Autonomic Management of Dynamically Partially Reconfigurable FPGA Architectures using Discrete Control

Xin An, Eric Rutten
INRIA, Grenoble, France *(xin.an@inria.fr, eric.rutten@inria.fr)*

Jean-Philippe Diguet, Nicolas le Griguer
Lab-STICC, Lorient, France *(jean-philippe.diguet@univ-ubs.fr, nicolas.le-griguer@univ-ubs.fr)*

Abdoulaye Gamatié
LIRMM, Montpellier, France *(abdoulaye.gamatie@lirmm.fr)*

Abstract

This paper targets the autonomic management of dynamically partially reconfigurable hardware architectures based on FPGAs. Discrete Control modelled with Labelled Transition Systems is employed to model the considered behaviours of the computing system and derive a controller for the control objective enforcement. We consider system application described as task graphs and FPGA as a set of reconfigurable areas that can be dynamically partially reconfigured to execute tasks. The computation of an autonomic manager is encoded as a Discrete Controller Synthesis problem w.r.t. multiple constraints and objectives e.g., mutual exclusion of resource uses, power cost minimization.

keywords: Hardware Architectures, Dynamically Partially Reconfigurable FPGA, Discrete Control.

1 Control of autonomic hardware

Controlling FPGAs. We apply the autonomic framework to the context of FPGAs (Field Programmable Gate Arrays), hardware devices that compute a logic function by configuring its gates in a programmable way. A recent progress is *dynamically partially reconfigurable* (DPR) FPGAs. They support partial reconfigurations where only part of gates are reconfigured and reconfigurations to be performed at runtime. Autonomic computing has been seldom applied to such hardware systems, though they represent a significant case of its relevance.

Control for autonomic management. We adopt control techniques to design the *MAPE-K* (Monitor, Analyse, Plan, Execute, based on Knowledge). Formal models are used to describe the possible behaviours of the system under design, and control objectives giving the adaptation policy are specified separately. A controller is then derived based on the system models and objectives. The use of classical control techniques and models, typically these based on continuous time dynamics and differential equations, has been explored for various computing systems [6] and sometimes applied for hardware architectures [5]. A similar approach can be adopted by using *discrete control* techniques, where systems are considered from the viewpoint of events and states. The behaviours can then be modelled in the form of Petri nets or automata for synchronisation [10].

Discrete control for autonomic FPGAs. We apply *discrete control* for the autonomic management of DPR FPGA based embedded systems. A systematic modelling framework is proposed, where system application behaviour, task implementations and executions, architecture reconfigurations and environment are modelled separately by using *Labelled Transition Systems* (LTS) or *automata*. *Discrete Controller Synthesis* (DCS) supported by a programming language and synthesis tool has been applied to compute an autonomic manager.

2 Background notions

2.1 FPGA-based architectures

Basic reconfigurable cell. A FPGA is composed of an array of logic cells and programmable routing channels to implement custom hardware functionalities. A program consists of one or more *bitstream*s, which are binary files storing information to configure logical cells and the routing switches. Recent large FPGAs contain more than 200K logic cells that can be combined and interconnected to implement very complex designs. Multicore architectures with tens of large hardware accelerators and processors can be implemented.

Run-time partial reconfiguration. In the new generation of FPGAs, the hardware configuration can be updated at run-time by using the partial reconfiguration feature. They have the ability to reconfigure hardware during the running of the static part, i.e., the part which does not contain any reconfigurable area. It assumes that the hardware reconfiguration does not disturb the execution

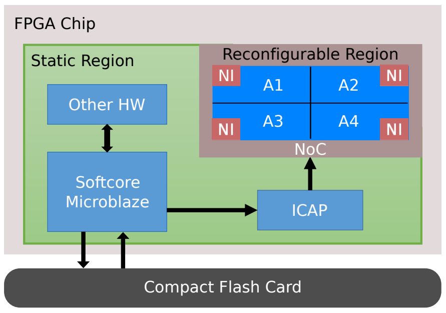


Figure 1: FPGA with a microblaze softcore.

of the application. The bitstreams therefore cover only some regions of the FPGA array.

Such DPR FPGAs make them suitable for addressing constraints on resources (re-using some areas for different functions for applications that can be partitioned into phases) by adapting resources to available parallelism according to environment variations. DPR FPGAs are a trade-off : that they are slower than dedicated Application-Specific Integrated Circuits (ASIC), but much faster than using general purpose CPUs.

Management of reconfiguration. From a technical viewpoint, each hardware configuration file used for the different implementations of the partially reconfigurable regions is stored into a compact flash card. It can be loaded by a processor (e.g. microblaze, which is a 32-bit soft-core processor as implementable on Xilinx FPGAs). It performs the reconfiguration using the ICAP (Internal Configuration Access Port) as in Figure 1.

The runtime management of reconfiguration involves a control loop, taking decision according to events monitored on the architecture, choosing the appropriate next configuration to install, and executing appropriate reconfiguration actions. The architecture dynamism increases the design complexity, for which a complete tool-chain is lacking [8]. Due to the relative novelty of DPR technologies, the management of reconfiguration has to be designed manually for important parts.

Amongst different approaches to address this issue, we investigate the adoption of an autonomic computing approach for the design of reconfiguration control. The MAPE-K structure is based on behavioural models (in the form of automata) for the knowledge about the reconfigurability of these hardware platforms, and discrete control techniques for designing the adaptation policies.

2.2 Discrete control

We consider the modeling framework [1] based on *labelled transition systems* (LTSs) and their parallel composition. LTSs are defined by a finite set of states, between which there are transitions (from source state to target state) with a label of form `c / a`: a firing condition `c` and an action `a`. When a LTS is in some state, if there is a transition for which the condition is true, then it is taken and the next state will be its target state. At the same time the action part will take the value `true`. Two or more LTSs can be composed (noted formally by "|"), representing that they run in parallel: one global step corresponds to one local step for every LTS.

The formalism of LTSs can be used to apply *discrete controller synthesis* (DCS), a formal operation on automata [3, 7]. DCS is an automatic and constructive method to ensure required properties on system behaviors. It applies to an LTS (originally uncontrolled), where inputs $\mathscr{I}$ are partitioned into two subsets, $\mathscr{I}_u$ and $\mathscr{I}_c$, the *uncontrollable* and *controllable* inputs. It takes into account some *control objectives*: properties that must be enforced by control. A controller is synthesized automatically, if it exists, from given LTS's and *objectives*, by applying appropriate algorithms [7] (not detailed here). Its purpose is to constrain the values of controllable variables, in function of states and of uncontrollable inputs, such that system behaviors satisfy the given objectives. The controller is *maximally permissive*, meaning that it allows the largest possible set of correct behaviors.

2.3 Discrete control as MAPE-K

Figure 2(a) shows the MAPE-K architecture of an autonomic system with a loop defining basic notions of Managed Element (ME) and Autonomic Manager (AM). The managed element, system or resource is monitored through sensors. An analysis of this information is used, in combination with knowledge about the system, to plan and decide upon actions. These reconfiguration operations are executed, using as actuators the administration functions offered by the system API. Self-management issues include self-configuration, self-optimisation, self-healing (fault tolerance and repair), and self-protection.

Autonomic managers work in closed loop: for this, one design methodology is to apply techniques from

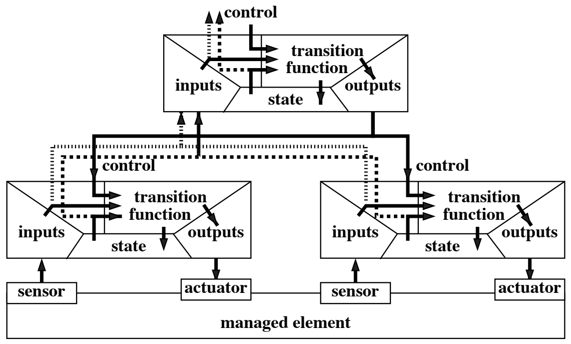
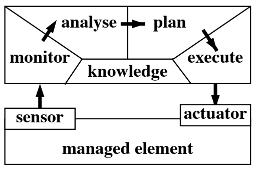
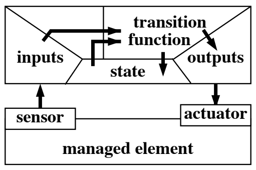


Figure 3: Autonomic coordination for multiple AMs.

(a) 

(b) 

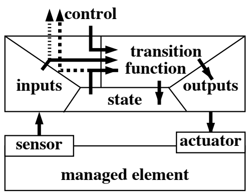
(c) 

Figure 2: Autonomic system: (a) the MAPE-K manager; (b) FSM autonomic manager; (c) controllable AM.

Control Theory [6], with the advantage of ensuring interesting properties on the resulting behaviour of the controlled system e.g., stability, convergence, reachability or avoidance of some evolutions. In most cases, continuous models are used, typically for quantitative aspects. More recently, some works relied on Discrete Event Systems (DES), using supervisory control [3], typically for logical or synchronisation purposes e.g., deadlock avoidance in multithreaded programs [10]. They are based on reactive systems models such as Petri nets or Finite State Machines (FSM), which we also call automata. As shown in Figure 2(b), this instantiates the general autonomic loop with knowledge on possible behaviours represented as a formal state machine, and planning and execution in the form of the automaton transition function with a control output, which will trigger the actuator.

Basic features required for a system to be managed in an autonomic fashion have been identified in previous work e.g., in the context of component-based autonomic management [9]: for an ME to be manageable it must be observable and controllable. The manager transforms flows of observations into flows of control choices and actions. Observability translates into outputs, as shown by dashed arrows in Figure 2(c) for an FSM AM, exhibiting (some) of the knowledge and sensor information (raw, or analysed); this can feature state information on the AM itself or of MEs below. Controllability translates to having the AM accept some influence on the decision, and it corresponds to additional input for control, as in Figure 2 for an FSM AM. Its values can be used in the guards and exhibit choices between different transitions.

This builds up to a hierarchical framework as in the structure shown in Figure 3. Given that AMs have been made observable and controllable, an upper-level AM can perform their coordination using their additional control input to enforce a policy. Considering the case of FSM managers makes it possible to encode the coordination problem as a DCS problem. The controller of this upper-level AM is synthesised by DCS.

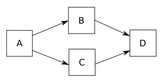
3 DCS for managing DPR architectures

We present the computing systems of interest through an illustrative example, first informally, then in the model.

3.1 DPR FPGAs

Hardware architecture. We consider a multiprocessor architecture implemented on an FPGA chip (see Figure 1), which includes a general purpose processor: Softcore Microblaze, and a reconfigurable area divided into four tiles: $A1$–$A4$. The communications between architecture components are achieved by a *Network-on-Chip* (NoC). Each processor and reconfigurable tile implements a NoC Interface (NI). Reconfigurable tiles can be combined and configured to implement and execute tasks by loading predefined bitstreams.

The architecture is equipped with a battery supplying the platform with energy. Regarding power management, an unused reconfigurable tile Ai can be put into sleep mode with a *clock gated mechanism* such that it consumes a minimum static power.

a) 

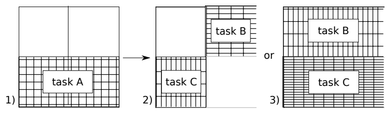
b) 

Figure 4: a) DAG application specification, and b) System configurations and reconfiguration.

Application software. We consider system functionality described as a *directed, acyclic task graph* (DAG). A DAG consists of a set of *nodes* representing the set of tasks to be executed, and a set of directed *edges* representing the precedence constraints between tasks. Figure 4(a) shows an example consisting of four tasks.

In our framework, we suppose each task performs its computation with the following four control points:

- *being requested* or invoked;
- *being delayed*: requested but not yet executed;
- *being executed*: to be executed on the architecture;
- *notifying execution finish*, once it reaches its end.

Occurrences of control points *being requested* and *notifying finishes* depend on runtime situations, and are thus unpredictable and uncontrollable. The way of *delaying* and *executing* tasks is taken charge by a runtime manager aiming to achieve system objectives.

Task implementation. Given a hardware architecture, a task can be implemented in various ways characterised by various parameters of interest, such as used reconfigurable tiles (*ur*), worst case execution time (WCET) (*wt*), and power peak *pp*. For example, two implementations of task *A* can be:

- *A* on $A1$: $wt = 50$, $pp = 20$;
- *A* on $A3+A4$: $wt = 10$, $pp = 30$;

In this preliminary work, we assume that WCET represents the time cost induced from the start of bitstream loading to the end of task execution. Among possible task implementations, a runtime manager is in charge of choosing the best according to system objectives.

System reconfiguration. Figure 4(b) shows three system configuration examples. In configuration 1, task A is running on tiles A3 and A4 while tiles A and B are set to the sleep mode. Configurations 2 and 3 show two scenarios with tasks B and C running in parallel. Once task A finishes its execution according to the graph of Figure 4(a), the system can go to either configuration 2 or configuration 3 depending on the system requirements. For example, if the current state of the battery level is low, the system would choose configuration 2 as configuration 3 requires the complete FPGA working surface and therefore consumes more power.

System objectives. System objectives define the system functional and non-functional requirements. This section gives the objectives considered in the paper, and categorises them as logical and optimal control objectives. Generally speaking, logical objectives concern state exclusions, whereas optimal objectives target the states associated with optimal costs. Considered logical and optimal control objectives are as follows:

1. resource usage constraint: exclusive uses of reconfigurable areas $A1$-$A4$;
2. energy reduction constraint: switch areas to sleep mode when executing no task;
3. power peak constraint: power peak of hardware platform is constrained w.r.t battery levels;
4. minimise power peak of hardware platform.

More system objectives can be addressed in our framework. We refer the readers to [2] for more details.

3.2 System modelling as a DCS problem

We specify the modelling of the computing system behaviour and control in terms of labelled automata. System objectives are defined based on the models. We focus on the management of computations on the reconfigurable tiles and dedicate the processor area $A0$ exclusively to the resulting controller.

Architecture behaviour. The architecture (see Figure 1) includes a processor, four reconfigurable tiles $\{A1, A2, A3, A4\}$ and a battery. Each tile has two execution modes, and the mode switches are controllable. Figure 5(a) gives the model of the behaviour of tile *Ai*. The mode switch action between Sleep (*Sle*) and Active (*Act*) depends on the value of the Boolean controllable variable c_a_i. The output act_i represents its current mode.

The battery behaviour is captured by the automaton in Figure 5(b). It has three states labelled as follows: H (high), M (medium) and L (low). The model takes input from the battery sensor, which emits level *up* and *down* events, and keeps track of the current battery level through output st.

Application behaviour. Software application is described as a DAG, which specifies the tasks to be executed and their execution sequences and parallelism. Its execution behaviour can be captured by using an automaton with states representing the set of tasks that are active in current states. The firing conditions of transitions are task *finish notifications*, which could enable the executions of (some of) its immediate succeeding tasks by emitting *start* requests of these tasks. An algorithm to systematically construct such an scheduling automaton for a DAG can be found in [2].

Task execution behaviour. In consideration of the four control points of task executions (see Section 3.1), the execution behaviour of task A associated with two implementations (see Section 3.1) can be modelled as Figure 5(c). It features an initial *idle* state I_A, a *wait* state W_A, and two *executing* states X_A^1, X_A^2 corresponding to two implementations of task A. Controllable variables are integrated in the model to encode the controllable points: being delayed and executed. Upon the receipt of *start* request r_A, task A goes to either:

- *executing* state $X_A^i, i \in \{1,2\}$ if the value of *controllable* variable c_i leading to X_A^i is *true*, or
- *wait* state W_A if delayed, i.e., the value of Boolean expression $c = \bigvee c_i, i \in \{1,2\}$ is false.

From wait state W_A, upon the receipt of event c_i, it goes to execution state X_A^i. When the execution of task A finishes, i.e., the end notification event e_A is received, the automaton goes back to *idle* state I_A. Output *es* represents its execution state.

Local execution costs. The execution costs of different task implementations are different. Three cost parameters are considered (see Section 3.1). We capture them by associating cost values denoted by a tuple (rs, wt, pp) with the states of task models, where: $rs \in 2^{RA}$ (RA is the set of architecture resources), $wt \in \mathbb{N}$ (a WCET value) and $pp \in \mathbb{N}$ (a power peak). The costs associated with

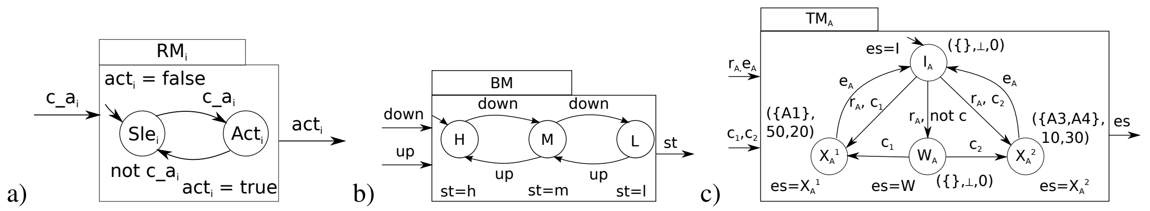


Figure 5: Models RM_i for tile Ai, BM for battery, and TM_A for the execution behavior of task A.

executing states are the values associated with their corresponding implementations. For *idle* and *wait* states, apparently $rs = \emptyset, pp = 0$. However, the wt values for *idle* and *wait* states depend on the execution times of their precedent tasks. We therefore represent it by using a special symbol $\perp$, and thus we have $wt \in \mathbb{N} \cup \perp$.

Global system behaviour model. The parallel composition of control models for reconfigurable tiles RM_1-RM_4, battery BM and tasks TM_A-TM_D, plus scheduler Sdl comprises the system model: $\mathscr{S} = RM_1|...|RM_4|BM|TM_A|...|TM_D|Sdl$ with initial state $q_0 = (Sle_1, ..., Sle_4, H, I_A, ..., I_D, I)$. Sdl represents the automaton that captures the application behavior as discussed in Section 3.2. It represents all the possible system execution behaviours in the absence of control (i.e., a runtime manager is not yet integrated).

Global costs. A system state q is a composition of local states (denoted by $q_1, ..., q_n$), and we define its global cost from the local ones as follows:

- used resources: union of used resources associated with the local states, i.e., $rs(q) = \bigcup rs(q_i), 1 \leq i \leq n$;
- power peak: the sum of values associated with the local states, i.e., $pp(q) = \sum(pp(q_i), 1 \leq i \leq n)$;

System objectives. The two types of system objectives: logical and optimal ones, can then be defined in terms of the states and the costs defined on the states or paths of the model. For example, Objective 1) exclusive uses of reconfigurable areas A1-A4 by tasks is defined by $\forall q_i, q_j \in q, i \neq j$, that $rs(q_i) \bigcap rs(q_j) = \emptyset$. We refer the readers to [2] for the detailed definition.

We have validated our models and manager computations experimentally by implementing a video processing system on an ML605 board from Xilinx containing an FPGA. The BZR language has been used to encode system models and objectives, and generate a correct autonomic manager in C code for the system. They are detailed elsewhere [2] due to lack of space.

4 Conclusion and Perspectives

Reconfigurable architectures, especially DPR FPGAs, constitute a platform for adaptive computing that is gaining widespread use. They are a typical target for autonomic computing approaches, although they are not often explicitly tackled that way. In this paper, we proposed a systematic modeling framework for DPR FPGA based embedded systems, and applied formalisms and tools from discrete control to encode and perform the autonomic manager computation as a DCS problem.

Perspectives include the ongoing work to enrich our models with reconfiguration costs, and the use of modular synthesis and compilation [4] for manager computing. We are working on more experimental systems, which will validate more completely our approach.

References

[1] ALTISEN, K., CLODIC, A., MARANINCHI, F., AND RUTTEN, E. Using controller-synthesis techniques to build property-enforcing layers. In *Proceedings of the European Symposium on Programming (ESOP'03)* (2003), pp. 174–188.

[2] AN, X., RUTTEN, E., DIGUET, J.-P., LE GRIGUER, N., AND GAMATIÉ, A. Autonomic management of reconfigurable embedded systems using discrete control: Application to fpga. Research Report RR-8308, INRIA, May 2013. `http://hal.inria.fr/hal-00824225`.

[3] CASSANDRAS, C., AND LAFORTUNE, S. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, 2008.

[4] DELAVAL, G., MARCHAND, H., AND RUTTEN, E. Contracts for modular discrete controller synthesis. In *Conf. on Languages, Compilers, and Tools for Embedded Systems* (2010), pp. 57–66.

[5] EUSTACHE, Y., AND DIGUET, J.-P. Specification and os-based implementation of self-adaptive, hardware/software embedded systems. In *Conf. on Hardware/Software codesign and system synthesis (CODES/ISSS)* (2008), pp. 67–72.

[6] HELLERSTEIN, J., DIAO, Y., PAREKH, S., AND TILBURY, D. *Feedback Control of Computing Systems*. Wiley, 2004.

[7] MARCHAND, H., AND SAMAAN, M. Incremental design of a power transformer station controller using a controller synthesis methodology. *IEEE Trans. on Soft. Eng. 26*, 8 (2000), 729 –741.

[8] SANTAMBROGIO, M. D. From reconfigurable architectures to self-adaptive autonomic systems. *IJES 4*, 3/4 (2010), 172–181.

[9] SICARD, S., BOYER, F., AND PALMA, N. D. Using components for architecture-based management: the self-repair case. In *Proc. Conf. ICSE* (2008).

[10] WANG, Y., LAFORTUNE, S., KELLY, T., KUDLUR, M., AND MAHLKE, S. The Theory of Deadlock Avoidance via Discrete Control. In *Conf. POPL* (2009).