# Language Support for Modular Autonomic Managers in Reconfigurable Software Components

Frederico Alvares
IMT Atlantique & INRIA, France
Email: frederico.alvares@imt-atlantique.fr

Gwenaël Delaval & Eric Rutten
Univ. Grenoble Alpes, INRIA, CNRS,
Grenoble INP, LIG, Grenoble, France
Email: {firstname.lastname}@inria.fr

Lionel Seinturier
University of Lille & INRIA, France
Email: lionel.seinturier@inria.fr

***Abstract*—Dynamic reconfiguration is a key capability of Component-based Software Systems to achieve self-adaptation as it provides means to cope with environment changes at runtime. The space of configurations is defined by the possible assemblies of components, and navigating this space while achieving goals and maintaining structural properties is managed in an autonomic loop. The natural architectural structure of component-based systems calls for hierarchy and modularity in the design and implementation of composites and their managers, and requires support for coordinated multiple autonomic loops.**

**In this paper, we leverage the modularity capability to strengthen the Domain-Specific Language (DSL) Ctrl-F, targeted at the design of autonomic managers in component-based systems. Its original definition involved discrete control-theoretical management of reconfigurations, providing assurances on the automated behaviors. The objective of modularity is two-fold: from the design perspective, it allows designers to seamlessly decompose a complex system into smaller pieces of reusable architectural elements and adaptive behaviours. From the compilation point of view, we provide a systematical and generative approach to decompose control problems described in the architectural level while relying on mechanisms of modular Discrete Control Synthesis (DCS), which allows us to cope with the combinatorial complexity that is inherent to DCS problems. We show the applicability of our approach by applying it to the self-adaptive case study of the existing RUBiS/Brownout eBay-like web auction system.**

## I. Introduction

Nowadays software systems have to fulfill a number of requirements in terms of operational cost, safety and quality of service (QoS) while facing highly dynamic environments (e.g., varying workloads and changing user requirements) and platforms (e.g., software/hardware resource availability and failures). That level of dynamicity together with the omnipresence and heterogeneity of modern software systems makes it imperative to engineer and architect these systems with principles of self-adaptiveness in mind. In this context, software components have attracted much attention over the last years as a seamless way to develop more flexible and configurable software systems [1], [2].

Self-adaptive behaviours in component-based applications often take the form of feedback control loops [2], [3], architecturally organized in Autonomic Managers (AM). These loops manage the navigation through the configuration space (i.e., assemblies of architectural elements, parameters, etc.) while achieving goals and maintaining structural properties at runtime. Decisions are taken periodically and/or in an event basis considering the observed past history (e.g., events, configurations) as well as behaviours defining the components' adaptive logics and policies that have to be enforced in the managed systems at runtime. In previous work [4][5][6], we relied on discrete control to conceive well-mastered AMs, with assurances on the way the navigation through the configuration space is performed. The *specification* is given with the support of Ctrl-F, a Domain-specific Language (DSL) allowing designers to describe components, assemblies, and their reconfigurations [5]. From the *specification*, in a first step, behavioural programs are automatically translated to a Finite State Automata (FSA) *models* as well as their associated declarative correctness invariants (contracts) [4]. In a next step of automated synthesis, these specifications are compiled into controllers enforcing the invariants with guarantees of correctness at *execution* time. This generative compilation flow can be seen at the top of Figure 1.

Ctrl-F extends classical software component-based ADLs like Fractal [7], by featuring first-class language support of reconfiguration behavior in component-based languages, which is achieved by explicitly declaring the possible components assemblies as configurations, and the events and conditions governing transitions between them. Our approach extracts the behavioral part of Ctrl-F and generates automatically FSA-based behavioral models, as well as control objectives, concretely represented in the Heptagon/BZR reactive language [8]. These models contain knowledge not only on the past history, but also on the space of reachable configurations, which is useful to control the managed application so as to avoid it to go into branches of adaptive behaviours leading to bad configurations. To that end, the compilation of Heptagon/BZR involves the transformation of the control objectives in contracts into executable code enforcing them, by the automated synthesis of controllers based on techniques stemming from Control Theory, namely the supervisory control of Discrete Event Systems and Discrete Controller Synthesis (DCS) [9]. This produces a reconfiguration management that is correct w.r.t. specifications. The compilation process eventually generates an executable code (in C or Java) for the model itself and the reconfiguration controller, which is then integrated in a concrete component-based runtime, with middleware-level execution management, e.g., FraSCAti [10].

Our past results covered DSL and the underlying behavioural modeling support for reconfiguration control in software components, but we have so far concentrated in

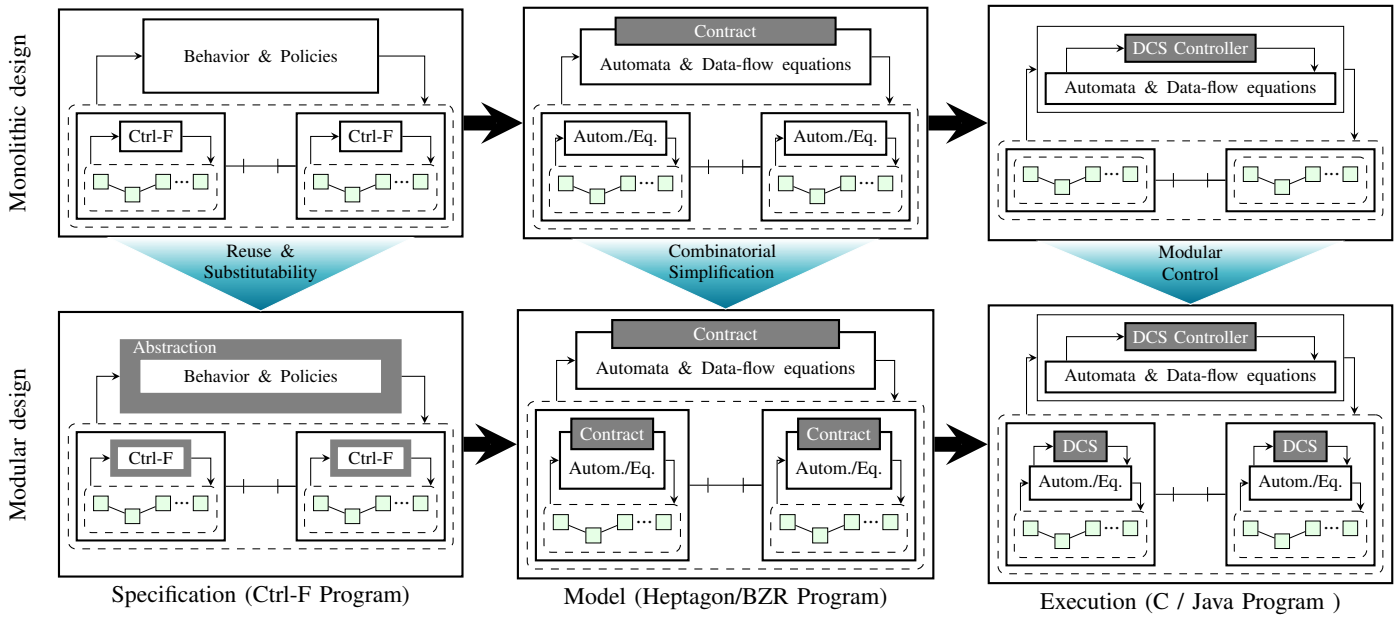



Fig. 1. Overview of our Approach: leveraging modularity in Ctrl-F, from specification, to model and compilation, to execution.

designing a single-level, monolithic solution, as depicted in the monolithic view on top of Figure 1. In fact, in spite of the fact that Ctrl-F, in previous work, already featured decoupling of behaviours in a hierarchical fashion, the corresponding models in Heptagon/BZR were built in a monolithic way, so was their compilation into the executable program and the DCS. The reason behind this is that behaviours and policies models need to be fully visible so that formal tools can perform model checking and DCS. Therefore, there is a lack of abstraction all the way through Ctrl-F approach from the specification to the executable code generated by Heptagon/BZR, which prevents modular autonomic components.

The need for modularity is strongly motivated in all the steps of autonomic components. First, by enabling modularity, we increase the possibility to reuse a component during its *specification*. Second, our approach relies on complex exponential DCS algorithms, whose combinatorial explosion can be broken thanks to modularity in the Heptagon/BZR *models*. Last, but not least important, having modular DCS allows for the distribution of synthesized controllers, i.e., the executable models and controllers can be deployed along with the managed components.

Our present work's contribution is to fulfill the needs explained above in the generative programming setting of Ctrl-F. We manage to integrate modularity in Ctrl-F so that it can exploit the particular notion of modular control in Heptagon/BZR in a way at the same time expressive and efficient. As sketched in the modular view of Figure 1, from the *specification* perspective, we introduce modularity constructs in the language allowing the abstraction of behavioural programs. This enables us to establish *contracts* in Heptagon/BZR at every level of the component hierarchy, ensuring an abstract behavioural model that can be controlled from the outside, i.e., by super-components. Finally, that way of *modeling* controllable behaviours allows for the synthesis of controllers in modular way, to be featured at runtime at every level of component hierarchy. The use of the modular version of Ctrl-F is illustrated with the RUBiS/Brownout [11], [12] case study, an eBay-like case study. We also experimentally compare the modular and monolithic Ctrl-F compilations in terms of compilation performance and behaviour execution.

In the remainder, Section II presents the concepts required for our contribution. Section III shows how Ctrl-F is extended with modularity-enabling features. The translation from Ctrl-F to Heptagon/BZR to benefit from modular compilation and DCS is detailed in Section IV. Section V presents the research prototype and analyzes the impact of modularity in Ctrl-F programs. Related Work is examined in Section VI, and Section VII concludes and provides research perspectives.

## II. Background

### A. The Ctrl-F Language

Ctrl-F is a domain-specific language that extends classic ADLs with high-level constructs for describing reconfigurations' behaviour and policies to be enforced all along the execution of the target system. The abstract syntax of Ctrl-F can be divided into a static one, which is related to the common architectural concepts (components, connections, configurations, etc.), in line with classical ADLs like Fractal [7]; and a dynamic one, which refers to reconfiguration behaviours and policies that must be enforced regardless of the configuration. Concretely, a Ctrl-F component $X$ consists of a tuple $(P_o^X, P_i^X, I_s^X, I_c^X, A^X,\ B^X, P^X)$, where $\{po_1, po_2, \dots\} \subseteq P_o^X$ and $\{pi_1, pi_2, \dots\} \subseteq P_i^X$ correspond respectively to ports used by the component to emit events produced by itself or to listen events produced by other components; $\{si_1, si_2, \dots\} \subseteq I_s^X$ and $\{ci_1, ci_2, \dots\} \subseteq I_c^X$ denote respectively the interfaces required and provided by the component; $\{a_1, a_2, \dots\} \subseteq A^X$

TABLE I
SUMMARY OF BEHAVIOUR STATEMENTS.

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

represents the set of attributes, to which values are assigned; $\{conf_1, conf_2, \dots, b_1, b_2, \dots\} \subseteq B^X$ corresponds to the behaviours expressing self-adaptation within the component. Notice that behaviours are defined as (sequential, parallel, alternative, etc.) compositions of configurations, which are themselves considered as behaviours. Lastly, $\{p_1, p_2, \dots\} \subseteq P^X$ represents the policies to be enforced at runtime.

Behaviors in Ctrl-F are defined imperatively and compositionally from behavior statements (i.e., involving *sub-behaviors*) and *configurations* that are considered as atomic behaviors. We assume that configurations do not have the capability to terminate or start themselves, meaning that they are explicitly requested or ended by super-behavior *statements*. Table I summarizes Ctrl-F's operators.

Ctrl-F's policies are high-level constructs that enable the definition of two kinds of constraints: (i) temporal (in the sense of the logical order of configurations) e.g., precedence: `conf1 -> conf2`, or imposed configuration sequencing: `conf1 between conf2,conf3`; and (ii) on component's attributes. Policies define a subset of all possible global configurations where the system should remain: it will be achieved automatically by using the choice points to control reconfiguration actions. Each program written in Ctrl-F is compiled so that the behavioral part is extracted and treated by generation of a program in Heptagon/BZR according to the semantics of the statements [4].

### B. The Ctrl-F compilation Toolchain

The compilation process can be split into two parts: (i) the reconfiguration logics, and (ii) the behaviour/policy control and verification. The reconfiguration logics is implemented by the *ctrlf2fscript* compiler, which takes as input a Ctrl-F definition and generates as output a script containing a set of procedures that implements the transitions from one configuration to another. To this end, we rely on existing differencing/match algorithms for object-oriented models [13]. The behaviour control 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. These methods are typically used by first executing `reset` and then by enclosing `step` in an infinite loop, in which each iteration corresponds to a reaction to an event. Each reaction produces a set of signals corresponding to the starting and stopping of configurations, which allows us to find the appropriate script with reconfiguration actions to be executed. We integrated the generated adaptive program in FraSCAti [10], a middleware enabling runtime reconfiguration on Service Component Architecture applications [14].

### C. Ctrl-F intermediate language: Heptagon/BZR

In our generative programming framework for Ctrl-F, in order to support its reconfiguration behavior part, Heptagon/BZR [8] serves as a tool-equipped intermediate language. This is a reactive language, whose compilation integrates a phase of discrete controller synthesis (DCS). DCS is a formal method, which allows to constrain a system with a generated controller, acting on controllable variables, so as to enforce temporal properties. The syntax of Heptagon/BZR includes a notion of contract, allowing to specify, for each node, the local property to be enforced, and on which controllable variables the local controller can act.

The use of contracts allows modular definitions of properties to enforce. When a node is instantiated at an upper level, its contract and the properties declared is used as abstraction of the internal effective behavior. Listing 1 shows an example of modularity: the `process` node is instantiated twice, modelling two parallel processes. At this level we consider that the requests (`r1`, `r2`) come from the environment, and the controller can act on the $s_i$ variables. The property we want to enforce is that only one process is active at any instant. Considering that DSC is of practical exponential complexity, this technique is of great interest. In fact, DCS itself is performed on a subpart of the system, the model from which the controller is synthesized can be much smaller than the global model of the system. Therefore, the gain in synthesis time can be high and it can be applied on larger and more complex systems.

Listing 1. Modularity example in Heptagon/BZR.

```
node twoprocesses(r1,r2:bool) returns (a1,a2)
  contract
  enforce not ((a1 = Active) & (a2 = Active))
  with (s1,s2:bool)
let
  a1 = process(r1,s1);
  a2 = process(r2,s2);
tel
```

## III. MODULARITY IN CTRL-F

Software components are modular in nature thanks to their compositional, and sometimes hierarchical, characteristics. Nevertheless, the lack of other means of modularity prevents to fully benefit from reuse and other properties like substitutability of both static and dynamic (i.e., behaviours and policies) parts of the component description. From the compilation point of view, it is also important to enable Ctrl-F language — and the underlying autonomic framework — to benefit from modularity of the Heptagon/BZR model, which allows

to enforce, by DCS methods, the inheritance or substitutability properties. This section introduces modularity in Ctrl-F.

### A. Modularity, reuse and substitutability

Modularity is a good way of fostering reuse. Breaking down a system in some modular parts enables to reuse these parts more easily across different systems. In relation with this increased reusability, modularity also enables to consider substitutability. Some parts of a system can be substituted so that a component can replace another one. This substitutability principle, which is also well-known in object-oriented software development with the inheritance relationship, enforces a good design principle of generalization and specialization. By starting with some high level goals, the component is progressively refined in more specialized versions. This mechanism focuses on the behaviour of the component and is clearly adequate for Ctrl-F whose purpose is to enable to reason about the behaviour, in terms of control policy, of software systems. We then define below such an inheritance relationship in Ctrl-F as a support to foster modularity. We extend Ctrl-F with the notion of inheritance by allowing components to be defined based on existing components. More concretely, if $Y$ *extends* (inherits from) $X$, `Y` contains all the elements contained in `X`, i.e., $P_o^Y = P_o^X \bigcup \bar{P}_o^Y, P_i^Y = P_i^X \bigcup \bar{P}_i^Y, ..., B^Y = B^X \bigcup \bar{B}^Y, P^Y = P^X \bigcup \bar{P}^Y$, where tuple $(\bar{P}_o^Y, ..., \bar{B}^Y, \bar{P}^Y)$ corresponds to the set of Ctrl-F component elements defined within the component `Y` itself, i.e., not those inherited from `X`.

Component `Y` extends from `X` by adding one attribute (`ext_a`) and two more configurations, which express two different assemblies built upon the two existing ones, that is, the addition of one instance of component `C4` to `conf1` and one instance of `C5` to `conf2`. At the end, there will be four assemblies, which will assign different values to the attributes (i.e., `a_1` and `ext_a`). By following the same idea, one could define other components (e.g., `Z`) that inherit from `X` to accommodate other configurations and attributes.

It is often useful to redefine parts of a component behaviour or some of its configurations, especially if one wants to reuse just a portion of the behaviours defined in a higher level. In Ctrl-F, this is possible by overriding the behaviours/configurations. More concretely, let $b^{ID}$ be the id of behaviour $b \in B^X$, if `Y` *extends* `X`, $B^Y = \bar{B}^Y \bigcup B^X - \{b \mid b \in B^X \exists \bar{b} \in \bar{B}^Y \cdot \bar{b}^{ID} = b^{ID}\}$. All the elements of a super component `X` are accessible from its inherited components. Lastly, we can instantiate component `Y` by indicating its type or by also specifying its abstract/generic type. In the former, the architectural elements of `Y` are exposed and accessible from component `main`, whereas in the latter, only those of component `X` are visible. That is to say that the elements of `X` serve as an abstraction of the elements of `Y` [1].

### B. Ctrl-F Modularity in a Concrete Example

We applied our approach to RUBiS/Brownout [12], an extended version of the well known eBay-like auction site RUBiS [11] benchmarking. The extended version consists in allowing multiple recommendations to the users when they arrive or navigate through the site, which is a very common functionality provided by real e-commerce sites. The standard or nominal state of an application can be to provide recommendations to users, and switching it off in case of high variability of workload or scarcity of resource capacity. We build a self-adaptive component-based application that reacts to variations by gracefully downgrading the user experience from higher mode to lower mode and vice-versa.

Listings 2, 3 and 4 show how we model RUBiS/Brownout with Ctrl-F. We define `Brownout` (cf. Listing 2) as an abstract component whose adaptive behaviour consists in going from nominal configuration to degraded one (without recommendation), and vice-versa, at every occurrence of event `tick`. The latter corresponds to a composition of events `ri` (short for request increase) and `rd` (short for request decrease), which means that the target system emits an event indicating that there has been a substantial increase/decrease in the number of client requests in the system. The difference between nominal and degraded configurations resides in the fact that they incur different loads on the system. The `Brownout` component is defined as abstract because the `load` attribute is not defined at this level, but will be by components extending `Brownout`.

Listing 2. Brownout example in Ctrl-F.

```
abstract component Brownout {
    port in tick as (ri or rd)
    attribute max_load
    attribute load

    configuration nominal { set max_load to 20 }

    configuration degraded  { set max_load to 10 }

    behaviour alternative {
        nominal | degraded
    }

    initial behaviour init {
        do
           alternative
        every tick
    }

    policy { load <= max_load }
}
```

Component `RUBiS` is defined as an extension of `Brownout` (cf. Listing 3). The component contains business functionalities offered by RUBiS, i.e., auction service, and implements brownout strategies by allowing different levels of Quality of Experience (high, medium,low), each incurring different loads on the system. Those levels could correspond to different numbers of recommendations enabled at a given time. The relationship between the abstract and inherited components are performed by the policy `load <= max_load`. The outside controller will force the local controller to go to either high, medium, low, according to the external conditions. Notice that in cases where all alternatives respect the stated policies, the controller's priority is given by the order of declaration, which is an implicit expression at the same time of controllability (with the alternative operator) and priority (with order of declaration) over configurations. Here, priority is given to configurations with highest quality, from `high` to `low`.

[1]Further details on the concrete syntax of the modular version of Ctrl-F are available on https://www.dropbox.com/sh/1u2z2nhziogyfr5/AADVlI44MiscBiKD46hy29Yqa?dl=0

Listing 3. RUBiS example in Ctrl-F.

```
component RUBiS extends Brownout {
    configuration high  { set load to 20 }

    configuration medium  { set load to 10 }

    configuration low  { set load to 0 }

    behaviour alternative {
        high | medium | low
    }
}
```

Listing 4. Component instantiating brownout component.

```
component main {
    port in ri, rd
    attribute global_load,
    attribute max_global_load default 3500

    configuration underloaded {
        Brownout inst1:RUBiS
        Brownout inst2:RUBiS
        ...
        set global_load to
            (50*(inst1.load + inst2.load + ...))
    }

    configuration overloaded {
        Brownout inst1:RUBiS
        Brownout inst2:RUBiS
        ...
        set global_load to
            (100*(inst1.load + inst2.load + ...))
    }

    initial behaviour init {
        underloaded when ri do
           (overloaded when rd do init end)
        end
    }

    policy {
        global_load <= max_global_load
    }
}
```

The RUBiS component is instantiated several times within component main (cf. Listing 4) for load balancing reasons. We define two configurations: underloaded and overloaded. They distinguish in the impact that the sub-components will have on the global load. In this example, the factors are of 50 and 100 and those are totally arbitrary numbers so is 3500 set as default value of attribute max_global_load, though those factors can be interpreted as a maximum number of requests, that is, the global load would be calculated in function of this factor (or maximum number of requests) and the sum of the load icurred by all sub-components. The behaviour is stated as follows: the component starts in configuration underloaded, it goes to configuration overloaded when an event ri occurs. From overloaded it goes back to the initial behaviour (and thus configuration underloaded) upon the occurrence of an event of type rd. Finally, the policy states that the global load should not exceed a certain threshold.

The controller generated in the component main's level will be in charge of controlling the underlying component instances based on that policy. This control governs the contract behaviour, which in turn guides the local controller of component RUBiS, i.e., based on the threshold (3500 in this case), when the component goes from underloaded to overloaded (resp. from overloaded to underloaded) it may force the sub-components to go from (resp. to) configuration nominal (resp. degraded), according to events ri and rd.

## IV. Implementing Ctrl-F Modularity using Heptagon/BZR

The modular compilation in Heptagon/BZR is achieved thanks to the hierarchical composability of both nodes and contracts, which ensures a prescribed underlying behavior assuming that some property at the environment holds. Thus, the property that links two components in an inheritance relation (the property denoted $\theta$ in Figure 2) can be enforced by controller synthesis, lifting the burden on the programmer of ensuring that these properties are actually satisfied. This section describes how the Ctrl-F compiler is extended to exploit these aspects, by automatic generation of Heptagon/BZR programs for modular DCS. Figure 2 depicts a general Ctrl-F to Heptagon/BZR translation scheme. An inherited component Y is modeled as an Heptagon/BZR node with a contract and behavior. It takes as input a request (r) and ending (e) signals, a set of events the component listens to ($v_i \in P_i^Y$) as well as a set of variables ($c_1, \ldots, c_q$) whose objective is to make the component controllable from outside. The node produces as output request and ending signals $(r_i, e_i)$ for each configuration $conf_i$ defined within the component.

In Heptagon/BZR, the interaction of components with their environment is achieved with contracts. This provides guarantees on the component's behaviour while assuming some property from the environment. Policies $p_1, \ldots, p_t \in P^Y$ defined within component Y are translated as predicates that have to be enforced by the contract with respect to the controllable variables $c'_1, \ldots, c'_n$ used in the body of the of the node, that is, as part of the component's behaviour. The contract also may explicit a behaviour in terms of automata, which in this case will correspond to the behaviour of the abstract component X. This behaviour serves as an abstraction (or simplification) of the component's actual behaviour that is defined in the body of the Heptagon/BZR node. Both behaviours (in the contract and in the body) are bound thanks to the extra property $\theta$ that also has to be enforced along with the other component's property. The idea is that the abstract behaviour (defined in the contract) can be controlled by its environment (via control variables $c_1, \ldots, c_n$) and a local controller, specified by the enforce/with of the contract, can locally control the component's actual behaviour (with controllable variables $c'_1, \ldots, c'_q$), while being guided by its abstract counterpart. In practice, this technique breaks the synthesis problem into subproblems that can be locally solved. Hence, the overall synthesis time is substantially reduced, enabling Ctrl-F to be applied to more complex cases.

Figures 3, 4 and 5 show how component Brownout, RUBiS and main are translated into Heptagon/BZR.

## V. Prototype and Performance Evaluation

### A. Research Prototype

The present work extends the prototype presented in our previous work [6] by taking profit of the modular code resulting from the Ctrl-F compilation chain. We took into

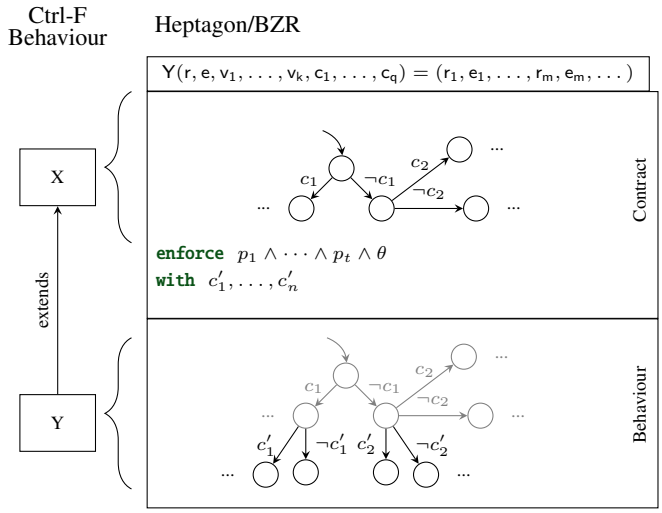



Fig. 2. General Translation Scheme.

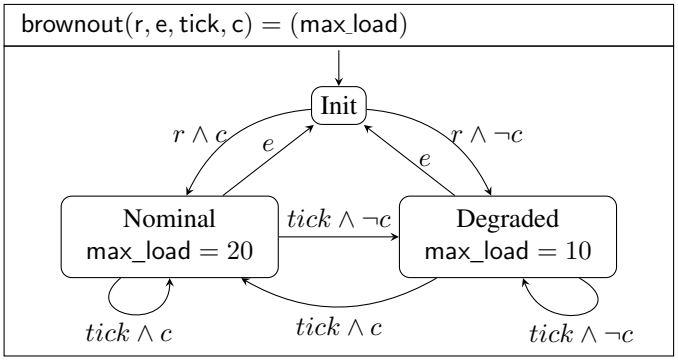



Fig. 3. Abstract Behaviour Translation Result.

consideration the possibility to encapsulate the generated executable code (corresponding to the FSA-models and synthetized controllers) separately within different components, and hence improving reuse, deployment and replacement of components encompassing the adaptive logics. The prototype relies on FraSCAti [10], a Java-based Service Component Architecture (SCA) (http://www.oasis-opencsa.org) middleware, which provides mechanisms for runtime reconfiguration.

Each Ctrl-F component is implemented as a SCA composite enclosing a *Manager* component, and the managed component itself. The former wraps the adaptation logics into four components constituting a MAPE-K control loop, whereas the managed component can be either simple (atomic) or composite, in which case it will follow the very same hierarchical structure recursively. A practical result of structuring managers in that way is that one can more easily replace or reuse managers. For example, suppose that the Ctrl-F behavioural program of a sub-component (e.g., *RUBiS*) is modified: we can then keep the manager of the super component (e.g., *main*), while replacing only the *Analyzer* and *Planner* of the sub-component. Similarly, if the behaviour of the super component changes, we can keep the managers of sub-components or reuse them, in case we want to use many instances of the same component. Further, we can do that without needing to

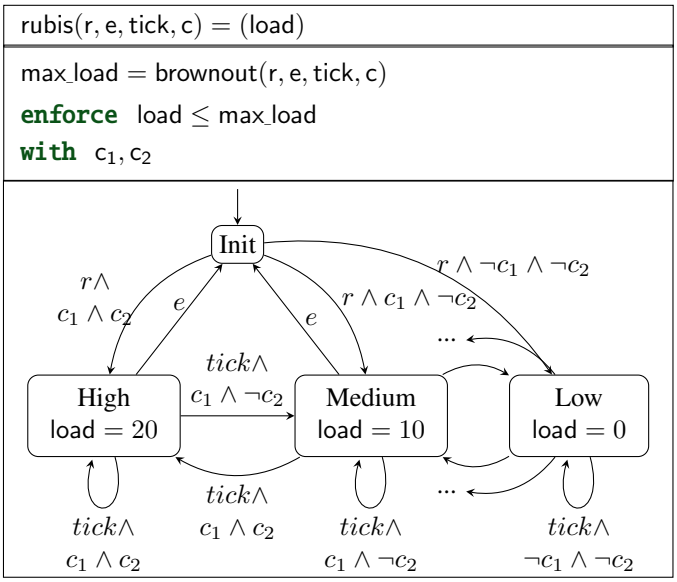



Fig. 4. Translation of component `RUBiS` to Heptagon/BZR.

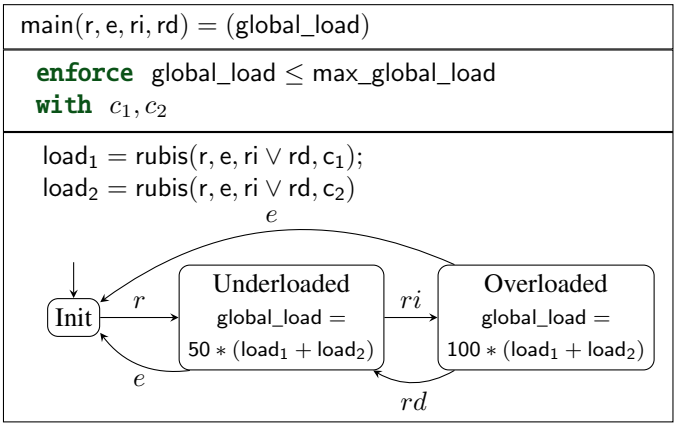



Fig. 5. Translation of component `main` to Heptagon/BZR.

recompile, and redeploy the whole application, which is not possible if the resulting generated code is monolithic.

### B. Compilation Performance Analysis

In order to evaluate the compilation performance, we executed a series of compilations of modular and monolithic versions of Ctrl-F programs, each of them was executed on a machine equipped with Intel 1.7GHz Core i7 processor, 8GB 1600 MHz DDR3, running MacOS 10.12.3. We generated several components structured in a hierarchical way. The components/composites at all levels oscillate between two modes: utilizing and not utilizing an exclusive resource. The control objective is then to ensure that no more than one (instance of) component in the entire hierarchy utilize that resource.

Table II shows the differences in terms of compilation time (CT, in seconds) and generated executable code size (CS, in KB) between the monolithic and modular approaches when varying the number of levels in the component hierarchy. As can be seen, the modular approach over-performs the

TABLE II
COMPILATION TIME AND CODE SIZE COMPARISONS.

| # Levels / # Configs. | Monolithic | | Modular | | |
|---|---|---|---|---|---|
| | CT (s) | CS (KB) | CT (s) | CS (MB) | ET ($\mu$s) |
| 3 / 32 | 0.44 | 270 | 0.23 | 120 | $\approx 10$ |
| 4 / 64 | 1.60 | 560 | 0.27 | 152 | $\approx 10$ |
| 5 / 128 | 15.1 | 884 | 0.30 | 180 | $\approx 10$ |
| 6 / 256 | 195.3 | 1800 | 0.36 | 208 | $\approx 10$ |
| 7 / 512 | 2506 | -[b] | 0.43 | 236 | $\approx 10$ |
| 10 / 4096 | -[a] | - | 0.69 | 480 | 526.8 |
| 50 / $> 10^{15}$ | -[a] | - | 4.73 | 1516 | -[b] |
| 100 / $> 10^{30}$ | -[a] | - | 13.90 | 3144 | -[b] |

[a]Not measured: compilation time higher than 3h
[b]Not measured: executable code too big to be compiled or executed

monolithic one in both compilation-time (including the DCS) and code size. The reason is that the combinatorial explosion of the DCS is broken down by the modularity applied to the hierarchical. Instead of exploring the entire configuration space, from the top-most composite to the single components, as a single monolithic DCS problem, the DCS tool benefits from the modular structure of behavioural models so as to perform DCS separately. That is, for a given composite, we only considered its own behavioral model and policies as well as the abstract behaviour of their immediate sub-components/composites. The generated executable code is also impacted, since the Heptagon/BZR compiler reuses already compiled modules (i.e., Heptagon/BZR nodes), instead of inlining copies of the same code, which is redundant. The last column shows the average execution time of one reaction step (ET, in $\mu$s). This reaction step comprises the computation of the controllable values by the synthesized controller, and the computation of the outputs by the code generated directly by the Heptagon/BZR compiler. This execution time is negligible, even with increasing-size controllers.

A recurrent issue when using modularity is that as sub-components are abstracted in the process, some information about their behavior is lost or hidden. Therefore, the controller synthesis computation, based on these abstracted behaviors, can be less precise, and thus lead to over-constrained controllers, or can even fail if the abstraction is not precise enough. For instance, in a monolithic version of the `main` node, without abstraction, a possible behavior would be to enforce the two components in the "Medium" mode. Due to abstraction, this behavior is not accepted in the modular version.

## VI. RELATED WORK

This section provides some discussions on a selection of relevant work related to our approach. Some works [15][16][17] propose modular interacting autonomic control loops structured in a hierarchical way. Contrary to those approaches, ours is not only structural, but also semantic in that behavioural models are explored all the way through the compilation chain, from the specification to the actual execution.

[18] and [19] propose a unified models for formally specifying self-adaptive systems. While the approaches are intended to serve as a (generic) reference model to specify self-adaptive systems of any domain, our approach focuses on a specific domain (i.e., software components). Rainbow [3], [20] provides an autonomic framework for Acme [21], [22] components. A DSL called Stitch is used to express autonomic behaviours (called strategies) in a tree-like manner. Our approach, however, provides an interesting set of behavioural statements which makes it more expressive in terms of behavioural definition. In addition, our approach is capable of ensuring correct adaptation behaviours.

A body of work [23], [24], [25], [26], [27] focus on how to plan a set of actions that safely lead systems to a target configuration. Hence, they are complementary to ours in the sense that our focus is on the choice of a new configuration and its control as the behavioural program progresses.

Kouchnarenko and Weber [28] propose the use of temporal logics to integrate temporal requirements to adaptation policies in the context of Fractal components [7]. While in those approaches, enforcement (resp. decisions over strategies' branches) and reflection are performed at runtime, in our approach, the decisional part of the autonomic manager is obtained in an off-line manner, through Heptagon/BZR. The advantage is that the costly decision is not done at runtime, but on the other hand dynamic model changes cannot be taken into account. In [29], the overhead incurred by verification at runtime can be mitigated by pre-computing probabilities as symbolic expressions [30], which is, to a certain extent, similar to what is done in the DCS technique. However, they focus on stochastic behaviours, whereas our model is deterministic; further, we provide an end-to-end DSL-based solution.

D'Ippolito et al. propose a multi-tier control approach in which one can define multiple models and controllers associated to different levels of assumptions (from the least to the most restrictive) and guaranteeable functionalities which are enabled/disabled at runtime. The same research group also proposes an approach for controller synthesis of event-based systems [31] and more recently a method to dynamically update synthesized controllers [32]. These approaches perform the controller synthesis by generating executable automata out of temporal logics formulae, i.e., it does not contemplate the notion of controllable and incontrollable, as in our DCS-based approach. Still regarding controller synthesis, Filieri et al. [33] [34] propose a methodology to automatically generate continuous controllers with formal guarantees from a set of high-level specifications, which differs from the DCS used in our approach in that the former deals with numerical and continuous variables, whereas our focus is on controlling logically the behaviour programmed within components.

As in our approach, in [35][36], the authors rely on Heptagon/BZR and DCS to model autonomic behaviours in the context of components to build modular coordinated controllers. We developed on top of these approaches to provide high-level language support.

## VII. CONCLUSION

This paper enhances Ctrl-F, a DSL to describe automic software components, by leveraging modularity capabilities. From the design point of view, we provide special constructs enabling more modular and consequently more reusable programs. The underlying formal tools can benefit from a modular

automata-based model so as to perform modular controller synthesis, which can break down the combinatorial explosion and thus improve scalability. Furthermore, the modular executable code resulting from the Ctrl-F compilation also allows for seamless integration with the runtime environment, which in our case is FraSCAti, a Service Component Architecture middleware. We illustrated our approach on the RUBiS/Brownout [11] [12] adaptive case study and performed a number of compilation in order to analyze the compilation performance, including DCS.

Perspectives include some potential applications of our results in the definition of specialized versions of Ctrl-F. In embedded systems, this concerns our work on model-based reconfiguration control in Dynamically Partially Reconfigurable hardware architectures based on FPGA [37]. In smart environments and the Internet of Things, it can add on to our activities in the management of the reconfiguration of functionalities and their implementation, on top of the rule-based middleware Linc. We believe that in both cases the specification and implementation of controllers can benefit from a DSL to produce well-formed models and objectives.

## REFERENCES


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

[2] J. Kramer and J. Magee, "Self-managed systems: An architectural challenge," in *2007 Future of Software Engineering*, ser. FOSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 259–268.

[3] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, no. 10, pp. 46–54, Oct. 2004.

[4] F. Alvares, E. Rutten, and L. Seinturier, "Behavioural model-based control for autonomic software components," in *Proc. IEEE Int. Conf. Autonomic Computing, ICAC*, 2015, pp. 187–196.

[5] ——, "High-level language support for reconfiguration control in component-based architectures," in *Proc. European Conf. Soft. Architecture, ECSA*, 2015, pp. 3–19.

[6] ——, "A domain-specific language for the control of self-adaptive component-based architecture," *Journal of Syst. and Software*, pp. –, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121217300201

[7] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "An open component model and its support in java," in *Proc. Int. Symp. on Component-based Software Engineering, CBSE*, May 2003.

[8] G. Delaval, É. Rutten, and H. Marchand, "Integrating Discrete Controller Synthesis into a Reactive Programming Language Compiler," *Discrete Event Dynamic Systems*, vol. 23, no. 4, pp. 385–418, 2013.

[9] P. Ramadge and W. Wonham, "On the supervisory control of discrete event systems," *Proc. IEEE*, vol. 77, no. 1, Jan. 1989.

[10] 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*, vol. 42, no. 5, pp. 559–583, 2012.

[11] "Rubis: Rice university bidding system," http://rubis.ow2.org, 2008, accessed: 2017-02-28.

[12] C. Klein, M. Maggio, K.-E. Årzén, and F. Hernández-Rodriguez, "Brownout: Building more robust cloud applications," in *Proc. 36th Int. Conf. on Software Engineering, ICSE*, 2014, pp. 700–711.

[13] Z. Xing and E. Stroulia, "Umldiff: An algorithm for object-oriented design differencing," in *Proc. 20th IEEE/ACM Int. Conf. on Automated Software Engineering*, ser. ASE '05, 2005, pp. 54–65.

[14] "Oasis open composite services architecture (csa)," http://www.oasis-opencsa.org, 2007, accessed: 2017-02-28.

[15] S. Frey, A. Diaconescu, D. Menga, and I. Demeure, "Towards a generic architecture and methodology for multi-goal, highly-distributed and dynamic autonomic systems," in *Proc. 10th Int. Conf. Autonomic Computing, ICAC*. USENIX, 2013, pp. 201–212.

[16] P. Vromant, D. Weyns, S. Malek, and J. Andersson, "On interacting control loops in self-adaptive systems," in *Proc. 6th Int. Symp. Soft. Eng. Adaptive and Self-Managing Systems, SEAMS*, 2011, pp. 202–207.

[17] D. Weyns, B. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. M. Göschka, *On Patterns for Decentralized Control in Self-Adaptive Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 76–107.

[18] D. Weyns, S. Malek, and J. Andersson, "Forms: Unifying reference model for formal specification of distributed self-adaptive systems," *ACM Trans. Auton. Adapt. Syst.*, vol. 7, no. 1, pp. 8:1–8:61, May 2012.

[19] M. Luckey, B. Nagel, C. Gerth, and G. Engels, "Adapt cases: Extending use cases for adaptive systems," in *Proc. 6th Int. Symp. on Soft. Eng. for Adaptive and Self-Managing Systems, SEAMS*, 2011, pp. 30–39.

[20] 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, SEAMS*, May 2009, pp. 132–141.

[21] D. Garlan, R. T. Monroe, and D. Wile, "Acme: Architectural description of component-based systems," in *Foundations of Component-Based Systems*, G. T. Leavens and M. Sitaraman, Eds., 2000, pp. 47–68.

[22] 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*, ser. CASCON '97. IBM Press, 1997, pp. 7–.

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

[24] C. E. da Silva and R. de Lemos, "Dynamic plans for integration testing of self-adaptive software systems," in *Proc. 6th Int. Symp. Soft. Eng. for Adaptive and Self-Managing Systems, SEAMS*, 2011, pp. 148–157.

[25] M. Tichy and B. Klöpper, "Planning self-adaption with graph transformations," in *Proc. 4th Int. Conf. on Applications of Graph Transformations with Industrial Relevance*, ser. AGTIVE'11, 2012, pp. 137–152.

[26] F. Boyer, O. Gruber, and D. Pous, "Robust reconfigurations of component assemblies," in *Proceedings of the 2013 International Conference on Software Engineering, ICSE*, 2013, pp. 13–22.

[27] S. Becker, S. Dziwok, C. Gerking, C. Heinzemann, W. Schäfer, M. Meyer, and U. Pohlmann, "The mechatronicuml method: Model-driven software engineering of self-adaptive mechatronic systems," in *Companion Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 614–615.

[28] O. Kouchnarenko and J.-F. Weber, "Adapting component-based systems at runtime via policies with temporal patterns," in *10th Int. Symp. Formal Aspects of Component Software*, vol. LNCS 8348, 2014, pp. 234–253.

[29] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola, "Self-adaptive software needs quantitative verification at runtime," *Commun. ACM*, vol. 55, no. 9, pp. 69–77, Sep. 2012.

[30] A. Filieri, C. Ghezzi, and G. Tamburrelli, "Run-time efficient probabilistic model checking," in *Proc. 33rd Int. Conf. on Software Engineering, ICSE*. ACM, 2011, pp. 341–350.

[31] N. D'ippolito, V. Braberman, N. Piterman, and S. Uchitel, "Synthesizing nonanomalous event-based controllers for liveness goals," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 1, pp. 9:1–9:36, Mar. 2013.

[32] L. Nahabedian, V. Braberman, N. D'Ippolito, S. Honiden, J. Kramer, K. Tei, and S. Uchitel, "Assured and correct dynamic update of controllers," in *Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '16. New York, NY, USA: ACM, 2016, pp. 96–107.

[33] A. Filieri, H. Hoffmann, and M. Maggio, "Automated multi-objective control for self-adaptive software design," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 13–24.

[34] ——, "Automated design of self-adaptive software with control-theoretical formal guarantees," in *Proceedings of the 36th International Conference on Software Engineering, ICSE*, 2014, pp. 299–310.

[35] T. Bouhadiba, Q. Sabah, G. Delaval, and E. Rutten, "Synchronous control of reconfiguration in fractal component-based systems: A case study," in *Proc. 9th ACM Int. Conf. Embedded Software*, 2011.

[36] G. Delaval, S. M.-K. Gueye, E. Rutten, and N. De Palma, "Modular coordination of multiple autonomic managers," in *Proc. 17th Int. ACM Symp. Component-based Software Engineering, CBSE*, 2014, pp. 3–12.

[37] X. An, E. Rutten, J.-P. Diguet, and A. Gamatié, "Model-based design of correct controllers for dynamically reconfigurable architectures," *ACM Trans. Embed. Comput. Syst.*, vol. 15, no. 3, pp. 51:1–51:27, May 2016.