workload decrease of the *soccer* instance at step 10, which makes it release one instance of *AppServer* (*as3*). This is repeated again at steps 13 and 14 for instances *politics* and *soccer* respectively, bringing their levels of *consumption* and *fidelity* at the same levels as in the beginning.

While the *Znn.com* example was useful to show the power of expressiveness of Ctrl-F all along the Sections 4, the adaptation scenario helped us to understand, in a pedagogical way, the dynamics behind the Ctrl-F high-level description of adaptive behaviours and policies, that is, how reconfigurations in component-based architecture can actually be controlled in a logic-temporal manner. It is important to remind that although *Znn.com* does not require complex adaptation behaviours, the integration with synchronous reactive programming based on FSA allows Ctrl-F, among other things, to be able to control applications while taking control decisions based sequences of states to take a decision, that is, systems with very complex behaviours.

7 Related Work

In the literature, there is a large and growing body of work on self-adaptive software systems. Our approach focuses on the language support for enabling self-adaptation in component-based architectures while relying on reactive systems and the underlying formal control tools for ensuring adaptation policies.

Classically, runtime adaption in software architectures is achieved by first relying on ADLs such as Acme [13] or Fractal [5] for an initial description of the software structure and architecture, then by specifying fine-grained reconfiguration actions with the help of dedicated languages like Plastik [1] or FPath/FScript [8] to lead the target 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.

Rainbow [12] is an autonomic framework for that comes with Stitch, a domain-specific language allowing for the description of self-adaptation of Acme-described applications. Basically, the language groups a set of system-level actions (add, remove, bind component) into *tactics*, which in turn, are aggregated within a tree-like strategy path. The strategy path actually regroups a set of adaptation steps, whose branches are selected according to runtime conditions and the leaves corresponds to the end of a strategy, that can be evaluated as successful or failed. We can draw an analogy between tactics and the set of actions triggered upon a reconfiguration; as well as strategies and behaviours in the Ctrl-F language. Nonetheless, besides the conditions-based behaviours (e.g., the *case* statement), our language provides an interesting set of behavioural statements including alternative and parallel, as well as the event-based ones like *every* and *when-do*. That makes Ctrl-F more expressive in terms of self-adaptation behavioural definition. Furthermore, thanks to the Ctrl-F's formal model in the reactive language Heptagon/BZR, it is possible to ensure correct adaptation behaviours either by verification or by enforcement, that is, by the generation of correct-by-construction controllers that enforce such behaviours.

In [21], 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. The approach relies 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. [22] propose an approach for optimal resolution of architectural variability specified in the Common Variability Language (CVL) [15].

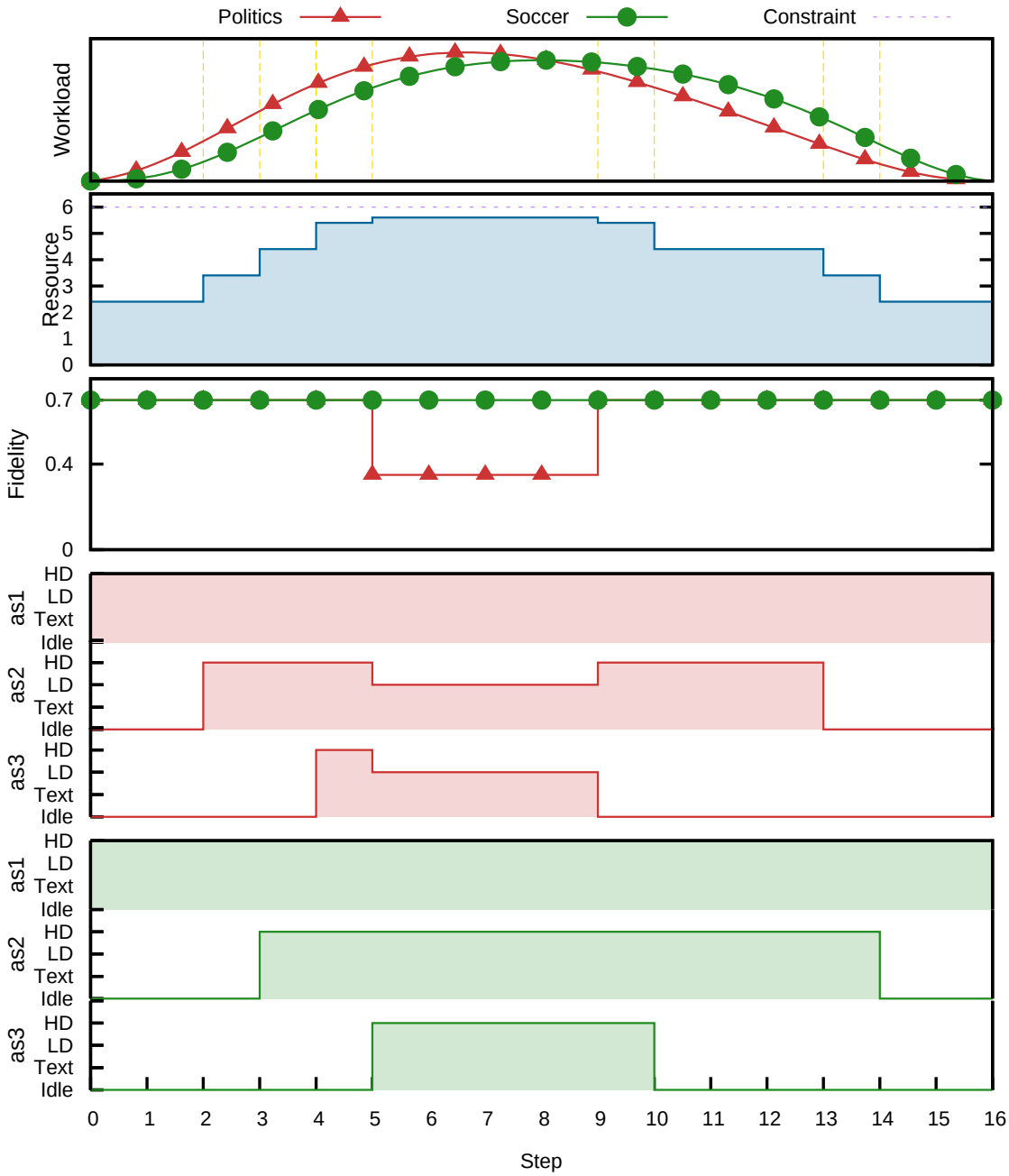


Figure 18: Execution of the Adaptation Scenario.

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. While in our approach, in the underlying synchronous reactive

model based on FSA, decisions are taken also based on the history of configurations which allows us to define more interesting and complex adaptation behaviours and policies.

In the domain of formal methods, Kouchnarenko and Weber [18] proposes the use of temporal logics to integrate temporal requirements to adaptation policies. The policies are used for specifying reflection or enforcement mechanisms, which refers respectively to corrective reconfiguration triggered by unwanted behaviours, and avoidance of reconfigurations leading to unwanted states. While in this approach, enforcement and reflection are performed at runtime in order to ensure correct behaviour, we rely on discrete controller synthesis so as to have correct-by-construction controllers ensuring both adaptation behaviours and policies.

As in our approach, in [11] and [4], authors also rely on Heptagon/BZR to model adaptive behaviours of Fractal components. 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 autonomic manager that bridges Ctrl-F and a real component platform. Delaval et al. [9] proposes the use of components to embody autonomic managers conceived with synchronous reactive programming. 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 in the sense that it does not provide high-level language support for describing those managers, although the authors provide interesting intuitions on a methodology to do so. Either Ctrl-F provides concepts dedicated for the specification of coordination between components' control. We do believe however that coordination is a major challenge that should be tackled by any modular self-adaptive software system . Hence the integration of coordination aspects to Ctrl-F should be seriously considered in future work (cf. Section 8).

8 Conclusion

Initiatives like autonomic computing advocate the use of feedback control loops in order to enrich software system with self-adaptive capabilities and thus cope with changing requirements and dynamical environments at runtime. In this direction, component-based software engineering has played a major role, since it enables the development of reusable, modular, reconfigurable and adaptable applications. In fact, by relying on architecture description languages, one can define initial configurations, while programing adaptation behaviours as routine of fine-grained actions on languages dedicated for the reconfiguration of architectural elements. Besides the complexity of dealing with fine-grained reconfiguration actions, especially in large architecture; another negative consequence of such an approach is that the space of reachable configurations is only known as a after-effect of those fine-grained actions, which makes it hard to ensure correction on the adaptive behaviours.

This paper presented Ctrl-F: a high-level domain-specific language that allows for the description of adaptation behaviours and policies of component-based architectures. A distinguished feature of Ctrl-F is that it is formalized with the synchronous reactive language Heptagon/BZR, and hence we can benefit, among other things, from formal tools for verification, control, and automatic generation of executable code. In order to show the expressiveness of our language, we applied it Znn.com, a self-adaptive news website; and integrated it with FraSCAti, a Service

Component Architecture middleware.

In addition to applying and experimenting Ctrl-F with other case studies and benchmarks, we indent to pursuit research to address issues related to modularity and coordination of controllers. We believe that there might be cases where the modular control of components becomes necessary. In particular, for communication cost reasons, one may want to distribute or deploy controllers in the same physical place (e.g., a distributed node) of the component it is intended to control. Moreover, one may want that (distributed) controllers react to their environments at their own paces. That creates a sort of asynchrony and independence among controllers that might be harmful, especially if they have conflicting behaviors. In this regards, it becomes imperative to conceive mechanisms for the coordination among component controllers as well as the appropriate high-level language constructs. Prior work such as [9] seem to be a good start point.

Contents

1 Introduction **3**

2 Background **4**
2.1 Component-based Architecture 4
2.1.1 Basic Concepts 4
2.1.2 Dynamic Reconfiguration 4
2.1.3 Service Component Architectures 5
2.2 Reactive Programming 6
2.2.1 Heptagon 6
2.2.2 Contracts and Discrete Controller Synthesis 7
2.2.3 Compilation and code generation 8

3 Example Application **9**
3.1 Overview of Znn.com 9
3.2 Znn.com Instantiation 10

4 Ctrl-F Language **10**
4.1 Overview and Common Concepts 10
4.2 Behaviours 14
4.2.1 Statements 14
4.2.2 Example in Znn.com 15
4.3 Policies 16
4.3.1 Constraints/Optimization on Attributes 16
4.3.2 Temporal Constraints 17

5 Modeling Ctrl-F in Heptagon/BZR **17**
5.1 General Model: The Component 18
5.2 Modeling Behaviours 20
5.2.1 When-Do 20
5.2.2 Case and Alternative 20
5.2.3 Every 21
5.2.4 Parallel 21
5.2.5 Znn.com Example 22
5.3 Modeling Policies 23
5.3.1 Constraints/Optimization on Attributes 23
5.3.2 Temporal Constraints 23

6 Implementation **24**
6.1 FraSCAti and Service Component Architecture 24
6.2 Compilation Tool-chain and Manage Prototype 25
6.3 Znn.com Adaptation Scenario 27

7 Related Work **28**

8 Conclusion **30**

References

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

[2] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, Jan 2003.

[3] N. Berthier and H. Marchand. Discrete controller synthesis for infinite state systems with reax. In *IEEE International Workshop on Discrete Event Systems*, pages 46–53, Cachan, France, May 2014.

[4] T. Bouhadiba, Q. Sabah, G. Delaval, and 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, pages 309–318, New York, NY, USA, 2011. ACM.

[5] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and 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, May 2004.

[6] S.-W. Cheng, D. Garlan, and 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*, pages 132–141, May 2009.

[7] J.-L. Colaço, B. Pagano, and 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, pages 173–182, New York, NY, USA, 2005. ACM.

[8] P.-C. David, T. Ledoux, M. Léger, and 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*, 2008.

[9] G. Delaval, S. M.-K. Gueye, E. Rutten, and 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, pages 3–12, New York, NY, USA, 2014. ACM.

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

[11] G. Delaval and E. Rutten. Reactive model-based control of reconfiguration in the fractal component-based model. In *13th International Symposium on Component Based Software Engineering (CBSE 2010)*, Prague, Czech Republic, June 2010.

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

[13] D. Garlan, R. Monroe, and D. Wile. Acme: An architecture description interchange language. In *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '97, pages 7–. IBM Press, 1997.

[14] D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural description of component-based systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

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

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

[17] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, Jan. 2003.

[18] O. Kouchnarenko and J.-F. Weber. Adapting component-based systems at runtime via policies with temporal patterns. In J. L. Fiadeiro, Z. Liu, and J. Xue, editors, *FACS 2013, 10th Int. Symposium on Formal Aspects of Component Software, Revised Selected Papers*, volume 8348 of *LNCS*, pages 234–253, Nanchang, China, 2014. Springer. Revised Selected Papers.

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

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

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

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

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

[24] I. Warren, J. Sun, S. Krishnamohan, and 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, pages 37–46, Washington, DC, USA, 2006. IEEE Computer Society.

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399