Discrete Control-Based Design of Adaptive and Autonomic Computing Systems*

Xin An[1], Gwenaël Delaval[2], Jean-Philippe Diguet[3], Abdoulaye Gamatié[4], Soguy Gueye[2], Hervé Marchand[5], Noël de Palma[2], and Eric Rutten[6]

[1] Hefei University of Technology, Hefei, China
xin.an@hfut.edu.cn
[2] LIG, Grenoble, France
{gwenael.delaval,Soguy-Mak-Kare.Gueye,noel.depalma}@imag.fr
[3] Lab-STICC, Lorient, France
jean-philippe.diguet@univ-ubs.fr
[4] LIRMM, Montpellier, France
abdoulaye.gamatie@lirmm.fr
[5] INRIA, Rennes, France
herve.marchand@inria.fr
[6] INRIA, Grenoble, France
eric.rutten@inria.fr
https://team.inria.fr/ctrl-a/members/eric-rutten

Abstract. This invited paper makes an overview of our works addressing discrete control-based design of adaptive and reconfigurable computing systems, also called autonomic computing. They are characterized by their ability to switch between different execution modes w.r.t. application and functionality, mapping and deployment, or execution architecture. The control of such reconfigurations or adaptations is a new application domain for control theory, called feedback computing. We approach the problem with a programming language supported approach, based on synchronous languages and discrete control synthesis. We concretely use this approach in FPGA-based reconfigurable architectures, and in the coordination of administration loops.

Keywords: Autonomic computing, adaptive systems, reconfigurable architectures, reactive systems, synchronous languages, discrete control.

1 Adaptive Computing Systems, and their Control

Computing systems are present in ever more aspects of society, and they have to comply with two complementary, and sometimes contradictory, requirements: adaptability to continuous changes in their environment or functionality, and dependability w.r.t. the goal they fulfill and the persons in their contact.

* This presentation is an overview of work done with support from several projects: Minalogic MIND, ANR Famous, CNRS PEPS API, ANR Ctrl-Green, Labex Persyval-Lab Projet Exploratoire Staars, Inria Action Exploratoire Ctrl-A.

R. Natarajan et al. (Eds.): ICDCIT 2015, LNCS 8956, pp. 93–113, 2015.
© Springer International Publishing Switzerland 2015

1.1 Administration Loops in Computing Systems

Motivations for being dynamically reconfigurable or adaptive are manifold: on the one hand, systems should dynamically react to changes in application objectives, in environment of operation, and also in their implementation platform or infrastructure, especially in open systems like the Cloud. On the other hand, systems are too large or complex to be administrated manually and must be automated, in order to avoid error-prone or slow decisions and manipulations.

This trend can be observed at very diverse levels of services and application software, middleware and virtual machines, operating systems, and hardware reconfigurable architectures. The automation of such dynamical adaptation manages various aspects such as computing and communication resources, quality of service, fault tolerance. It can concern small embedded systems like sensors networks, up to large-scale systems such as data-centers and the Cloud. For example, data-centers infrastructures have administration loops managing their computing resources, typically with energy-aware objectives in mind, and possibly involving management of the cooling system. At a lower level, FPGA-based architectures (Field-Programmable Gate Arrays) are hardware circuits that can be configured at run-time with the logics they should implement: they can be reconfigured dynamically and partially (i.e. on part of the reconfigurable surface) in response to environment or application events; such reconfiguration decisions are taken based on monitoring the system's and its environment's features.

Autonomic computing [21,20] is an approach for the design of systems evolving in a self-managed way while continuing to run and deliver the service. It is based on an engineering of the administration of systems in the form of a feedback loop, automating the decisions and actions to be taken according to observations on the state and events of the system.

1.2 The Need for Control

The other vital requirement for these systems is dependability, be it w.r.t. damage in the finality of the system (information, business, ...) or w.r.t. safety (goods, persons, ...) [6]. The need for guarantees and assurances on the behavior of these automated systems can benefit from generally meaningful and classical formal methods in Computer Science like Model Checking for logical or temporized aspects, or can make use of models from performance evaluation, or concerning probabilistic aspects (e.g. Markov chains).

A specificity of autonomic systems is that they are based on a feedback loop, the behavior of which calls for a corpus of design theories and techniques stemming from Control Theory, where they have been studied for many decades. This control oriented approach to autonomic computing [18] is a new interaction between control and computer science, along with classically established ones:

- computer science for control systems, widely considered in embedded and real-time systems for the digital implementation of control;

– theoretical computer science and control theory, designing hybrid systems as mathematical models to combine discrete and continuous dynamics;
– control theory for computing systems, considered here, for designing well-behaved automated computer management loops.

The autonomic loop is also naturally reactive, hence a new potential domain for reactive languages and models, like synchronous languages [3], different from hard real-time safety critical embedded systems, and bringing different perspectives for their validation and verification tools.

1.3 Approach and Outline

We address the problem of combining adaptivity and dependability, requiring run-time abilities to detect or even predict changes requiring an adaptation, decide upon the appropriate adaptation, with possible anticipation, and give guarantees on this appropriateness as a notion of correction of the control.

We propose an approach which is language-based and tool-supported, and which we validated early on by confronting the method and its supporting language and tools to concrete real-life systems from different domains, in order to insure relevance and generality. This paper makes an overview of the approach, as well as mentioning different facets of the work, that have been developed in more specialized and detailed presentations elsewhere.

In the remainder, first Section 2 recalls basic notions in relevant domains: Autonomic Computing in Section 2.1 ; reactive systems and their control in Section 2.2. Then Section 3 presents the BZR language on which the approach is based. Subsequently, Section 4 shows how the approach is validated in a range of domains : software components and coordination of multiple autonomic loops in Section 4.3 ; reconfigurable FPGA architectures in Section 4.2. Lastly, Section 5 concludes, discusses results and draws perspectives.

2 Background

2.1 Autonomic Computing

The aim of Autonomic Computing is to have networked computing systems able to manage themselves, trough decisions made automatically, without direct human intervention. The Autonomic Computing Initiative (ACI) initiated by IBM aims at providing the foundation for autonomic systems [21]. It is inspired by the autonomic nervous system of the human body. This nervous system controls important bodily functions (e.g. respiration, heart rate, and blood pressure) without any conscious intervention. In the past dozen years Autonomic Computing has gained momentum, both academically and industrially [20].

Autonomic objectives have been defined for self-management aspects, often called self-*, covering essential features:

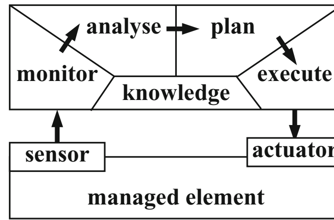


Fig. 1. MAPE-K autonomic manager for administration loop

- Self-configuration: automatic configuration of the system components at deployment time, or also later during runtime, typically without stopping;
- Self-healing: automatic discovery, and correction of faults;
- Self-optimization: automatic monitoring and control of resources to ensure the optimal functioning with respect to the defined requirements;
- Self-protection: identification and protection from arbitrary attacks: this security aspect can be addressed on external or internal aspects [9].

Interestingly, these objectives can interact, and their interferences can require coordination, typically between self-protection and self-optimization.

The autonomic loop is a general feedback loop structure to take this into account [21]. In this closed loop, systems are instrumented with monitors or sensors, and with reconfiguration actions or actuators; these two kinds of interfaces with the managed element (ME) have to be related by a control and decision component, the autonomic manager (AM), which implements the dynamic adaptation policy or strategy. It can be defined as shown in Figure 1 with the MAPE-K approach, with sub-components for:

- Monitoring: extracting relevant information from sensors, probes or monitors instrumenting the managed element, and available at its API;
- Analysis: using the monitored information as well as other knowledge e.g., on past history, to decide on reactions to take;
- Planning: transforming the decisions into actions
- Execution: implementing the action according to the managed element control interfaces or actuators;
- Knowledge: storing and maintaing relevant information of the managed element, used in the other sub-components, and updated by them.

Such autonomic loops can be designed and developed in many different ways, relying on techniques from e.g. Artificial Intelligence, but an important issue remains in providing guarantees on the behavior of such automated closed-looped systems as the are generally difficult to master. A typical example is the so-called "state-flapping" problem [20, p. 7:21], where reconfigurations altern back and forth between two states because transition conditions are too close.

Control for feedback computing is therefore a particularly interesting approach where this feedback loop is considered as a case of a control loop, where techniques stemming from control theory can be used to design efficient, safe, and predictable controllers [18]. Control theory provides designers with a framework of methods and techniques to build automated systems with well-mastered behavior. It involves sensors and actuators that are connected to the process or "plant" i.e., the system to be controlled. A model of the dynamic behavior of the process is built, and a specification is given for the control objective, and on these bases the control is derived, following a formal computation. Although there are approaches to the formal derivation of software from specifications, this methodology is not usual in Computer Science, where often a solution is designed directly, and only then it is analyzed and verified formally, and the distinction between the process and its controller is not made systematically. This approach, sometimes called Feedback Computing [18,29], although well identified, is still only emerging. Works are scattered in very separate and dispersed efforts, in different communities. Some surveys exist [5,8,10], offering a classification [25], or concentrating on Real-Time computing systems [2].

The control approach advantages [29] come from its rigorous methodology for modeling, designing, and analyzing feedback loops. It supports the design of controllers that effectively manage uncertainties in computing systems, without needing accurate models. They bring interesting properties of stability or robustness, which, in the context of computer systems improves predictability. On the other hand, there are of course difficulties and limitations. Making the mapping from high-level management objectives to actual system-level sensors and actuators, and to appropriate control models can be hard. Modeling computing systems does not easily fit classical control methodologies: difference in cultures shows for example in that many classical control problems are formulated as regulation or tracking problems, rather than optimization, and deal only with continuous metrics. On the side of the objects of control, most computing systems were not designed to be controllable in the first place, and it is a real architectural research problem to build and instrument them appropriately.

2.2 Reactive Systems, their Programming, and Discrete Control

The AM shown above is intrinsically a reactive component, therefore some design approaches originally intended for embedded systems and general feedback loops can be of interested for autonomic managers, for which they can be adapted.

Reactive systems and synchronous languages 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 classically famous languages like StateCharts [17]. We adopt the approach of synchronous languages [3], because we then have access to the control tools used further. Well known languages feature the imperative Esterel, the equational declarative Lustre and Signal. The synchronous paradigm refers

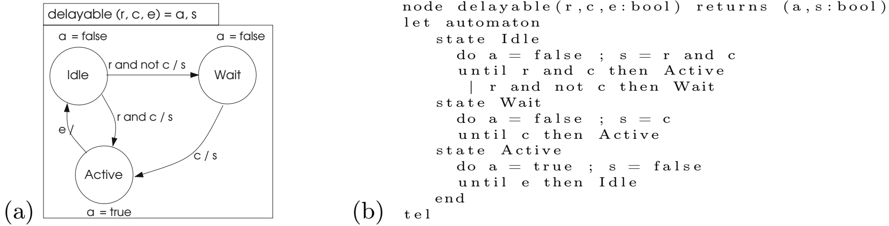


Fig. 2. Heptagon/BZR example: : (a) graphical / (b) textual syntax

to the automata parallel composition that we use in these languages, allowing for clear formal semantics, while supporting modelling asynchronous computations [15]: actions can be asynchronously started, and their completion is waited for, without blocking activity continuing in parallel.

The Heptagon/BZR language [13] supports programming of mixed synchronous data-flow equations and automata, called Mode Automata, 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. This can already be seen as a programmatic solutions for reconfiguring the data flow between in- and outputs, providing for control and coordination of data-flow tasks.

Figure 2 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 3, with two instances of the `delayable` node. Particularly, in the second instance, input c is fed with the constant flow **`true`**: hence, the behavior of this instance is specialized in that the requests are always immediately starting the task. They run in parallel, in a synchronous way: one global step corresponds to one local step for every node. In particular, when r_1, r_2, c_1 are received **`true`** at the same step from the initial state, the resulting state is such that $a_1 \wedge a_2$.

Synchronous languages are tool-supported, and compilers automatically generate executable code, e.g., in C or Java, typically structured as a `reset` function to initialize variables, and a `step` function, implementing the global transition

function. The code calling this function is application-dependent, and can be an infinite loop, a periodical call, or an event-based or interruption mechanism.

Verification and discrete control are the available when using a reactive language, which gives all the support of the the classical formal framework of Labelled Transition Systems (LTS): they involve two main features. On the one hand there is a memorization of a *state*, the current value $x(k)$ resulting from the previous transition at $k-1$ (with an initial value $x(0)$). On the other hand is a *transition function* T computing the next value of the state in function of the current observed input value $y(k)$ and current state. It also computes output values $o(k)$, to send commands to the controlled system:

$$(\mathbf{x}(k+1), \mathbf{o}(k)) = T(\mathbf{y}(k), \mathbf{x}(k)), x(0) = x_0$$

Particularly, we benefit from state-space exploration techniques, like Model-Checking, in order to check whether or not a temporal logic formula is satisfied by all the possible executions of the program.

More originally, the LTS of a program can be applied the operation of Discrete Controller Synthesis (DCS). Initially defined as supervisory control of discrete event systems in the framework of language theory [26], DCS has been adapted to symbolic LTS and implemented in synchronous tools [23]. The LTS variables $\mathbf{y}$ are partitioned into controllable ones $\mathbf{c}$ and uncontrollable ones $\mathbf{uc}$. For a given control objective (e.g., staying invariantly inside a given subset of states, considered "good", or keeping some states reachable), the DCS algorithm automatically computes, by exploration of the state space, the constraint on controllable variables, depending on the current state, for any value of the uncontrollables, so that remaining behaviors satisfy the objective. This constraint is inhibiting the minimum possible behaviors, therefore it is called *maximally permissive*. The resulting synthesized controller C gives values to controllable variables c, which are part of the parameters of the transition function T. In brief:

$$\begin{array}{rcl} \mathbf{x}(k+1) & = & T(\mathbf{uc}(k), \mathbf{c}(k), \mathbf{x}(k)), x(0) = x_0 \\ \mathbf{c}(k) & = & C(\mathbf{uc}(k), \mathbf{x}(k)) \end{array}$$

Algorithms are related to model checking techniques for state space exploration. If no solution is found, because the problem is over constrained, then DCS plays the role of a verification. Discrete control objectives can be logical : ensuring, for a given subset of states characterized by a predicate, its invariance (by control, it will not be left), reachability (from all visitable states), attractivity (no cyclic sequence of transitions can avoid it). They can involve weights associated

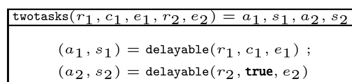


Fig. 3. Heptagon/BZR example: nodes composition

with states for quantitative aspects : bounding capacity, optimizing performance. Multiple criteria optimization can also be supported. In the framework of the synchronous languages and technology, the tool Sigali [23] is integrated in the programming environments, and in the compiler of the BZR language [19].

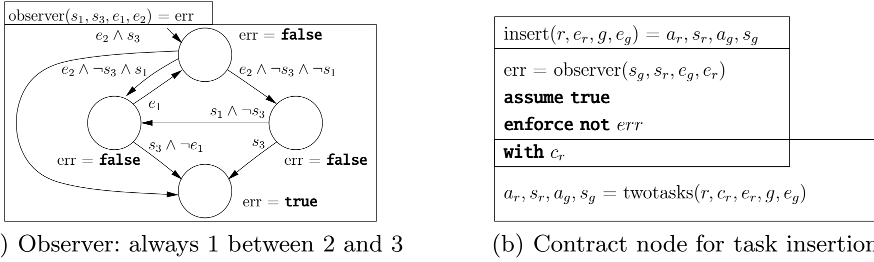
Discrete feedback computing i.e., applying discrete control theory to computing systems, is more recent than using classical control theory [18], as was noted by other authors, essentially because it is less well-known than classical control and less developed than classical verification and model-checking. Earliest works deal with controlling workflow scheduling [27] or application-specific task schedulers [24,22]. A whole line of work focuses on the computing systems problem of deadlock avoidance in shared-memory multi-threaded programs [28]. Another kind of software problem concerns run-time exceptions raised by programs and not handled by the code [14].

Some related work can be found in computer science, in the notions of program synthesis. It consists in translating a property on inputs and outputs of a system, expressed in temporal logics, into a lower-level model, typically in terms of transition systems. For example, it is proposed in a UML-related framework, with the synthesis of StateChart from Live Sequence Charts [16]. These program synthesis approaches do not seem to have been aware of Discrete Control Theory, or reciprocally: however there seems to be a relationship between them, as well as with game theory, but it is out of the scope of this paper. Also, interface synthesis [7] is related to Discrete Controller Synthesis. It consists in the generation of interfacing wrappers for components, to adapt them for the composition into given component assemblies w.r.t. the communication protocols between them.

3 The BZR Language for Tool-Supported Design

Given our goal of combining adaptivity and dependability in adaptive and reconfigurable computing systems, we will combine reactive languages and models with the autonomic computing structures and objectives. A central aspect in our tool-supported approach is the specification and programming language Heptagon/BZR, which provides for high-level design of controllers, and encapsulates discrete control as a compilation operation.

Contracts on nodes are defined in the Heptagon/BZR language [19] using a behavioral contract syntax [13]. It allows for the declaration, using the **`with`** statement, of *controllable variables*, the value of which are not defined by the programmer. These free variables can be used in the program to describe choices between several transitions. They are defined, in the final executable program, by the controller computed off-line by DCS, according to the Boolean expression given in the **`enforce`** statement. Knowledge about the environment such as, for instance event occurrence order can be declared in an **`assume`** statement. This is taken into account during the computation of the controller with DCS. Heptagon/BZR compilation invokes a DCS tool, and inserts the synthesized controller in the generated executable code, which has the same structure as above:



(a) Observer: always 1 between 2 and 3 (b) Contract node for task insertion

Fig. 4. Observer and contract node

reset and *step* functions. Figure 4(b) shows an example of contract coordinating two instances of the `delayable` node of Figure 2(a), as assembled together in the `twotasks` node of Figure 3. The `insert` node has a **`with`** part declaring controllable variable c_r, and the **`enforce`** part asserts the property to be enforced by DCS: **`not`** *err*. The **`assume`** part is set to **`true`**, meaning that there is no assumption on the environment. The contract can itself feature a program, typically automata observing traces and defining states, to express a variety of safety properties. For example, an error state can be defined where the intended property is false, with the intention to keep it outside an invariant subspace. Such an observer is illustrated in Figure 4(a) : given input flows for the starting and stopping events of three tasks, it outputs value true on flow *err* when a sequence is observed such that task 3 is started (upon s_3) after task 2 (upon its end event e_2), without a complete execution of task 1, from s_1 to e_1, having taken place in between : this sequence violates the property that we have always 1 between 2 and 3. The contract in Figure 4(b) uses this observer for having always an execution of the simple task between two executions of the delayable task; this amounts to make invariant the state space where *err* is false. To enforce this, *cr* will be used by the synthesized controller to delay the starting of the delayable task until a full execution of the other one ends. The constraint produced by DCS can have several solutions: the Heptagon/BZR compiler generates deterministic executable code by favoring, for each controllable variable, value **`true`** over **`false`**, in the order of declaration.

The need for modularity comes when designs become complex. Advantages of our DCS-based approach, more constructive than classical verification, are:

– (i) high-level language support for controller design (tedious and error-prone to code manually at lower C or Java level) with declarative objectives ;
– (ii) correctness of the controller, w.r.t. the objectives, by definition of the algorithms (hard to guarantee manually) ;
– (iii) maximal permissiveness of controllers : they are minimally constraining, and in that sense optimally flexibile (even harder to obtain manually);
– (iv) automated formal synthesis of these controllers (rather than tedious hand-writing followed by verification);
– (v) automated executable code generation in C or Java.

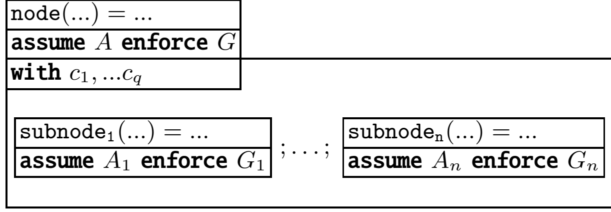


Fig. 5. Modular contracts in Heptagon/BZR

However, when considering a large number of managers, this monolithic approach might not succeed, because exploring the large state space is very time consuming, and can fail due to computing resource limits, which limits the scalability of the approach. Furthermore, a modification, even partial, leads to a recompilation of the overall coordinated composition invalidating previous generated codes which limits the re-usability of management components.

To address this issue, we exploit modular DCS, where the control objectives can be decomposed in several parts, each part managed by a controller. Each controller manages a limited number of components. This decreases the state space to explore for the synthesis of each controller. The recompilation of a controller that has no impact on other controllers does not require the recompilation of the latter. This makes possible the re-use of controllers generated codes.

Modular contracts in Heptagon/BZR are based on the modular compilation of the nodes: each node is compiled towards one sequential function, regardless of its calling context, the inside called nodes being abstracted. Thus, modular DCS is performed by using the contracts as abstraction of the sub-nodes. One controller is synthesized for each node supplied with local controllable variables. The contracts of the sub-nodes are used as environment model, as abstraction of the contents of these nodes, to synthesize the local controller. As shown in Figure 5, the objective is to control the body and coordinate sub-nodes, using controllable variables $c_1, ..., c_q$, given as inputs to the sub-nodes, so that G is true, assuming that A is true. Here, we have information on sub-nodes, so that we can assume not only A, but also that the n sub-nodes each do enforce their contract : $\bigwedge_{i=1}^{n}(A_i \implies G_i)$. Accordingly, the problem becomes that: assuming the above, we want to enforce G as well as $\bigwedge_{i=1}^{n} A_i$. Control at composite level takes care of enforcing assumptions of the sub-nodes. This synthesis considers the outputs of local abstracted nodes as uncontrollable variables, constrained by the nodes' contracts. A formal description, out of our scope here, is available [13].

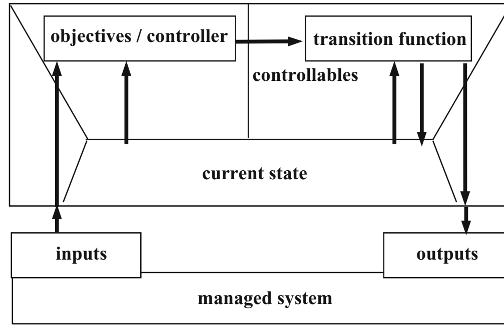


Fig. 6. Autonomic loop based on discrete control

4 Discrete Control-Based Autonomic Managers

The specification and programming language presented before, independently of AM design, can now be integrated in a method first exposed generally in Section 4.1, and then illustrated by brief summaries of works concerning reconfiguration control in DPR FPGA-based architectures in Section 4.2, and the coordination of administration loops in Section 4.3.

4.1 General Design Method

An interpretation of the MAPE-K loop of Figure 1 in terms of the discrete control framework of Section 2.2 is illustrated in Figure 6. For simplification, the Monitoring and Execution parts are considered as simply forwarding sensor inputs and action outputs to the ME. The Knowledge part is assimilated to the current state of the reactive system. The Analysis part is in charge of making decisions w.r.t. choices in reconfiguration, according to the adaption strategy or policy : we assimilate it to the specification of the control objectives, at design time, which is then transformed by DCS and compilation into a controller taking the same place at execution time. The decisions are encoded as values on choice points corresponding to the controllable variables, which are internal to the AM. The Planning part in assimilated to the transition function, computing, from the previous choice and current state (as well as inputs, here shown through the controller for simplification), which are the actions to be executed to implement them. Our discrete control-based approach is an effective tool-supported method for the design of AMs, which at the same time provides the designer with guarantees on the behaviour of the controller, through the use of DCS.

Typical modeled features appearing in the design of AM are related to computing systems and their common aspects in different fields. Of course the management of data-centers supporting the Cloud is not identical to the reconfiguration controllers of embedded FPGA architectures. However there are similarities in the manipulated objects, all resorting to computing.

The computing activities under control are tasks for which typical observability involves the state of activity (idle, active, waiting or other application specific aspects), as well as relevant events notifying end of task or check points ; controllability through choice points concerns firing the task or not, and choosing variants or modes of the delivered service distinguished by the use of different implementations and resources. Such tasks can be scheduled according to an application workflow: this knowledge can usefully be integrated in the model, in that it provides a predictive view of tasks to come in the future, which can influence the present choices. Resources can also feature observability (their usage level according to metrics, their charge level in the case of energy supply) and controllability (adding or removing computing resources, sleep modes).

The adaptation policies or strategies have to be explicited and formulated formally themselves, in the form of control objectives. The typically concern resource access control (exclusivity or bounded capacity), application termination (reaching a target state), fault tolerance (maintaining activity on other resources upon failures), or more elaborate sequencing patterns. These commonalities observed in case studies suggest a level of generality of the problem and the proposed solutions.

Granularity levels can be quite diverse, both in time or pace (the period of the loop can be in minutes or even hours) and detail (the managed computations can be whole systems). At the **lowest level of MEs**, computations should, just as usual, of course be as fast as possible and the possible overhead of monitoring and sensing should not interfere with system performance. Especially in parallel and distributed systems the feedback loop should not impose costly synchronizations.

At the **level of an AM**, it is indeed the case however that the MAPE-K loop is not supposed to run at that pace, and is mostly much slower or even sporadic when event-based. This depends completely on the level of the decision to be made, and on the duration of the execution of actions implementing it. As such, it is similar to the period of control systems begin determined by the dynamics of the process that has to be controlled. This dictates the maximal allowable period between two significants events to be observed, and not to be missed ; in turn, the latter gives an upper bound to the decision technique used, which should not cost more time (hence limiting e.g., optimization techniques).

At the **level of AMs coordination** the pace can be naturally considered even slower, and the feedback loop makes a step sufficiently rarely (compared to e.g., processor frequency) for enabling the use of synchronizations such as in distributed algorithms (e.g., leader election) when needed.

In our case, the most costly part in our method is DCS, but it is performed off-line and therefore is not limited by the ME dynamics, but rather by the design-time computing resources. The run-time cost is only that of the execution of a decision diagram function i.e., very low. However an important aspect is the size of the state space, in which DCS algorithms are exponential: therefore it is vital to determine the highest possible level of the model, abstracting away from fine-grain computations and from detailed fine-grain state-spaces.

4.2 Reconfiguration Control in DPR FPGA-Based Architectures

Dynamically reconfigurable hardware has been identified as a promising solution for the design of energy efficient embedded systems. However, its adoption is limited by the costly design effort including verification and validation, which is even more complex than for non dynamically reconfigurable systems. Therefore, we appley our tool-supported formal method to automatically design a correct control of the reconfiguration [1]. We design generic modeling patterns for a class of reconfigurable architectures, taking into account both hardware architecture and applications, as well as relevant control objectives. We validate our approach on case studies implemented on Dynamic Partial Reconfigurable (DPR) FPGA.

The considered class of architectures is presented informally through an example. Three levels are modeled separately, for which we will control the interactions according to global objectives:

- **architecture** is multiprocessor on n reconfigurable tiles $A1$–An, plus a general purpose processor $A0$ (e.g., ARM core). A tile Ai can be configured by uploading a bitstream encoding the function to be executed, and put to sleep mode with a *clock gated mechanism* to consume a minimum static power. A battery supplies energy, with a sensor for charge going up or down.
- **tasks** are defined with choices, upon request, between starting immediately or delaying ; between different bitstreams characterized each by : tiles used, WCET (Worst Case Execution Time), reconfiguration time, power peak.
- **application** is specified as a dependency graph between tasks: upon end notification of a task, requests are emitted for its following task(s);

The control problem is to use choices in order to satisfy global constraints according to resource state and activities in parallel or further in the application. The desired reconfiguration policy informally involves :

1. resource usage constraint: e.g., exclusive use of reconfigurable tiles $A1$-$A4$;
2. energy constraint: switch tiles to active mode if and only if needed;
3. power peak constraint: bounded by a maximum w.r.t battery level;
4. reachability: application graph execution can always finish once started;
5. optimizing e.g., global power peak is also possible [1].

A typical example of decision to be made is that, upon progress in the task graph, new tasks must be started by choosing the mode or bitstream compatible with available resources constraints, taking into account possible futures in the application, which can require to keep resources for a later task. Figure 7 summarizes, in the framework of Figure 6, what we need to formalize, and identifies controllable and uncontrollable variables, as well as the relevant state information, which determine the model abstraction level.

Generic models are shown in Figure 8, with patterns for modelling relevant states and controllable or uncontrollable events. Architecture is modeled in (a)

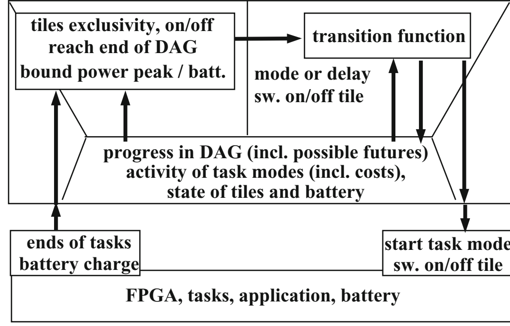


Fig. 7. Autonomic loop for DPR FPGA

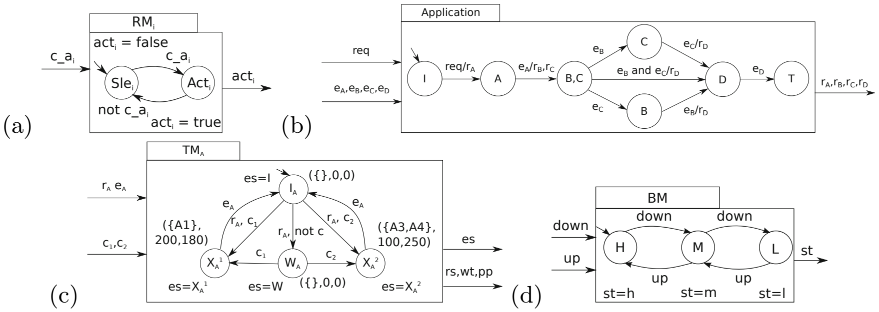


Fig. 8. Generic models: (a) tile i, (b) task graph, (c) two modes task, (d) battery

for each tile (with state act_i, controlled by c_a_i), and (d) battery observer (giving its state st, high, medium or low, according to received sensor values). In (c) a task example with two modes has uncontrollable requests r_i and end notifications e_i, and controllables c_1, c_2 to choose between modes, emitting state es and values characterizing them. In (b) an application example with 4 tasks $(A; (B||C); D)$ shows states recording progress in the graph, giving predictive model on which tasks can be fired in the future, in reaction to end notifications e_i, emitting requests r_i towards tasks, with terminal state T. In order to model possible behaviors of the system, these patterns are instantiated for all components, and composed in parallel: $(RM_1||...||RM_4||BM||TM_A||...||TM_D||Application)$ with values from modes of the active tasks being also composed, defining global values for the different resources metrics (e.g., sum of local pp_i into PP, union of rs).

Invariance and optimal control objectives are defined in contracts upon this global behavior model. Controllable variables are declared in the **`with`** statement, other inputs being uncontrollable inputs. The policy above can be formulated in a generic way [1] in terms of properties mentioned in Section 2.2: Objective 1 to 3 are *invariance* objectives on state variables and associated metrics constant values, e.g., for $3 : PP < (v_1 \text{ if } st = h \text{ else } v_2 \text{ if } st = m \text{ else } v_3 \text{ if } st = l)$. Objective 4 concerns *reachability* of terminal state T.

Two experimental case studies have been implemented to demonstrate the previous control models on real FPGAs [1]. After compilation towards executable code, the controller is running on a Microblaze soft core (i.e. $A0$) on the FPGA.

4.3 Coordination of Administration Loops

Real autonomic systems require multiple management loops, each complex to design, and possibly of different kinds (quantitative, synchronization, involving learning, ...). However their uncoordinated co-existence leads to inconsistency or redundancy of action. Therefore we apply our method to the discrete control of the interactions of managers [11]. We follow a component-based approach and explore modular discrete control, allowing to break down the combinatorial complexity inherent to the state-space exploration technique. It also allows re-using complex managers in different contexts without modifying their control specifications. We validate our method on a multiple-loop multi-tier system.

The administration loops and their need for coordination are considered in the context of JEE multi-tier applications which consist of: an apache web server receiving incoming requests, and distributing them with load balancing to a tier of replicated tomcat servers. The latter access to a database through a mysql-proxy server which distributes the SQL queries, with load balancing, to a tier of replicated mysql servers The global system running in the data-center consists of a set of such applications in parallel.

A set of autonomic managers are used to administrate the system: **Self-sizing** decides on the degree of replication of servers depending of the system over- or under-load measured through the CPU usage. It aims at lowering the resources usage while preserving the performance. It can add new replicas (which takes time), or remove some (considered immediate); each of these two actions can be inhibited. **Self-repair** targets a load balancer as well as replicated servers. It manages `fail-stop` failure detected through `heartbeat`. It aims at preserving the availability of the service. It triggers repair actions (taking time), which can be inhibited. **Consolidation** targets the global virtualized data-center. It adapts the computing capacity made available in a virtualized data-center, to either decrease or increase it. It can be controlled by delaying the actions.

Coordination problems can occur when e.g., several loops react to the same overload whereas one would have sufficed, or a failed load balancer leads to down-sizing followed by upsizing again right after repair. Also, consolidation requires to operate on a stable system in order to be consistent. The desired coordination policy informally involves the following constraints:

1. In a replicated tier, avoid size-up when repairing.
2. In a load-balanced tier, avoid size-down when repairing the load-balancer.
3. In general, avoid size-down in a successor tier when repairing a predecessor.
4. At global data-center level, when consolidating, avoid self-sizing or repairing.
5. Wait until repairs or add finish before consolidation decreasing, and until removals finish before increasing.

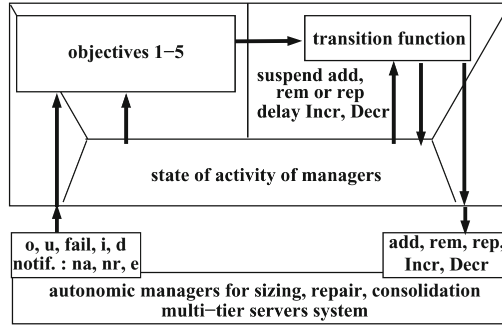
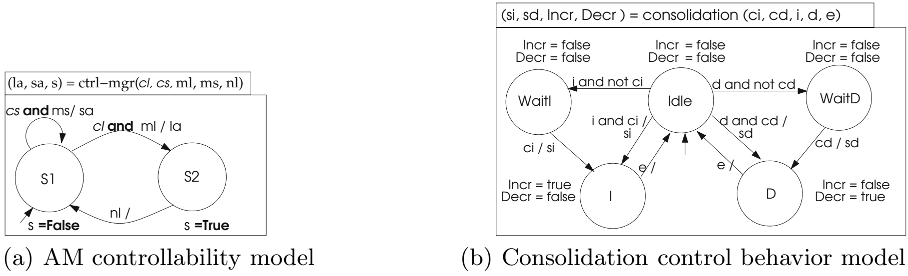


Fig. 9. Autonomic loop for Coordination

Figure 9 summarizes, in the framework of Figure 6, what we need to formalize. It shows that the relevant information here is the state of activeity of AMs, abstracting away from the way the individual controllers are working: we do not consider how they perform, but observability on their activity and controllability on their actions. The pace at which the coordination loop must work is defined by the input events, much slower than the underlying data-center computations.

Their coordination by modular control is based on generic models for each of them. **Self-sizing** control is an instance of node `ctrl-mgr` in Figure 10(a), with outputs `la` for long action `add`, `sa` for short action `rem` and `s` for busy state `adding` ; and with inputs for control: `ca` and `crm` for the actions, and for monitoring: `ml` for overload `o`, `ms` for underload `u`, and notification `nl` for adding `na`. This defines `(add, rem, adding) = self-sizing(ca, crm, o, u, na)`. **Self-repair** control is a simpler case, with only a long action of repairing, also an instance of `ctrl-mgr` with outputs: long action `rep`, and busy state `repairing` ; and inputs: control `ca`, failure `fail`, and notification of repair done `nr`. Unused parameters can be, for inputs, given the constant value `false`, and for outputs be left unused. This defines: `(rep, repairing) = self-repair(cr, fail, nr)`. **Consolidation** control in Figure 10(b) presents essentially the waiting mechanism of the delayable action of Figure 2(a), for each of its two long actions, the activity of which is given by `Incr` and `Decr`. In the initial Idle state, when i is **`true`** (increase is required), if ci is **`true`** it goes to I and emits si to start the increase plan, otherwise it goes to WaitI and awaits ci to go to Incr and emit si. When in Incr, it awaits until the notification of end e then returns back to Idle. The case for decrease is similar.

Coordination Objectives. The models are instantiated for each AM in the system, and their composition gives the global behavior before control. The control is specified on this composed behavior. We formalize the strategy above:



(a) AM controllability model (b) Consolidation control behavior model

Fig. 10. Modelling managers control

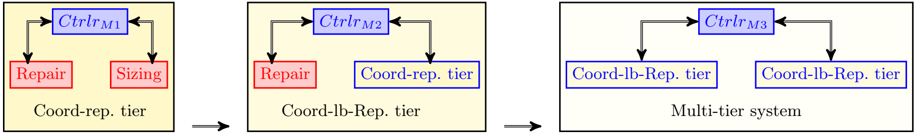


Fig. 11. Bottom-up re-use of nodes

1. **not** (repairing **and** add).
2. **not** (repairingL **and** rem) with repairingL from load-balancer self-repair.
3. between predecessor and successor tiers: **not** (repairing$_{pred}$ **and** rem$_{succ}$).
4. **not** ((Incr **or** Decr) **and** (repairing* **or** adding* **or** rem*)) where * stands for conjunction of all corresponding states.
5. **not** ((repairing* **or** or adding*) **and** sd) and **not** (rem* **and** si).

These properties can be grouped into a global contract to synthesize an invariance controller, but in order to have scalability and reusability, the system can be built up bottom-up as shown in Figure 11. A first node Coord-rep. tier cares for coordination within a replicated tier, with Objective 1. In order to be re-usable, the node has to have a contract that exposes to upper nodes that when a long action is started it is actually executed [11]. It is re-used in Coord-lb-Rep. tier, composed with the load-balancer self-repair, with Objective 2. For complete application, this node can be re-used twice, to form node Multi-tier system, with Objective 3. At data-center level, this last node is instantiated for each application, and composed with the consolidation model, with Objectives 4 and 5. This decomposition of DCS operations improves synthesis time dramatically, and the modular code generation enables distributing the controller. These controllers have been validated on an experimental data-center [11].

5 Discussion and Perspectives

Results. This invited paper makes an overview of our works addressing discrete control-based design of adaptive and reconfigurable computing systems, also called autonomic computing. We propose a tool-supported method, involving

a reactive language and its compiler encapsulating techniques stemming from discrete control theory. We validate the approach in domains ranging from software components and smart environments to hardware reconfigurable architectures. Our results demonstrate that control-based techniques for the design of autonomic loops can augment computing systems with, at the same time, self-adaptation capabilities and also predictability.

Limitations and extensions are of course made visible by our experiments, and there is still much to be done for supporting efficiently at the same time predictable and adaptive computing systems through behavioral model-based methods. Some perspectives are as follows.

Modeling is bound to be an important part of the work for spreading behavioral model-based control methods. Indeed, as is the case also for classical control-based approaches [29], formulating autonomic management problems in terms of systems behaviors and control objectives is hard, especially as computing systems are usually not at all designed to be controllable. Our work explored aspects related to computing resources like servers or cores on a multiprocessor architecture, but other aspects of computing systems should be considered, such as memory management issues (migration for proximity with a cache ; Software Transactional Memory contention management), communications (choice of media between e.g., wire, WiFi, Bluetooth, ..., w.r.t. throughput, energy cost, ...) or also security aspects. There often are favorable situations where the choices to be made at run-time are between predefined sets of configurations: the control is to enable efficient and appropriate use of these resources following dynamical changes in the environment and system. Changes anticipated at design time can be reified as variation points and exploited by the controller.

Expressivity and scalability of the modeling formalism need to be extended in order to improve the applicability of discrete control to realistic systems. Expressivity can be extended in order to account for more aspects of the systems, and incorporate logico-numeric properties [4], or have a combined control where, amongst the possibly several solutions satisfying the objective, determined by discrete control, a choice can be specified according to other criteria e.g., probabilistic or related to the continuous dynamics of the system.

Scalability requires efficiency in the DCS tools, which can benefit from progress made in Model Checking: algorithms have similar cost, intrinsically exponential in the worst case. This limits dramatically the above requirements of expressivity, as timed or hybrid formalisms have even higher costs. However, there is a way to attack this problem in the modeling phase, by identifying carefully chosen levels of abstraction, for acceptable overhead / cost, at least in small to medium systems. With modular and hierarchical decomposition of the problems, controllers can be built for the different levels of decision, with different paces, as mentioned in Section 4.1.

High-level languages and Domain Specific Languages (DSLs) are a useful help for usability of these methods: their aim is to allow of designers to describe their

systems in terms of the entities and components they manipulate, rather than in terms of the formalisms. we are building upon the experience of Nemo [12] in order to define a component-based systems language, extending known Archtecture Description Languages (ADLs), where not only assemblies of components are structurally defined, but also the different configurations and the reconfiguration behavior between them are described explicitly. As for other languages built on top of underlying tools, this poses problems for diagnostic, when there are several layers of translation between DSL and verification or synthesis tool. Also, the specifity of DCS is that it works like a constraints solving tool, making diagnostic in case of failure difficult to isolate and locate in the program.

Adaptive control is a desirable feature for autonomic computing, especially in open systems, typically the Cloud, where new components can enter, and others can leave the system to be managed. Another source of change is that new policies or strategies must be enforced, which translates into a change of control objective. Adaptive discrete control has hardly been studied in research on Discrete Event Systems, and can be seen as a new challenge motivated from applications such as Autonomic Computing. Directions to address it can be seen amongst having an upper controller switching between previously prepared controllers, or, for slow-paced systems, having a DCS phase at run-time (for reasonably sized subsystems) producing a new controller.

References

1. An, X., Rutten, E., Diguet, J.-P., le Griguer, N., Gamatié, A.: Autonomic management of dynamically partially reconfigurable fpga architectures using discrete control. In: In Proc. of the 10th International Conference on Autonomic Computing (ICAC 2013) (June 2013)
2. Årzén, K.-E.: al. Conclusions of the ARTIST2 roadmap on control of computing systems. ACM SIGBED (Special Interest Group on Embedded Systems) Review 3(3) (July 2006)
3. Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Guernic, P.L., de Simone, R.: The synchronous languages twelve years later. Proc. of the IEEE, Special issue on Embedded Systems 91(1), 64–83 (2003)
4. Berthier, N., Marchand, H.: Discrete Controller Synthesis for Infinite State Systems with ReaX. In: IEEE International Workshop on Discrete Event Systems, Cachan, France, pp. 420–427 (2014)
5. Brun, Y., et al.: Engineering self-adaptive systems through feedback loops. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 48–70. Springer, Heidelberg (2009)
6. Calinescu, R., Ghezzi, C., Kwiatkowska, M., Mirandola, R.: Self-adaptive software needs quantitative verification at runtime. Communications of the ACM 55(9), 69–77 (2012)
7. Chakrabarti, A., de Alfaro, L., Henzinger, T.A., Mang, F.Y.C.: Synchronous and bidirectional component interfaces. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 414–427. Springer, Heidelberg (2002)

8. Cheng, B.H.C., et al.: Software engineering for self-adaptive systems: A research roadmap. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009)
9. Chess, D.M., Palmer, C., White, S.R.: Security in an autonomic computing environment. IBM Syst. J. 42(1), 107–118 (2003)
10. de Lemos, R., et al.: Software engineering for self-adaptive systems: A second research roadmap. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 7475, pp. 1–32. Springer, Heidelberg (2013)
11. Delaval, G., Gueye, S.M.-K., Rutten, E., De Palma, N.: Modular coordination of multiple autonomic managers. In: Proceedings of the 17th International ACM Sigsoft Symposium on Component-based Software Engineering, CBSE 2014, pp. 3–12. ACM, New York (2014)
12. Delaval, G., Rutten, É.: A domain-specific language for multitask systems, applying discrete controller synthesis. EURASIP Journal on Embedded Systems 2007, 084192 (2007)
13. Delaval, G., Rutten, E., Marchand, H.: Integrating discrete controller synthesis into a reactive programming language compiler. Discrete Event Dynamic Systems 23(4), 385–418 (2013)
14. Gaudin, B., Vassev, E.I., Nixon, P., Hinchey, M.: A control theory based approach for self-healing of un-handled runtime exceptions. In: Proceedings of the 8th ACM International Conference on Autonomic Computing, ICAC 2011, pp. 217–220. ACM, New York (2011)
15. Halbwachs, N., Baghdadi, S.: Synchronous modeling of asynchronous systems. In: Sangiovanni-Vincentelli, A.L., Sifakis, J. (eds.) EMSOFT 2002. LNCS, vol. 2491, pp. 240–251. Springer, Heidelberg (2002)
16. Harel, D., Kugler, H., Pnueli, A.: Synthesis revisited: Generating statechart models from scenario-based requirements. In: Kreowski, H.-J., Montanari, U., Orejas, F., Rozenberg, G., Taentzer, G. (eds.) Formal Methods in Software and Systems Modeling. LNCS, vol. 3393, pp. 309–324. Springer, Heidelberg (2005)
17. Harel, D., Naamad, A.: The statemate semantics of statecharts. ACM Trans. Softw. Eng. Methodol. 5(4), 293–333 (1996)
18. Hellerstein, J., Diao, Y., Parekh, S., Tilbury, D.: Feedback Control of Computing Systems. Wiley-IEEE (2004)
19. Heptagon/BZR language, `http://bzr.inria.fr`
20. Huebscher, M.C., McCann, J.A.: A survey of autonomic computing: degrees, models, and applications. ACM Comput. Surv. 40(3), 7:1–7:28 (2008)
21. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer 36(1), 41–50 (2003)
22. Kloukinas, C., Yovine, S.: Synthesis of safe, qos extendible, application specific schedulers for heterogeneous real-time systems. In: Proceedings of 15th Euromicro Conference on Real-Time Systems, pp. 287–294 (July 2003)
23. Marchand, H., Bournai, P., Le Borgne, M., Le Guernic, P.: Synthesis of discrete-event controllers based on the signal environment. Discrete Event Dynamic Systems: Theory and Applications 10(4), 325–346 (2000)
24. Marchand, H., Rutten, É.: Managing multi-mode tasks with time cost and quality levels using optimal discrete control synthesis. In: 14th Euromicro Conference on Real-Time Systems (2002)

25. Patikirikorala, T., Colman, A., Han, J., Wang, L.: A systematic survey on the design of self-adaptive software systems using control engineering approaches. In: ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), Zurich, Switzerland (2012)
26. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete event processes. SIAM J. Control Optim. 25(1), 206–230 (1987)
27. Wallace, C., Jensen, P., Soparkar, N.: Supervisory control of workflow scheduling. In: Advanced Transaction Models and Architectures Workshop (ATMA), Goa, India (1996)
28. Wang, Y., Lafortune, S., Kelly, T., Kudlur, M., Mahlke, S.: The theory of deadlock avoidance via discrete control. In: Principles of Programming Languages, POPL, Savannah, USA, pp. 252–263 (2009)
29. Zhu, X.: Application of control theory in management of virtualized data centres. In: Fifth International Workshop on Feedback Control Implementation and Design in Computing Systems and Networks (FeBID), Paris, France (2010), http://controlofsystems.org/febid2010/program.html