Logico-numerical Control for Software Components Reconfiguration

Nicolas Berthier, Frederico Alvares, Hervé Marchand, Gwenaël Delaval, Eric Rutten

HAL Id: hal-01644754
https://inria.hal.science/hal-01644754

Logico-numerical Control for Software Components Reconfiguration

Nicolas Berthier⋆ and Frederico Alvares∘† and Hervé Marchand† and Gwenaël Delaval‡ and Éric Rutten‡

Abstract—**We target the problem of the safe control of reconfigurations in component-based software systems, where strategies of adaptation to variations in both their environment and internal resource demands need to be enforced. In this context, the computing system involves software components that are subject to control decisions. We approach this problem under the angle of Discrete Event Systems (DES), involving properties on events observed during the execution (e.g., requests of computing tasks, work overload), and a state space representing different configurations such as activity or assemblies of components. We consider in particular the potential of applying novel logico-numerical control techniques to extend the expressivity of control models and objectives, thereby extending the application of DES in component-based software systems. We elaborate methodological guidelines for the application of logico-numerical control based on a case-study, and validate the result experimentally.**

I. Controlling Components Reconfigurations

A. Software Components and their Reconfigurations

The design of complex computing systems has been improved with the help of structural organization support in the form of software components, encapsulating functionalities that can then be accessed through well-defined interfaces independent from particular implementations. One motivation of software components is that they facilitate the reusability of software subsystems in different contexts, simplified by the abstraction level at which their integration has to be considered. The components structure also makes it possible to design hierarchical components, named composites, that are built by assembling sub-components and defining links between their respective interfaces. The component-based framework Fractal is an example of this approach [1].

Another important motivation for software components concerns the need for software systems to adapt themselves to variations in their lifecycle (e.g., installation, deployment, running, migration, suspension), in their running environments (e.g., variations in the load of requests for an Internet service, or of the data-dependent computational load for a numerical simulation), or in their execution platform (e.g., variations in the available resources like memory, processing units, communication bandwidth). Adapting to these variations can be done at the component level by dynamic reconfigurations of the component assemblies, by adding or removing components, or by modifying the links between them. Such reconfigurations are applied according to what is called an adaptation policy: it has to be defined by the

system designer, and can integrate complex features such as ensuring at the same time resources constraints (e.g., w.r.t. energy) and quality of service assurances. The control of dynamic reconfigurations can be automated in a feedback loop, e.g., following the approach of Autonomic Computing that emerged in distributed computing systems [2].

B. Logico-numerical Control of Components Reconfigurations

We target the problem of the safe control of reconfigurations in component-based software, in order to enforce strategies of adaptation of the system to variations in its environment or internal resources. We approach this problem under the angle of *Discrete Event Systems* (DES), involving properties on events observed during the execution, like requests of computing tasks or work overload, and a state space representing different configurations such as activity or assemblies of components. We build upon previous work on this topic where we explored the integration of reactive languages and tools for the design of controllers, including an application of *Discrete Controller Synthesis* (DCS) in a particular tool-supported approach using BZR [3, 4]. We have recently defined and implemented Ctrl-F, a high-level language for the specification of reconfiguration behaviors in component-based systems [5]. Its compilation integrates the use of BZR, whose support for control was limited to Boolean and static cost functions until recently [6].

In this paper, we consider the potential of applying novel logico-numerical control techniques to extend the expressivity of control models and objectives, and thereby extend the application of DES in component-based software systems. The new synthesis algorithms for logico-numerical control, developed by Berthier and Marchand [6, 7], involve static analysis techniques to handle values' domains like Integers and Reals. It is implemented in the tool ReaX, showing great improvements in both efficiency and expressivity of the systems that can be handled. A new version of the BZR compiler integrates this upgrade.

However, using BZR and ReaX's capabilities for logico-numerical control requires choosing a synthesis algorithm and tuning various parameters that all have a great impact on both the performance of ReaX and the behavior of the resulting controlled system. In order for the design method to be applicable for the design of actual concrete systems, there is thus a need for methodological guidelines for the construction of logico-numerical models suitable for synthesis, as well as for the choice of an adequate synthesis algorithm and the tuning of its parameters.

⋆ Department of Computer Science, University of Liverpool, UK
∘ IMT Atlantique & LS2N, France
† INRIA Rennes - Bretagne Atlantique, France
‡ Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, F-38000 Grenoble France

C. Contributions

The novel contributions we expose in this paper are threefold: First, we explore the potential of logico-numerical control techniques (presented in Section II) in component-based software (Section III), to extend the expressivity of control models and objectives, and thereby extend the application of DES in this domain. Second, we exemplify our approach using a software components reconfiguration control case study in Section IV. Last, in Section V, we elaborate methodological guidelines for the successful application of logico-numerical DCS based on the case study by illustrating the steps required to obtain a suitable controller, and validate the result experimentally. We give related works and conclude in Sections VI and VII.

II. LOGICO-NUMERICAL CONTROL

A. Principles of DCS

The control theory of Discrete Events Systems (DES — Ramadge and Wonham [8], Cassandras and Lafortune [9]) allows the use of constructive methods ensuring, *a priori* and by means of control, required properties on a system's behavior. Usually, the starting point of these theories is: given a model for the system and control objectives, a *controller* must be derived by various means such that the resulting behavior of the closed-loop system meets the control objectives.

A typical example is the *safety control problem* for symbolic systems (i.e., described using state and input variables with associated dynamics), where the desired objective is the enforcement of some *invariant a priori* not satisfied by the system: a controller is to be computed that restricts the admissible values for a subset of the input variables (referred to as *controllable variables*) so that the resulting *controlled system* satisfies the invariant. Finding DCS algorithms computing such controllers in the case of finite-state symbolic systems (i.e., where the state and input variables are Booleans only) has been the objective of several studies, e.g., by Marchand et al. [10].

B. Performing Logico-numerical DCS with *ReaX*

Berthier and Marchand [6] extended the above problem to *logico-numerical systems* where the domain of state and input variables can also be Integers or Reals: i.e., the system can no longer be described as a finite-state automaton. We suggested a symbolic algorithm for solving the problem, accompanied by an implementation in the *ReaX* tool; this algorithm relies on abstract interpretation techniques [11] to compute an over-approximation of a fixpoint in a finite number of iterations. The over-approximation implies that the computed controllers are not maximally permissive in general.

Berthier and Marchand [7] further advanced a variant of the previous algorithm to obtain controllers avoiding *deadlocks* (situations where no solution exists for choosing the controllable variables) in the controlled system for a particular class of systems where previous solutions couldn't avoid them.

```
node tasks(nb_req,ended:int) = (waiting,active:int)
    contract
    assume (nb_req >= 0)
        & (0 <= ended) & (ended <= active)
    enforce active <= 10
    with (act: bool)
var activated: int;
let
  waiting = (0 -> pre waiting) + nb_req - activated;
  activated = 0 -> if (pre waiting > 0) & act
                 then 1 else 0;
  active = (0 -> pre active) + activated - ended;
tel
```

Listing 1.   Heptagon/BZR example.

The algorithms mentioned above are parametric in the sense that one has to select values for a number of parameters in order to use them: two of the most influential ones are: (i) the choice between *boxes* or *convex polyhedra* for the representation of linear numerical constraints; and (ii) the number of iterations before forcing convergence by using a *widening operation*. We give more details about their impact on the efficiency of the synthesis algorithms and the controller produced in Section V-C.1.

C. DCS Through *Heptagon/BZR*

Heptagon/BZR [3, 4] (BZR in the following) is a reactive data-flow language where programs are built as parallel and hierarchical compositions of data-flow *nodes*, each having *input* and *output flows*. The body of a node describes how input flows are transformed into output flows, in the form of a set of *equations* and/or *automata*. These equations define the values of outputs (and possible local variables), using the current values of inputs, and the current *state* of the node: this state can be either memorized values (expressed by *previous* values of variables), or state of automata which can itself embed equations defining variables. New values for input flows are given at each *execution step*, where transitions and equations are then evaluated all together, and values of output flows are updated accordingly.

1) Contracts: An *invariant* and *controllable variables* can be specified using *contracts*. When it encounters a node featuring a contract, the BZR compiler involves DCS to automatically produce a *controller* constraining the values of the controllable variables so as to guarantee that the resulting *controlled node* satisfies the invariant. Contracts also allow to specify some *assumptions* on inputs, e.g., that the value of some input is always positive, or less than some given constant.

2) Example: We show in Listing 1 an example of BZR node. This node tasks models the control of an unbounded set of tasks. The input variable nb_req describes a flow holding the number of new requests of tasks at each instant. These new tasks are initially put in a waiting state: the waiting output flow holds at each instant the number of waiting tasks.

In the syntax of Heptagon/BZR, **pre x** denotes the value

of x at the previous instant; the "`->`" operator allows the initialization of memorized flows, as `x -> y` is the value of `x` at the first instant, and the value of `y` on subsequent ones. `0 -> pre x` can then be read as "the previous value of `x`, initialized as 0 on the first instant". As it must be kept in memory from one instant to the next, this value is part of the state of the underlying system; i.e., it is logico-numerical if `x` is of type `int` or `float` for instance.

In the `tasks` node, the controller can choose, using the *controllable variable* `act`, to activate or not one waiting task (if there is one). The `activated` local variable contains at each instant the number of newly activated tasks (0 or 1). Then, active tasks can be ended by the environment: the input `ended` contains the number of ended tasks.

Now let's suppose we want to impose an upper-bound on the number of active tasks: the contract associated to this node contains an "**enforce**" part, defining the property to be enforced on the system by the generated controller. The "**assume**" part contains assumptions on the environment: here, we assume that at each instant, the number of requests is positive, and the number of ended tasks is lesser than or equal to the current number of active tasks. We detail in Section V-A how these assumptions are used to help synthesizing controllers.
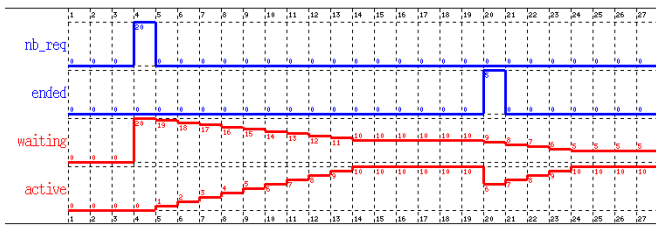

Fig. 1.   Simulation of the node `tasks`.

We show in Figure 1 the simulation of the node `tasks` with manually selected values for its inputs. At the 4th instant, the system receives 20 requests. These requests are put in waiting state, and the generated controller subsequently activates 10 of these requests one by one. Next, the controller blocks the activation of the remaining tasks, until 5 of the active tasks are ended (20th instant). Then it activates 5 new tasks.

Heptagon/BZR uses the `ReaX` DCS tool as back-end to synthesize controllers. The `ReaX` tool and the Heptagon/BZR compiler are designed and integrated so as to be able to produce executable code, co-executing the initial Heptagon/BZR program with the generated controller [6].

III. Managing Software Component Reconfigurations using Ctrl-F

A. Software Components

In Component-Based Software Development [12], a *component* is the most elementary unit of processing or data and it is usually made of two parts: the implementation and the *interface*. The implementation describes the internal behavior of the actual component, whereas the interfaces define how the component should interact with the environment. A component can be simple or *composite* (i.e.,

composed of other sub-components). Components may also feature a set of *attributes* that can be accessed by their *super-component(s)*; some of them may also be modified dynamically. A *connector* (or binding) connects two or more components' interfaces, and corresponds to interactions among components. A *configuration* is a directed graph of components and connectors describing the application's structure and/or a description of how the interactions among components or architectural elements evolve over time.

The transition from one configuration to another is achieved by adding or removing architectural elements (e.g., components, bindings), or changing modifiable attribute values. These reconfiguration actions can be performed dynamically, i.e., at runtime, by hard-programming them with introspection and intercession mechanisms [1, 13]. Reconfiguration actions can also be implicitly defined by tackling adaption at the configuration level: behavioral programs specify the order and conditions under which configurations may take place, and low-level reconfiguration actions are automatically derived from the source and target configurations [14].

B. Ctrl-F Language & Tool-chain

In previous work, we have designed the high-level language Ctrl-F, dedicated to the specification of reconfiguration behaviors in component-based systems [5]. We have also implemented a compilation tool-chain for this language.

1) Ctrl-F Specification Language: A specification written using Ctrl-F involves *behaviors* and *policies*.

Behaviors in Ctrl-F are described in an imperative way by composing either *configurations* or *sub-behaviors* using both *deterministic* (conditional, loop, sequence, or parallel) and *non-deterministic* (alternative) statements. Discrete configurations fully describe the connections between all connectors of the components—architectural elements—involved, and the values of their modifiable attributes. (For the sake of clarity, we represent each configuration with their name, and do not give their complete descriptions.) Conditions (in conditional and loop constructs) are events occurring in the system (e.g., variation of some input measurement). *Alternative* statements under-determine behaviors and leave the choice of configurations unspecified.

In turn, policies in Ctrl-F describe subsets of all possible (global) configurations that the managed components should be in. Policies are expressed using high-level constructs that enable the definition of: (i) *temporal* constraints (in the sense of the logical order of configurations); and (ii) *logical* constraints (relations on component's attributes).

Policies are automatically achieved by selecting reconfiguration choices offered by alternative statements. Indeed, a distinguishing feature of Ctrl-F is that its compilation involves DCS (through a translation in BZR—*cf.* Section II-C) for the automated enforcement of policies [5].

2) Automated Code Generation: From a broader perspective, our method shall provide automatic executable code generation from a component-based architecture definition
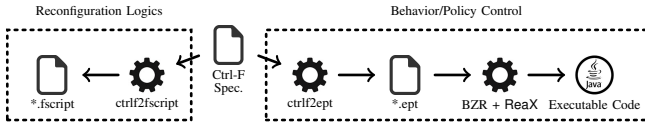
Fig. 2. Executing Ctrl-F Specifications: Compilation Tool-chain.

in Ctrl-F, to the correct-by-construction code in a general-purpose executable language, that is essential to have an impact in the domain of self-adaptive software systems.

The compilation process, illustrated in Figure 2, can be split into two parts. The *reconfiguration logics* part (on the left) takes as input a Ctrl-F definition and generates a script containing a set of *reconfiguration procedures* to be executed to drive the set of controlled component instances from one configuration to another. In turn, the *behavior/policy control* part (on the right) automatically translates a Ctrl-F description into a BZR program. This program encodes all components' behaviors along with a contract ensuring that the policies are invariants. A controller is then derived using a DCS tool such as ReaX, ensuring that the contract is met by taking appropriate decisions at runtime when a choice among several alternative behaviors is required; in other words, all choice is made so that the policies are always satisfied. When compiled using BZR, this program typically produces an object featuring a step() method that behaves according to the behavior/policy control part of the Ctrl-F specification. The step() needs to be fed with inputs encoding events and measurements, and produces outputs identifying reconfiguration procedures to be executed.

3) Integration with a Concrete Component Middleware: Our approach enables the code resulting from the Ctrl-F compilation chain to be integrated with a concrete component platform. We rely on FraSCAti [15], a Java-based middleware that features runtime reconfiguration to the standard Service Component Architecture[1]. While Ctrl-F provides high-level language support for logico-numerical control of self-adaptive components, FraSCAti's role is to bridge application and system levels, in a distributed way.

Hence, not only the business logics of Ctrl-F components are encapsulated within FraSCAti, but also the control loop that manages them. This means that the control loop logics is wrapped into a FraSCAti component that is capable of: (i) handling events; (ii) executing the step() with the events previously identified as input; and (iii) interpreting the outputs of the step() method and executing the corresponding reconfiguration procedures. In turn, the latter correspond to scripts containing commands that are actually executed by the FraSCAti middleware so the reconfiguration can actually take place at runtime in the controlled system.

IV. SOFTWARE COMPONENTS CONTROL USE-CASE

In order to illustrate the software component control problem, let us consider a scalable web application modeled using a component *WebApp*, graphically depicted in
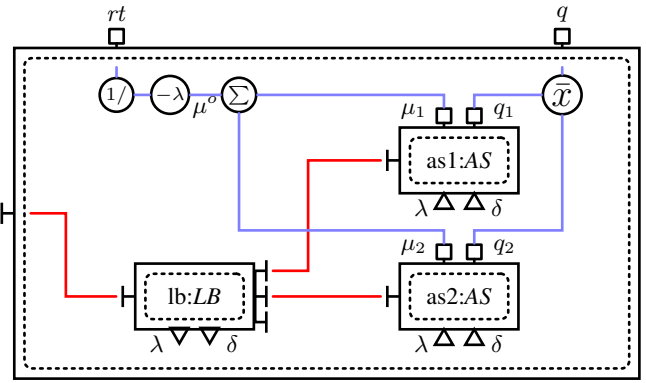
[1] http://www.oasis-opencsa.org/sca

Fig. 3. Graphical Architecture Description of the *WebApp* Component.

Figure 3. *WebApp* is a super-component that encompasses one *load balancer* (*LB*) component, bound to n instances of the *application server* (*AS*) component (two are shown in Figure 3). In order to cope with varying demands in terms of workload, every instance of *AS* can be individually and dynamically activated.

The current workload $\lambda \in \mathbb{R}$ given in requests per second, as well as an event δ that occurs every time an input measurement varies, are internally measured and computed and emitted by the instance "lb" of *LB* component.

Each instance "as$i$" of *AS* component features two attributes: its Quality of Experience (QoE) $q_i \in \mathbb{R}$, and its capacity to absorb incoming workload $\mu_i \in \mathbb{R}$ given in terms of service rate (λ). Several discrete configurations are available for *AS* component instances (not represented in Figure 3): idle when the associated application server is disabled; nominal when it operates in nominal mode; and degraded when it operates in degraded mode. Each one of these configurations is associated with a set of values for μ_i and q_i: e.g., idle: $\mu_i = 0$ and $q_i = 0$; degraded: $\mu_i = 20$ and $q_i = 1$; nominal: $\mu_i = 10$ and $q_i = 2$.

Just as *AS*, *WebApp* comprises two attributes: the overall QoE $q \in \mathbb{R}$ and the response time $rt \in \mathbb{R}$. The former is defined as the average of the QoE of all active *AS* instances, and the latter as $rt = \frac{1}{\mu^o - \lambda}$ (for $0 \leqslant \lambda < \mu^o$), where $\mu^o \in \mathbb{R}$ denotes the total application capacity in terms of service rate, computed as the sum of all active sub-components' capacity; i.e., $\mu^o = \sum_{i=1}^{n} \mu_i$, where μ_i corresponds to the attribute μ of *AS* instance "as$i$". In plain words, the response time rt increases as the number of requests per second grows and gets closer to the capacity. Available discrete configurations for *WebApp* are conf$_0$, conf$_1$, and conf$_2$, where conf$_i$ denotes that i instances of *AS* among its sub-components are not in their idle configuration. Thereby, the set of all global discrete configurations of *WebApp* comprises the Cartesian product of the configurations of every one of its application servers "as$i$".

1) Example Alternative Statements: The behavior specifications in Ctrl-F for *WebApp* and *AS*'s instances may involve *alternative* statements. *AS*'s behaviors specification

for instance, can involve the statement

$$\texttt{idle} \mid \texttt{degraded} \mid \texttt{nominal}, \tag{1}$$

expressing that, at runtime, exactly one of the available discrete configurations should take place at any time. Similarly for *WebApp*, the following alternative is used to under-specify the choice of configuration:

$$\texttt{conf}_0 \mid \texttt{conf}_1 \mid \texttt{conf}_2. \tag{2}$$

2) Example Policy: Attributes of *WebApp* can be used to specify adaptation policies that always have to be met when the modeled application executes. For instance, a policy imposing bounds on response time and average quality of experience can be expressed in Ctrl-F as

$$rt \leqslant 0.3 \wedge q \geqslant 1.5. \tag{3}$$

V. MODELING AND CONTROL USING LOGICO-NUMERICAL DCS

As stated in Section III-B, one important step of the methodology we put forward involves finding a solution to a DCS problem. As Ctrl-F's behavior or policy specifications may involve quantitative aspects modeled using numerical attributes or input measurements, we need to solve a DCS problem on a logico-numerical system. This is for instance the case for the policies involved in our example.

We now explain the method we employed for tuning the model and parameters of the logico-numerical synthesis algorithm implemented in ReaX to obtain a controller. We also present a helper tool ctrl2lut that we developed to assist designers in this process, by allowing them to perform extensive simulations of the controlled systems. We derive general guidelines to help the users of Ctrl-F, as well as any other potential user of ReaX, to successfully perform logico-numerical DCS.

A. Assuming a Realistic Input Space

First of all, one can observe that it is always possible to violate the desired policies (expressed in predicate (3)) by setting a sufficiently high input workload for which there does not exist a configuration of the components providing enough computation capacity to keep rt bellow its assigned threshold.

As a result, in order for the synthesis algorithm to compute a correct controller, one needs to explicitly assume that its inputs actually reflect a realistic run-time situation.

In practice, these restraints on inputs are to be placed within the "**assume**" section of the contracts of BZR nodes, as illustrated in Listing 1 for the inputs nb_req and ended. In this latter example, one can see for instance that the lower-bound 0 <= ended is both sensible w.r.t. the semantics of the model, and required for successful synthesis as any value for ended is uncontrollably subtracted from active.

In the case of our *WebApp* example, one property one may want to assume on the workload is that it never varies "too much" between two successive instants where control decisions are taken. This boils down to limit the difference between the current input value of λ and its value at the previous instant. The discrete controller synthesis is then now performed by assuming that the predicate

$$|\texttt{pre } \lambda - \lambda| < \delta_\lambda \tag{4}$$

always holds for some constant $\delta_\lambda \in \mathbb{R}^+$. Additional basic assumptions can be made on the measured workload that lead to further restricting its domain of values considered realistic using the predicate

$$0 \leqslant \lambda \leqslant \lambda_{\max}. \tag{5}$$

Of course, the choice of suitable values for the constants δ_λ and $\lambda_{\max} \in \mathbb{R}^+$ is application-dependent.

Helped by the above assumptions on the input of the controlled system, ReaX is able to restrict the set of transitions considered during the synthesis to those always satisfying predicates (4) and (5). In effect, this restriction of the (still infinite) input space allows the tool to avoid considering unrealistic behaviors (i.e., transitions due to unrealistic inputs), and leads to successful synthesis of controllers ensuring the desired policies.

B. Performing Closed-loop Simulations with *ctrl2lut*

After a first step leading to the successful synthesis of a controller, one needs to assess that it does not prevent the controlled system to evolve as expected. Indeed, sometimes the only successful way of preventing the reachability of undesired configurations through DCS is simply to prevent the system from evolving at all; of course as no tractable algorithm exists for enforcing some level of progress of controlled logico-numerical systems, one can at least assess it by means of extensive tests.

One basic way of performing extensive simulations that we suggest is to actually execute the resulting controlled system by feeding it with arbitrary and randomly generated inputs and then manually check that it behaves as expected. Further, in the case of a model with associated assumptions (e.g., predicates (4) and (5)), one also needs to ensure that the latter are satisfied by the inputs used for feeding the controlled system.

The ctrl2lut tool we implemented for this purpose comes in handy in this particular case: it indeed allows one to automatically generate two *stochastic reactive programs* suitable for performing extensive co-simulations using Lutin [16]. In our case one of the program encodes the controlled system itself, and the other some environment that is used to generate randomized inputs:

The *controlled system* is the direct encoding of the system along with its associated controller computed by ReaX. Two flavors are available for the generation of this program: either (i) the implicit semantics of BZR is enforced, in which case the choice of values for the controllable variables by the controller is deterministic (they are ordered, and then assigned to true whenever possiblethis is what happened for the variable act in the simulation of tasks shown in Figure 1); or (ii) the controller non-deterministically draws a value for every controllable variable among all admissible

solutions at any step. In the former case the controlled system is deterministic, yet some behaviors that could still have been allowed by control may become unreachable due to the determinisation of the controller; in the latter case such behaviors are still reachable.

In turn, the stochastic program representing the *environment* is fed with the outputs of the controlled system, and outputs the inputs of the latter to form a closed loop. This environment ensures that the generated inputs satisfy *at least* the assertion specified when modeling the system to control; e.g., predicates (4) and (5). This program can further be refined to make the simulated workload λ (input of the controlled system, hence output of the simulated stochastic environment) globally follow a periodic behavior with increasing and decreasing phases. We refer the reader to the work of Jahier et al. [16] for details on how this can be done using Lutin.

C. Guidelines for Using Logico-numerical Control

1) Tuning Synthesis Algorithm Parameters: As mentioned in Section II-B, several parameters are available for tuning the efficiency of the synthesis algorithm for logico-numerical systems, both in terms of execution time (and maximum memory occupancy) as well as precision of the result. For illustrative purposes, we have selected some of them that we think are the most relevant for the success of the modeling approach we put forward:

Using boxes instead of convex polyhedra: although more computationally expensive, the latter induce less approximations if the model and invariant feature numerical expressions involving more than one non-constant input or state variable. For instance, predicates (3) and (5) do not require the power of convex polyhedra *per se*, yet predicate (4) does;

Widening delay: the synthesis algorithm uses widening operations to force convergence of its otherwise potentially non-terminating iterative computations. As these operations are often responsible for most of the loss of precision during the computations, delaying them may either: (i) lead to more permissive controllers; (ii) make an otherwise failing synthesis succeed;

Using the deadlock-free algorithm: essentially, this variant of the algorithm operates on more intricate data-structures to avoid deadlocks; one should note however that the original model we obtain for our running example does not induce deadlocks. Yet, this variant of the algorithm is also much more precise, and able to represent and manipulate invariants whose negation cannot be expressed using a conjunction of numerical constraints (like the invariant $0 \leqslant x \leqslant 10$ for instance).

2) Assessing Parameters' Impacts: We generated multiple controllers with varying choices for the parameters exposed above, and then assessed the behaviors of each resulting controlled systems based on manual analyses of the execution traces obtained using closed-loop simulations carried out with Lutin, with the help of ctrl2lut.

First of all, the cheaper, non deadlock-free variant of the algorithm always succeeded in producing controllers in less than a second, whatever the widening delay and the choice of boxes or convex polyhedra[2]. However, all of the resulting controllers were very conservative in the sense that they forced the system to always be in the configuration having the highest capacity whatever the input workload. Using boxes with the deadlock-free variant led to similar synthesis timesand conservative controllers.

In the end, the deadlock-free variant with convex polyhedra (the most precise, yet also the most computationally expensive) took about 10 seconds to compute controllers. As in the case of the non deadlock-free variant, no widening is actually needed for the algorithm to terminate.

Regarding the effect of widening delays in the deadlock-free variant, one significant observation is that shorter delays (with widenings starting before 3 iterations) led to slightly more conservative controllers than in the case of controllers computed with greater widening delays: the threshold workload levels above which configurations with higher computation capacities were forced were lower for the former (controllers computed with shorter widening delays) than for the latter, more permissive controllers.

Further, the widening operation used by the deadlock-free variant is quite expensive, so in effect delaying it is both (potentially) more precise, and more likely to give results faster if the transition function is quite simple (i.e., if numerical expressions remain quite simple, such as involving at most two variables, like in our example).

3) Reals vs Integers: From our experience, with the kind of models obtained in the case study, using numerical variables defined on the domains of either Reals or Integers do not have a significant impact on the performance of the synthesis tool. Regarding stochastic simulations of the resulting system however, one should note that using Integers instead of Reals may induce additional costs due to the drawing of Integers in non-empty convex polyhedra that may not comprise integral values.

D. Execution of the Example

1) Concrete Application and Infrastructure: We apply our scalable web application model presented in Section IV to control RUBiS/Brownout [17], an extended version of the well known eBay-like auction site RUBiS [18], which is widely used for benchmarking. The extended version consists in allowing recommendations to the users when they navigate through the site, which is a very common functionality provided by real e-commerce sites. The standard or nominal state of the 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 model this behavior, and equivalently the one of *WebApp*, using the alternatives exemplified in statements (1) and (2). Naturally, the nominal mode has a smaller capacity and higher QoE, whereas the degraded one is capable of accommodating more requests at the expense of a lower QoE; we thus use the

---

[2]The absence of performance impact of the widening delay suggests that no widening is actually required for the algorithm to terminate in this case.
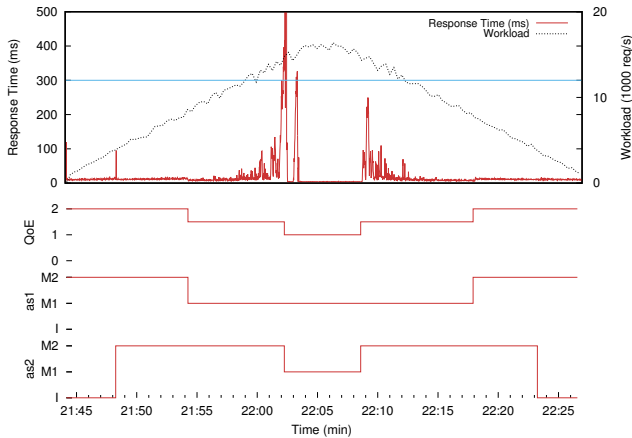
Fig. 4. Trace of the Resulting Executable Code under a Varying Workload.

values given as examples for μ_i and q_i in Section IV. For this scenario, the desired policy that is enforced using DCS states that the response time attribute of the component *WebApp* never exceeds 300 ms: the associated invariant is then $rt \leqslant 0.3$.

We deploy the RUBiS/Brownout application in a set of servers of Grid'5000[3], which is a French national-wide grid of clusters that are used as infrastructure testbed. Our setup consists of three physical machines: one used to simulate the client requests, another to host the virtual machines containing the Brownout/RUBiS components, and another for management purposes, including the code resulting from the Ctrl-F compilation encapsulated by the FraSCAti components. Each machine is equipped with 12 Intel Xeon E5-2630 CPU cores and 32 GB of RAM. In total, we created three virtual machines: one containing one instance of the *LB* component, and the other two accommodate each one instance of component *AS* that can be started or stopped according to the control commands. Each virtual machine has the same configuration: 4 CPU cores and 8 GB of RAM.

In order to simulate the client activity, we created a 45-minutes synthetic workload consisting of an increase followed by a decrease of load. The workload is injected to the target application thanks to the tool Gatling[4].

2) Results: Figure 4 shows the behavior of the the Ctrl-F-enabled version of Brownout/RUBiS application, subject to the workload (dotted line) mentioned above. The plots illustrate the service response time, the QoE, and the configuration changes for each instance of *AS* (I, $M1$ and $M2$ corresponding respectively to idle, degraded and nominal).

The application starts with one instance of *AS* and does not change its configuration as long as the workload remains low. As the workload increases, the controller first forces the application to start the second instance so that the response time invariant can be kept. Then, as soon as the application can no longer be scaled out (we are limited to two instances of *AS*), the controller forces the degradation of the QoE

[3]http://www.grid5000.fr
[4]http://www.gatling.io

of the *AS* instances, one at a time, so as to be able to handle the peak of load and keep the response time under the required threshold. On the other way around, as the workload decreases, the application first makes sure that the QoE is fully recovered before stopping the second *AS* instance, and gets back to the initial state.

VI. RELATED WORKS

In the area of DES modeling and synthesis techniques, applications to the control of computing systems is still quite recent, and has been explored for example on problems like the deadlock avoidance in multi-thread programs running on multi-core processor architectures [19, 20]. Other works, e.g., by Hajjar et al. [21], target especially the control of software components where verification and controller synthesis are combined for the construction of assemblies of reusable Commercial off-the-shelf (COTS) components. While in prior work [22, 5] we provided DCS-based safe control of reconfigurations in component-based software systems, it was limited to Boolean and static cost functions aspects: in this paper we extend the component control models and objectives with logico-numerical aspects.

Other related works in the area of Software Engineering feature the proposal of Kouchnarenko and Weber [23] to use temporal logics in order to integrate temporal requirements to adaptation policies in the context of Fractal components [1]. Whereas in their approach adaptation strategies are enforced and/or reflected ("reified") at runtime, in our approach the controller is obtained through DCS, i.e., it is computed and compiled off-line, producing as result a correct-by-construction controller enforcing the desired adaptive behaviors; off-line detection of desired adaptation behaviors that are unattainable is also possible using our approach. D'Ippolito et al. [24] propose to define multiple models and controllers associated with different levels of assumption (from the least to the most restrictive) and guaranteeable functionalities. The level of control is then determined according to the validity of assumptions at runtime. They use discrete control in the sense that some layers consider more logical constraints and objectives, relying on techniques stemming from AI planning.

VII. CONCLUSION

In this paper, we have demonstrated the potential of logico-numerical control techniques for extending the expressivity of control models and adaptation policies for component-based software systems. We have exemplified and validated the use of these new techniques by designing a reconfiguration controller for an example web application use case. We have also elaborated on the various choices offered by the tool ReaX for tuning the logico-numerical DCS algorithms, and have given some methodological guidelines for the application of such control.

The intrinsic combinatorial complexity of DCS involved in the compilation of the Ctrl-F language raises an issue regarding the scalability of the approach. One answer to be

explored is to decompose control problems specified in Ctrl-F in a systematical way by relying on the mechanisms of modular compilation and modular DCS available in BZR. Yet, using these mechanisms is also non-trivial as it requires the specification of assumptions and guarantees for each node, in such a way that they both convey enough information to the upper control layer, and stay concise and abstract in order to maintain a gain in complexity. Our perspective is to augment Ctrl-F with a hierarchical structure allowing for the automated generation of such hierarchical constructs.

As shown in this paper, software component-based systems naturally involve reconfigurations between different structures of components assemblies. Another important aspect is that the adaptation policies themselves may become dynamic, for instance to react to changes in operation conditions. Also, the set of components in the system may also change, by appearing or disappearing in a context of mobile computing. Therefore an important perspective to work on regards the notion of adaptive control that would be defined for DES, as well as developed and integrated in the ReaX and BZR framework.

REFERENCES

[1] 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.

[2] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003.

[3] 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*, ser. EMSOFT'11, Taipei, Taiwan, Oct. 2011, pp. 309–318.

[4] G. Delaval, E. 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, Dec. 2013.

[5] F. Alvares, E. Rutten, and L. Seinturier, "A domain-specific language for the control of self-adaptive component-based architecture," *Journal of Systems and Software*, pp. –, 2017.

[6] N. Berthier and H. Marchand, "Discrete Controller Synthesis for Infinite State Systems with ReaX," in *IEEE International Workshop on Discrete Event Systems*, Cachan, France, 2014, pp. 420–427.

[7] ——, "Deadlock-Free Discrete Controller Synthesis for Infinite State Systems," in *54th IEEE Conference on Decision and Control*, ser. CDC '15, Dec. 2015.

[8] P. Ramadge and W. Wonham, "The control of discrete event systems," *Proceedings of the IEEE; Special issue on Dynamics of Discrete Event Systems*, vol. 77, no. 1, pp. 81–98, 1989.

[9] C. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*. Springer, 2007.

[10] 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*, vol. 10, no. 4, pp. 325–346, Oct. 2000.

[11] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '77. New York, NY, USA: ACM, 1977, pp. 238–252.

[12] 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. Cambridge University Press, 2000, pp. 47–68.

[13] 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.

[14] 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.

[15] 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.

[16] E. Jahier, N. Halbwachs, and P. Raymond, "Engineering functional requirements of reactive systems using synchronous languages," in *8th IEEE International Symposium on Industrial Embedded Systems*, ser. SIES. IEEE, 2013, pp. 140–149.

[17] C. Klein, M. Maggio, K.-E. Årzén, and F. Hernández-Rodriguez, "Brownout: Building more robust cloud applications," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 700–711.

[18] "RUBiS: Rice University Bidding System," http://rubis.ow2.org, 2008, accessed: 2017-02-28.

[19] Y. Wang, H. Cho, H. Liao, A. Nazeem, T. Kelly, S. Lafortune, S. Mahlke, and S. A. Reveliotis, "Supervisory control of software execution for failure avoidance: Experience from the gadara project," in *Proc. of the 10th IFAC Int. Workshop on Discrete Event Systems (WODES'10)*, Sept. 2010.

[20] Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, and S. Mahlke, "The theory of deadlock avoidance via discrete control," in *Principles of Programming Languages, POPL*, Savannah, USA, 2009, pp. 252–263.

[21] S. Hajjar, E. Dumitrescu, L. Pietrac, and E. Niel, "A design method for synthesizing control-command systems out of reusable components," in *IEEE International Workshop on Discrete Event Systems*, Cachan, France, 2014.

[22] 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, Jun. 2010.

[23] 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*, ser. LNCS, vol. 8348, Nanchang, China, 2014, pp. 234–253.

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