Available online at www.sciencedirect.com

ScienceDirect

IFAC Papers Online CONFERENCE PAPER ARCHIVE

IFAC-PapersOnLine 48-7 (2015) 139–146

Distributed Execution of Modular Discrete Controllers for Data Center Management ⋆

Gwenaël Delaval * **Soguy Mak-Karé Gueye** * **Éric Rutten** **

* *LIG/Université Grenoble Alpes, Grenoble, France, (e-mail: {gwenael.delaval, soguy-mak-kare.gueye}@inria.fr)*
** *INRIA, Grenoble, France, (e-mail: eric.rutten@inria.fr)*

Abstract:
Complex computing systems are increasingly designed so that they are self-adaptive, and adopt the autonomic computing approach for their administration. Real systems require the co-existence of multiple such autonomic management loops. Their uncoordinated execution can lead to problematic interferences and jeopardize performance as well as consistency. This is a typical example of the general need for methodological support for the design of well-coordinated managers, without breaking their natural modularity. We address the problem with a method stressing modularity, and focusing on the discrete control of the interactions of managers. This paper focusses on proposals for the distributed execution of modular controllers, first in synchronized way, and then relaxing this synchronization. We apply and validate our method on a multiple-loop multi-tier system in a data-center.

Keywords: Control of computing systems, Discrete Control, Modularity, Distribution

1. INTRODUCTION

Computing systems are more and more required to be able to adapt their execution and behavior dynamically, in reaction to changes in their workload, environment, or application modes. The control of such reconfigurations in the computing infrastructures is an active research domain, with strong impact on application domains such as embedded systems, Cloud computing or smart environments. Typically, in data-center platforms that support Cloud Computing and web servers, as we consider in the *Ctrl-Green* project [1], large distributed systems, involving numerous networked computers require to be controlled dynamically, in order to answer to varying workloads while minimizing costs and maintaining Quality of Service, with managers for resources, dependability, and energetic efficiency. Such dynamical management is done in control loops and called Autonomic Computing [Kephart and Chess (2003)].

Recently, the design of such Autonomic Computing loops has been approached under the angle of Control Theory [Hellerstein et al. (2004)]; mostly, classical control techniques are used, to handle quantitative aspects. Other aspects, more related to logical problems, can be approached and solved as Control problems of Discrete Event Systems (DES), using for example Petri net models [Wang et al. (2010)]. Our work builds upon previous results on using Discrete Controller Synthesis (DCS) methods, based upon the programming language and tool Heptagon/BZR, applied to data center administration, especially for the coordination of multiple loops, multi-tier systems [Delaval et al. (2013a)].

The work presented here aims at obtaining distributed discrete controllers in this context, motivated as follows:

- the controlled computing systems have a naturally distributed architecture,
- managing locally monitors informations (sensors) and reconfiguration actions (actuators) can dramatically reduce communication overhead,
- robustness is improved, in that failure of one of the distributed modules involves repairing only this one; failure management can involve reconstructing the current state of the failed controller.

We approach distribution by exploiting the support of hierarchy of nodes and modularity in the Heptagon/BZR language, based on modular DCS and modular code generation. Our approach involves :

(1) describing local controllers on each site, in the form of local behaviors and objectives,
(2) modeling the distributed platform, in a global model, composing the local ones, and a model of the communications between them,
(3) performing modular compilation and DCS on the whole program / model, which work as a verification at the global level,
(4) using the modularly generated executable code for each site.

We apply and validate our method on a multiple-loop multi-tier system in a data-center.

⋆ This research is partly supported by ANR INFRA (ANR-11-INFR 012 11) under a grant for the project Grtl-Green.

[1] http://www.en.ctrlgreen.org/

10.1016/j.ifacol.2015.06.485

2. BACKGROUND

In this section we briefly recall the formal methods and tools upon which we base our approach, and some preliminary work in the application domain. To build the Discrete Event System model, we use the Labelled Transition Systems underlying the reactive languages of the synchronous approach. They have been used as a basis for the definition of a Discrete Controller Synthesis approach [Marchand and Samaan (2000)], adapting the classical framework of [Ramadge and Wonham (1989)] to models obtained from synchronous languages. An advantage of this approach is that it is tool-supported, by compilers like Heptagon/BZR [2] and by the DCS tool Sigali [Marchand et al. (2000)]. This point is essential in order to consider effective applications to concrete computing systems, as we aim in this work.

2.1 Reactive languages

Reactive systems are characterized by their continuous interaction with their environment, reacting to flows of inputs by producing flows of outputs. They are classically modeled as transition systems or automata, with languages like StateCharts [Harel and Naamad (1996)]. We adopt the approach of synchronous languages [Halbwachs (1998)], because we then have access to the control tools used further. The synchronous paradigm refer to the automata parallel composition that we use in these languages, allowing for clear formal semantics, while supporting modelling asynchronous computations [Halbwachs and Baghdadi (2002)]: actions can be asynchronously started, and their completion is waited for, without blocking other activity continuing in parallel. The Heptagon/BZR language [Delaval et al. (2013b)] supports programming of mixed synchronous data-flow equations and automata, called Mode Automata [Maraninchi and Rémond (2003)], with parallel and hierarchical composition. The basic behavior is that at each reaction step, values in the input flows are used, as well as local and memory values, in order to compute the next state and the values of the output flows for that step. Inside the nodes, this is expressed as a set of equations defining, for each output and local, the value of the flow, in terms of an expression on other flows, possibly using local flows and state values from past steps.

Figure 1(a,b) shows a small Heptagon/BZR program. The node **delayable** programs the control of a task, which can either be idle, waiting or active. When it is in the initial Idle state, the occurrence of the **true** value on input **r** *requests* the starting of the task. Another input **c** can either allow the activation, or temporarily block the request and make the automaton go to a waiting state. Input **e** notifies termination. The outputs represent, resp., **a**: activity of the task, and **s**: triggering the concrete task start in the system's API. Such automata and data-flow reactive nodes can be reused by instantiation, and composed in parallel (noted "**;**") and in a hierarchical way, as illustrated in the body of the node in Figure 1(c), with two instances of the **delayable** node. They run in parallel, in a synchronous way: one global step corresponds to one local step for every node.

[2] `http://bzr.inria.fr`

2.2 Discrete Controller Synthesis (DCS)

Among the methods of design and validation, the controller synthesis is one of the most attractive. It helps refine an incomplete specification in order to achieve a certain goal such as the satisfaction of a property not yet checked with the original system. DCS, computes a control logic by construction. It is based on formal methods for the synthesis of a controller enforcing properties on a system to be controlled. It requires a model of the behavior of the system to be controlled and a specification of properties to achieve. The latter are expressed in terms of control objectives, such as invariance. The model system formally described all possible behaviors, the correct and incorrect behavior regarding the desired properties.

The synchronous langage Heptagon/BZR [Delaval et al. (2013b)] integrates DCS in their compilation. This langage allows an easy use of DCS by introducing the notion of contract in a modeling system. the contract is described declaratively and consists of three statements: **assume**, **enforce** and **with**. The contract contains properties that the functioning of system must meet. These properties are declared as control objectives in the **enforce** statement. When the model that describes the dynamics of the system does not meet the properties, Heptagon/BZR generates a control logic that enforces the latter when controllable inputs are defined in the model. The latter inputs are declared as local variables in the **with** statement of the contract. The generated control logic determines the values to assign to the controllable variables in order to restrain the modelled behaviors to satisfy the properties. Relevant properties on the environment are declared in the **assume** statement of the contract. This is taken into account during the synthesis of the control logic.

Figure 1(c) shows an example of contract coordinating two instances of the **delayable** node of Figure 1(a). The **twotasks** node has a **with** part declaring controllable variables c_1 and c_2, and the **enforce** part asserts the property to be enforced by DCS. Here, we want to ensure that the two tasks running in parallel will not be both active at the same time: **not** $(a_1$ **and** $a_2)$. Thus, c_1 and c_2 will be used by the synthesized controller to delay some requests, leading automata of tasks to the waiting state whenever the other task is active.

Heptagon/BZR always generates a maximally permissive solution for a synthesis problem, which is then made deterministic in order to be executable. In absence of controllable variables, the program is fully deterministic and no control can be performed. Then, the synthesis tool only checks, by model-checking, that the property in the **enforce** part is verified by the program.

2.3 Modular Control

Modular DCS consists in taking advantage of the modular structure of the system to control locally some subparts of this system [Marchand and Samaan (2000)]. The benefits of this technique is firstly, to allow computing the controller only once for specific components, independently of the context where this component is used, hence being able to reuse the computed controller in other contexts. Secondly, as DCS itself is performed on a sub-