Discrete and Logico-numerical Control for Dynamic Partial Reconfigurable FPGA-based Embedded Systems : a Case Study

Soguy Mak-Karé Gueye, Gwenaël Delaval, Eric Rutten, Jean-Philippe Diguet

Discrete and Logico-numerical Control for Dynamic Partial Reconfigurable FPGA-based Embedded Systems : a Case Study

Soguy Mak-Karé Gueye*, Gwenaël Delaval*, Éric Rutten* and Jean-Philippe Diguet†
*Univ. Grenoble Alpes, CNRS, Inria, LIG, F-38000 Grenoble France
†CNRS, Université Bretagne Sud, LAB-STICC, F-56321 Lorient, France
Email: {soguy-mak-kare.gueye,gwenael.delaval,eric.rutten}@inria.fr, jean-philippe.diguet@univ-ubs.fr

***Abstract*— Embedded systems need to be more and more self-adaptive, in order to better manage their constrained resources, and to better take into account evolutions in their environment and in their computing architecture. They can benefit from Field Programmable Gate Array (FPGA) architectures , which supports Dynamic Partial Reconfiguration (DPR) of the running fonctions, enabling improved performance and power consumption. The reconfigurations need to be decided and controlled in a closed loop. We approach this problem by applying techniques from the area of Supervisory Control for Discrete Event Systems (DES), where the space of configurations at the different levels (application, tasks, hardware platform) are modeled with automata, using the tools Heptagon/BZR and ReaX. This paper contributes with (i) generic modeling of the behaviors and objectives ; (ii) applications of Discrete Controller Synthesis (DCS), especially logico-numerical control ; (iii) concrete implementation on a FPGA hardware platform for an embedded video processing case study.**

Keywords : Discrete Event Systems, Application, Hardware/Software Embedded Systems

I. Introduction

A. Reconfigurable FPGA-based computing architectures

Embedded systems can benefit from Field Programmable Gate Array (FPGA) architectures to improve performance compared to software-based system and to satisfy resource constraints, particularly by exploiting dynamic reconfiguration, which enables to realize adaptive hardware algorithms e.g., to meet performance or to reduce power consumption. Furthermore, Dynamic Partial Reconfiguration (DPR) improves the flexibility of FPGA by enabling reconfiguration of some subsystem, while the rest continues running in parallel.

The reconfigurations need to be decided and controlled in a closed loop. Changing mission parameters of embedded systems might lead to reorganizing the processings and the allocation of the resources based on their redefined priority and execution requirements. This management can be automated by control loops as addressed in Autonomic Computing [10]. Manual programming of controllers could be error-prone, costly and complex due to the combinatorial design space.

Instead, we propose a design approach based on techniques from the area of Supervisory Control for Discrete Event Systems (DES), where the space of configurations at the different levels (application, tasks architecture) are modeled with transitions systems, particularly automata. This approach produces correct-by-construction controllers enforcing desired control objectives, and avoids error-prone manual programming and tedious debugging.

B. Logico-numerical control with Heptagon/BZR

We use a programming language and its support tools, Heptagon/BZR [6] and ReaX [3], and their Discrete Controller Synthesis (DCS) capabilities. They provide us with high level programming for formal specification of possible configurations, symbolic DCS, as well as powerful compilers to automatically generate a correct executable code implemented in C. This point is particularly important for the concrete application of control techniques, which can therefore be integrated in the underlying FPGA-based hardware platform.

We are exploring this same approach of DES model-based design of reconfiguration controllers by working in parallel on different classes of computing systems. We consider small, embedded hardware systems [11], like in this paper, as well as larger, more distributed software systems like web servers in the Cloud [2], [5], or heterogeneous platforms, like in the Internet of Things (IoT) (e.g., smart buildings [15]). These works are distinguished by their very different target computing systems, w.r.t. execution platforms and application domains: therefore they bring complementary experience, towards the identification of generic modeling methods, in order to pose and solve logical control problems in computing systems.

In this particular work, we focus on features of, on the one hand, logico-numerical control, for control objectives involving input and state integer values w.r.t. variations in required performance values or priority levels ; and on the other hand, conditional objectives in order to specify complex objectives as a composition of multiple basic ones. We also exploit modular control for scalability.

C. Contributions

Novel results of this paper concern : (i) generic modeling of the behaviors and objectives for the reconfiguration control problem in DPR FPGA within a multi-layered framework ; (ii) applications of Discrete Controller Synthesis (DCS) to design controllers, beyond earlier work [1], especially logico-numeric control to react to requested

performance or QoS, and modularity for scalability ; (iii) concrete implementation on a FPGA hardware platform for an embedded video processing case study.

In the remainder, Section II introduces background notions, in FPGA and in DES ; Section III defines the targeted class of systems ; Section IV proposes systematic and generic models behaviours and objectives ; Section V presents a case study, and Section VI concludes.

II. Background

A. Dynamic Partial Reconfigurable FPGA

Dynamic Partial Reconfiguration (DPR) is a promising solution for applications that require high performance and high flexibility since it provides a way to modify (part of) the implemented logic in the FPGA. A dynamic partial reconfiguration consists in loading a bitstream which contains only the configuration for the target region of the FPGA. The unmodified regions can continue to work without interruption. This allows an FPGA with DPR capability to support more hardware implementations than would be possible statically. Hence, multiple applications can run on a single FPGA by sharing hardware resources, as in Section III.

Research works like [4], [7] have focussed on the dynamically reconfigurable hardware to meet both performance and cost required in most of embedded system. They demonstrate how dynamic reconfigurable hardware can be suitable for implementing compute-intensive embedded applications while minimizing the costs. [7] experienced sequences of reconfigurations to run a fingerprint recognition application. They show how the reconfiguration overhead can be minimized to avoid performance degradation when performing sequences of reconfigurations. However, they pay less attention on the design of the reconfiguration manager which must, at run-time, choose from several possible configurations, the appropriate one satisfying execution constraints under uncertainties at runtime.

Dynamic reconfiguration requires making decisions about the choice of whether to reconfigure or not, and if so, of the new configurations, depending on occurring events and sensor values in a system, on past events and sequences history, and on predictive knowledge about possible outcomes of reconfigurations. For the design of such feedback loops in computing systems, we follow the approach of Autonomic Computing [10] for performing self-configuration, self-optimization, self-healing or self-protection. This approach is exploring amongst others the application of Control Theory to design the feedback controllers themselves. Most of the state of the art concerns software systems, but some works also consider reconfigurable hardware [14].

In this paper, to deal with the logical control problems, we use Discrete Controller Synthesis (DCS) [13], through a reactive language and tool, Heptagon/BZR [6] and ReaX [3], which provides us with executable code generation for the implementations. We build upon previous work [1] where we had proposed generic behavioral models and objectives of invariance and optimization, in the context of a closed system. Here, we are considering novel approaches involving logico-numeric control and modular DCS to modeling a multi-layer system, where the reconfiguration controller must interact with a higher level deciding dynamically on e.g., varying levels of performance and quality required.

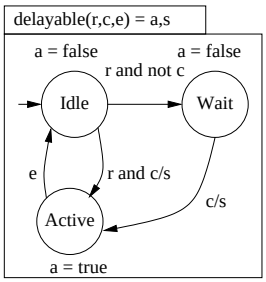
B. A reactive language and Discrete Controller Synthesis

In this section we briefly recall the formal methods and tools upon which we base our approach. 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, adapting the classical framework of [13] to models obtained from synchronous languages. An advantage of this approach is that it is tool-supported, by compilers like Heptagon/BZR (`http://bzr.inria.fr`) and by the DCS tool Reax [3]. This is essential for effective applications to concrete computing systems.

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 [9]. We adopt synchronous languages [8], 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 modeling asynchronous computations : actions can be asynchronously started, and their completion is waited for, without blocking other activity continuing in parallel. The Heptagon/BZR language [6] supports programming of mixed synchronous data-flow equations and 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.

Figure 1 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 operating system. Such



(a) Graphical syntax

```
node delayable(r,c,e:bool)
    returns (a,s:bool)
 let automaton
   state Idle  do a = false ; s = r and c
    until r and c then Active
     | r and not c then Wait
   state Wait do a = false ; s = c
    until c then Active
   state Active do a = true ; s=false

    until e then Idle
   end
  tel
```

(b) Textual syntax

Fig. 1: Delayable task

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 2(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) 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 satisfied in the original system. DCS, computes a control logic correct 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 of the system formally describes all possible behaviors, the correct and incorrect behavior based on the control objectives. It also exposes the controllability of the system, in the form of controllable in variablesputs, which is exploited by DCS to synthesize a control logic solution, if it exists.

3) Heptagon/BZR: We use the synchronous langage Heptagon/BZR [6] which integrates DCS in its compilation. It allows an easy use of DCS by introducing the notion of contract in the modeling of system. The contract is described declaratively and consists of three statements: **assume**, **enforce** and **with**. The contract contains predicates that the functioning of system must invariantly satisfy. 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, through DSC, generates a control logic that enforces the latter when controllable variables are defined in the model, declared as local variables in the **with** statement. 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.

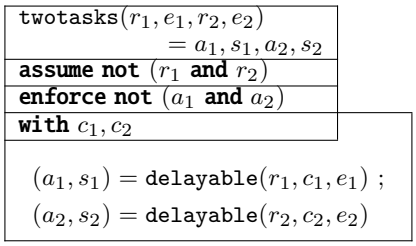


Fig. 2: Exclusion contract.

Figure 2 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 can not be 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 a request to start a task, hence leading the task to the waiting state whenever the other is active.

4) Modular contracts in Heptagon/BZR: Modular DCS consists in taking advantage of the modular structure of the system to control locally some subparts of this system. 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 subpart of the system, the model from which the controller is synthesized can be much smaller than the global model of the system. Therefore, as DCS is of practical exponential complexity, the gain in synthesis time can be high and it can be applied on larger and more complex systems.

Heptagon/BZR benefits from 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 3, 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

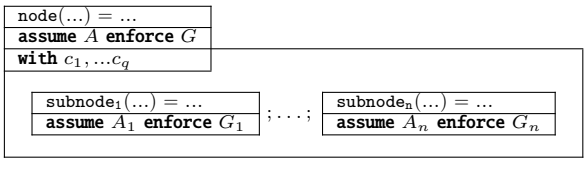


Fig. 3: Modular contracts in Heptagon/BZR.

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 is available [6].

III. FPGA RECONFIGURATION CONTROL PROBLEM

In this Section we describe informally the class of FPGA-based hardware systems which we target. We identify the aspects relevant for the reconfiguration control problem to be solved, so that the subsequent modeling is performed at the appropriate level of abstraction.

A. Target class of reconfigurable systems

a) DPR FPGA architecture and configurations: The system architectures we address are boards equipped with a dynamically reconfigurable hybrid FPGA (e.g. Xilinx Zynq) including ARM processors. As shown in Figure 4, two DDRAM memories are connected to the FPGA, the first one is usual and implements the ARM memory. The second one is used to store bitstreams; and also as shared memory for hardware and software tasks to communicate with each other. The FPGA programmable circuit is divided into tiles which will be shared by the tasks at runtime. The sharing leads naturally to perform sequences of reconfigurations so that all tasks requiring hardware can be executed. Tiles are considered to be equipped with a clock gating mechanism to put them to sleep mode, lowering their energy consumption when not used. They can also become unavailable e.g., due to a fault. The ARM processors have several level of DVFS (Dynamical Voltage Frequency Scaling) providing for different speeds, and different energy consumptions.

At any moment, a configuration of the architecture consists of the set of bitstreams uploaded on the tiles, each being the hardware implementation of a computation, the sleep/active or unavailable state of tiles, and the current DVFS level of the ARM processors.

b) Computations: They are organized as application composed of tasks. A task can be active or not. It could also be waiting for a processing resource occupied by another task. It can have different versions, defined by different algorithms and/or HW and SW implementations. They can differ in terms of used computing resources, performance and processing quality. For example some using more FPGA surface and exploiting parallelism, or smaller and more iterative.

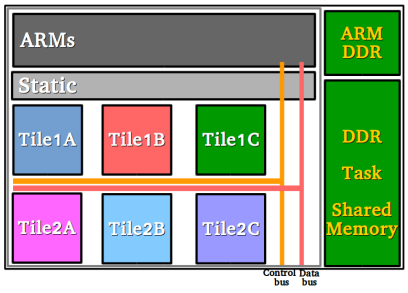


Fig. 4: FPGA board

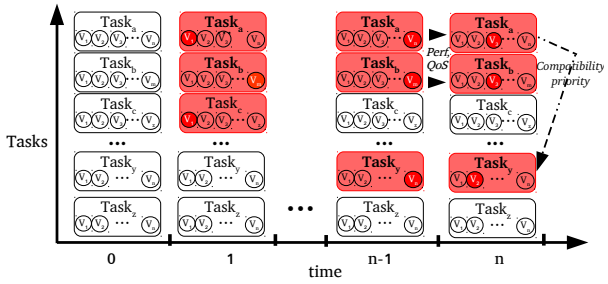


Fig. 5: Reconfigurations: subsets of active tasks, versions

The embedded system is provisioned offline with all the required bitstreams for the tasks. Tasks can not all be running simultaneously, due to surface limitations, so a subset can be active at each time according to a scheduling policy that must include the loading of bitstreams. The subset of tasks to be active at a given time is determined based on the environment and the system state and the processing results.

Figure 5 shows an example of execution timeline : the set of all available tasks, and for each of them its different versions, is shown vertically, at time 0 with none active. Across time, an application activates a subset of the tasks (e.g., a, b, c at time 1), each of them in a chosen version v_i, and this changes in later periods of time.

B. Management policy

The management policy to be enforced concerns performance, quality of service and energy optimization, typically : keeping the value of performance in defined intervals, e.g., task execution time between minand max-thresholds ; ensuring coherent usage of the resources e.g., mutual exclusion of tiles,or bounded number of users. ensuring configuration of the resources in order to reduce energy consumption while maximizing the performance.

C. Control loop

The global self-adaptation manager, is responsible for dynamically adapting the configuration of the system. This involves the processing resources, the active tasks

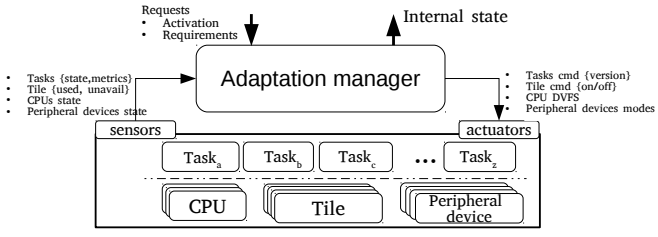


Fig. 6: Self-adaptation Manager

and the applications. The processing resources are reconfigured to reduce the energy consumption or to enhance the performance of the active tasks in order to meet their execution requirements. As shown in Figure 6, the management decisions are based on the subset of tasks that must be running as well as their execution requirements and priority, received from the mission level.

IV. Modelling the reconfigurable systems

A. Need for logico-numerical and conditional objectives

We are going to model the class of systems previously described as a Discrete Control problem, taking into account the following characteristics.

The execution requirements (for performance, quality of service) are expressed with numerical values, in the form of an acceptable range, i.e, interval. Therefore we need to express **logico-numerical objectives** such as maintaining the execution time of a task between a minimum threshold and a maximum threshold.

Also, for a more robust control, it is necessary to consider situations in which (part of) the basic objectives can not be satisfied. It can happen that measured performance of the system (e.g., execution time) is different from the one expected (e.g., the Worst Case Execution Time, WCET) or that two or more basic objectives become conflicting. We want to have controllers that enforce the (relatively simple) basic control objectives when possible, but are also able to adopt a degraded behavior when it becomes impossible i.e., to choose the best possible configuration while informing the upper decision layer (the mission level) so that it can modify the values of requirements, so that the controller can enforce basic objectives again. For this we will express **conditional objectives** such that DCS generates a robust control logic aware of the achievability of a control objective, that can disable it when not achievable, and able to resolve conflicting objectives through priorities.

B. Configurations space

1) Architectural Resources: We model individually each of the elements of the architecture at the appropriate level of abstraction described before. We use Hpetagon/BZR hierarchical automata : this improves the structure and clarity of the model, while the Heptagon/BZR compiler transforms them so that there is no impact on the DCS algorithms.

Figure 7 (a) shows the generic model of a tile. It is initially `Available`, and can become unavailable due to failure `f`, and be repaired `re` (both uncontrollable). In the `Available` state, it can be in `Sleep` state or in `Active` state, with transitions controllable by `c`.

Each ARM CPU processor can be characterized as in Figure 7 (b) by its levels of frequency, where XXX stands for corresponding values of consumption and speed, and a potential sleep mode, sparing energy. Peripheral devices of such a FPGA board, for example a camera, could

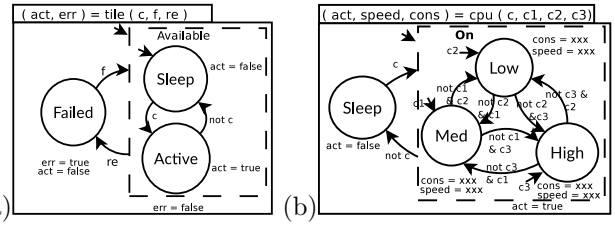


Fig. 7: Models : Reconfigurable tile (a), ARM CPU (b)

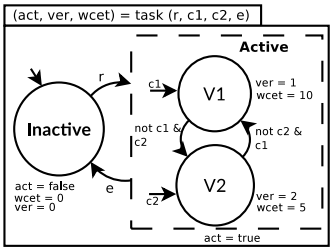


Fig. 8: Computation task

also be modeled, be characterized by more device-specific aspects like e.g., resolution, black / white vs. color image.

2) Computations: Figure 8 shows an example of the model of a task. The task is initially `Inactive`. When requested by `r` it becomes `Active`. It goes back to `Inactive` after completion notified by input `e`.

In `Active`, the task can be in versions `V1` or `V2` which correspond to two different implementations of the task. The transitions between the states `V1` and `V2` are controllable by the input `c1` and `c2`. In each of the versions states, equations give th evalues of variable that characterize them, either in terms of consumed resources (e.g., size in memory, response time, communication bandwidth), or in performance offered (e.g., levels of Quality of Service (QoS), precision in a numerical computation, depth of a search algorithm). In the example of Figure 8 shows, their are distinguished by a different value of WCET : that way, the controller can be designed with a model of expected response times, and switch versions if needed, to go to one expected to run faster.

Additional elements of modeling are the possibility to have a wait state if the start of the task can be delayed (e.g., in order for a resource to be freed), or several parallel spaces of versions w.r.t. different metrics.

Applications are composed of a set of tasks. It can be a structured set like a Directed Acyclic Graph (DAG) of dependecies, or a data-flow graph [1], [11], for which the semantics can be given by an automaton. In this paper we consider simple sets of tasks decided upon at each instant by the upper level mission manager.

3) Global model of the system: It is constructed from the previous modeling patterns, as a parallel composition of instances of all local behavioral models for each element, architectural or computational. Additional equations can define global values for costs and performances from local ones. This global transition system represents the full configurations space, before control is enforced.

C. Control objectives

Following the DCS approach, we define declarative objectives typically under the form of predicates to be made invariant by control. Classical ones involve simple exclusiveness properties of resources like the tiles. More interestingly, others involve more advanced logico-numerical or conditional aspects.

1) Logico-numerical objectives: A management strategy consists in switching to a version which has a lower (resp. greater) `wcet` than the previous version when the observed execution time is greater than a defined maximum threshold (`max_thres`) (resp. minimum threshold (`min_thres`)). In H/BZR, we declare the objectives as :

```
   ((time_t > max_thres) ⇒ ((0 fby wcet) > wcet))
and ((time_t < min_thres) ⇒ ((0 fby wcet) < wcet))
```

where `0 fby wcet` denotes the value `0` on the first instant, and the previous value of `wcet` on following ones. These two objectives allow to make a semantic link between the input `time_t`, which is measured from the environment, and the computed variable `wcet`. Hence, these objectives will make the controller try to increase or decrease the computation time, using the computed information on the global theoretical WCET.

These objectives are *logico-numerical* as they involve a numerical input (`time_t`) and a numerical state (defined by the **fby** operator), mixed with Boolean inputs and states (activation of tasks, states of devices). Such logico-numerical objectives can be handled by the ReaX synthesis tool.

2) Conditional objectives: Classically, supervisory control of DES provides for controllers which enforce the objectives at all times, if they exist. In practice, designing a complex system involves defining multiple partial objectives, between which there might be exceptions or interferences. Composing them can require solving the latter, for example with priorities, themselves changing over time according to external conditions.

The practical need is then to be able to write simple partial objectives, and to have ways to combine them in order to coordinate them into more complex ones. We want to be able to specify that a basic objective should be either enforced when possible, or replaced by another, degraded mode, with a variable notifying this explicitely e.g., to be sent to an upper decision layer.

We propose a methodology for declaring conditional objectives and priorities using controllable variables and implications in the invariance predicates. We exploit the fact that H/BZR produces determinized controllers that, at each step, assign correct values to the Boolean controllable variables by favoring **true** over **false**, with respect to their declaration order. We declare a conditional objective $\texttt{Obj}_c$, using a control variable $\texttt{c}_{pos}$, as follows : ($\texttt{c}_{pos} \Rightarrow \texttt{Obj}_c$). This will guarantee that $\texttt{c}_{pos}$ is **false** only if the right part cannot be enforced. In any state in which $\texttt{Obj}_c$ cannot be satisfied, the value **false** is assigned to the variable $\texttt{c}_{pos}$, otherwise the latter has the value **true** and the objective $\texttt{Obj}_c$ is enforced.

D. Modular design

The modular design of the model can be used to break down the synthesis time, which can be the bottleneck of this method on large systems. In our example, the control problem can be decomposed in two levels: at task level, and for each task, a local synthesized controller handles the objectives on WCET and bound execution time for this specific task; and at the main level, a global synthesized controller enforces the coordination of all tasks, specifically exclusive use of resources (like tiles) depending on some versions of these tasks. For this global controller to be successfully synthesized, the contracts at task level are equipped with controllability objectives, defining how inputs at task level can be used to control, e.g., the current executed version of the task.

V. Case study

In this section we apply our modeling and control approach on a concrete example. This system has been completely implemented on a FPGA board. We describe here the aspects directly related to the discrete control itself : other aspects, related to the hardware system design, are out of the scope of this paper, and available elsewhere [11]. The example is very simple, in order to facilitate the experiment, but it illustrates the essential features of our approach, and particularly the design of conditional and logico-numerical control objectives. We discuss all phases of the design, until execution on the FPGA, and including a variant on the design of the controller in order to obtain a better behaviour.

A. Control problem

We consider a FPGA-based embedded system on a UAV with a task called search-landing-area. It receives a flow of images from the camera and performs a sequence of transformations on each image in order to determine suitable areas for the UAV to land. It has two versions : software (`sw`) taking 1500 ms to process an image and hardware (`hw`) : 55 ms. The FPGA board has one CPU ARM and one Hardware processing resource (tile). The hardware version of the task is designed to run on it.

Depending on the urgency of landing, the version of the task might change. The control strategy consists of keeping the execution time of the task between a minimum and a maximum thresholds, which define the range of acceptable performance. When measured execution time is below the minimum threshold, we can reconfigure towards a version which uses less hardware resources, if its performance is inside the interval. This enables to allocate them to other tasks needing them most urgently.

B. Global control model

Figure 9 shows the global composition of the models (task, CPU and tile). It is associated with a behavioral

```
main(r, e, time_t, min_thres, max_thres, f, rp)
                = act_t, res, wcet, act, err, objective
assume true
enforce objective
with cp1, cp2, c1, c2, ct

  (act_t, res, wcet) = searchArea (r, c1, c2, e) ;
  (speed, ...) = cpu (c, c1, c2, c3) ;
  (act, err) = tile (ct, f, rp) ;
  objective = (
  (cp1 ⇒ ((time_t < min_thres) ⇒ ((0 fby wcet) < wcet)))
   and (cp2 ⇒ ((time_t > max_thres) ⇒ ((0 fby wcet) > wcet)))
   and (not (res = 2) ⇒ not act)  and (err ⇒ not (res = 2)) ) ;
```

Fig. 9: Global program : behaviors and contract

contract which contains the declaration of the management policies to enforce, defined by the Boolean variable `objective`, involving integer inputs and state.

Figure 10 shows a snapshot of the simulation of the synthesized controller using Sim2Chro from Verimag. We observe, at step 3, when the task is requested, that the manager triggers the SW version. At step 26, the measured execution time of the task becomes greater than the maximum threshold (`time_t > max_thres`). The manager switches to the HW version which is the fastest version and requires the tile.

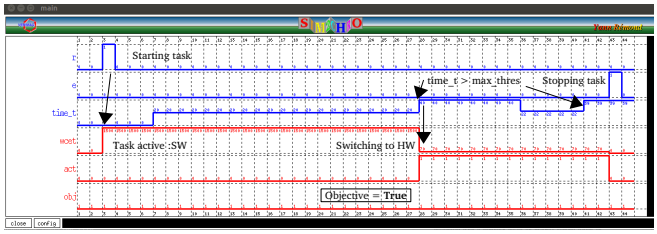


Fig. 10: Simulation for the conditional objectives

C. Compilation : Monolithic vs modular synthesis

In this section, we compare the monolithic DCS and the modular DCS. We consider the same FPGA-based embedded system with two tasks. Each task has two versions (`sw` and `hw`). The `hw` versions can not be active simultaneously because there is only one tile in the system. The control objectives consist in :

1) Maintaining the execution time of each task between a minimum threshold and a maximum threshold.
2) Enforcing mutual exclusion with respect to the tile

We compile the programs and synthesize the controllers on a machine with 4 GiB of RAM, and processor Intel Celeron(R) CPU N284 @ 2.1GHz x 2. For the monolithic model, the synthesis takes 15s; and decreases to 3s for the modular model. Both models includes 27 state variables.

D. Implementation on a FPGA platform

1) DE1-SoC FPGA: The board is based on a Altera/Intel Cyclone® V SoC chip which supports DPR. It includes a Hard Processor System (HPS) and a FPGA. The HPS comprises an ARM Cortex A9 dual-core processor, a DDR3 memory port, and a set of peripheral devices. The FPGA implements the reconfigurable tile (one in this experiment) and different peripheral controllers. We run a Linux OS on the HPS side. We implement a cma_driver module for the interaction with the hardware implementation of the task. The module allocates a continuous area of physical memory in the kernel space. We use the Robot Operating System (ROS) [12] as support for communication between manager and controlled task.

2) Executions: In the following executions, we first send to the manager a request to start the search landing area task. When the task is active, we change the value of the minimum and maximum thresholds to see which decisions the manager takes. Figure 11 shows an execution in which we set the value of the thresholds as follows. Initially the maximum threshold is set to 2000 ms while the minimum threshold is set to 0 ms. After 5 min of execution the maximum threshold is set to 70 ms. Later, it is set to 1000 ms and the minimum threshold is set to 100ms. After, 17 min the maximum threshold is set to 1800 ms then 400 ms (3 min later) and 20ms (5 min later). 5 min later we send request to stop the task.

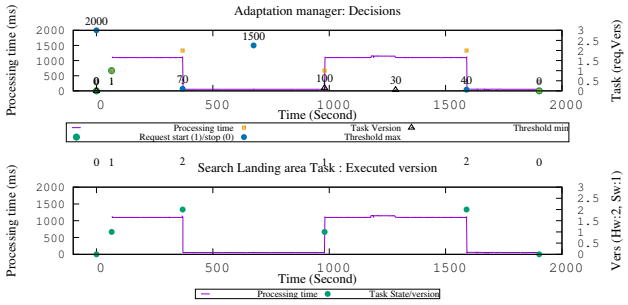


Fig. 11: Execution trace of Search-landing-area task

As shown in Figure 11, the manager dynamically adapts the executed version depending on the value of the minimum and maximum thresholds. When `max_thres` is set to 70 ms, when the manager observes that `(time_t > max_thres)` it switches the hardware version. When `min_thres` is set to 100 ms while max_thres is set to 1000 ms, it does nothing even if `(time_t < min_thres)`. But when `max_thres` is set to 1800 ms, the manager switches the software version. It returns to to hardare version when `max_thres = 400`. When `max_thres = 20` it maintains the hardware version even if the latter does not satisfy the objective.

E. A variant of the control to avoid oscillations

a) Oscillations: Setting the minimum and maximum thresholds to certain values can lead to oscillations. In Figure 12 the maximum threshold is initially set to 2000ms while the minimum threshold is set to 0ms. After *388 sec* the maximum threshold is set to 70ms. Later, after *693 sec* it is set to 1000ms instead of 1500ms. Finally it is set to 400ms after *1557 sec*. The minimum threshold is to 100ms after *998 sec* and later to 30ms.

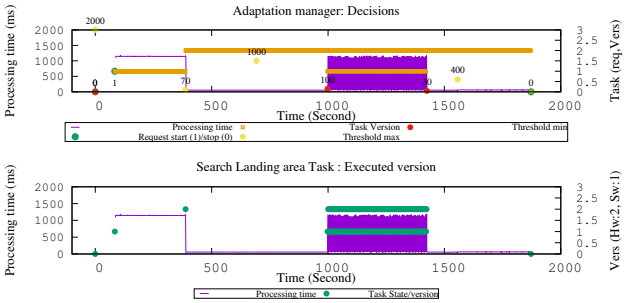


Fig. 12: Adaptation manager : log

As shown in Figure 12, when the minimum threshold is set to 100 (time 1000) while the maximum threshold is 1000, we observe oscillations. Indeed, the hardware version is faster than the minimum threshold, while the software version is slower than the maximum threshold. Hence the manager keeps switching between hardware and software in order to try to satisfy the policy.

b) Improving the objective to avoid oscillations: An additional policy prevents from selecting a version that has `wcet` greater than the maximum threshold :

`cp1` $\Rightarrow$ `(((0 fby wcet) > wcet)` $\Rightarrow$ `(wcet < max))`

So when the execution time is lower than the minimum threshold, only the subset of versions which have `wcet` lower than the maximum threshold will be candidate for replacing the active version.

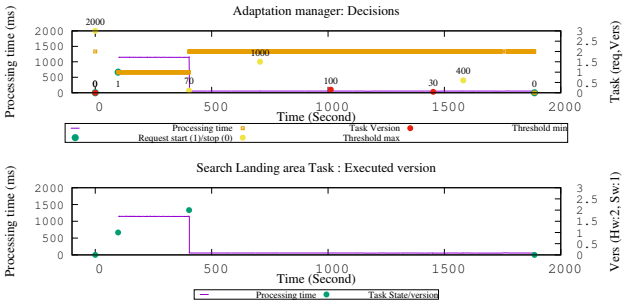


Fig. 13: Adaptation manager

As shown in Figures 13, with the additional policy the oscillations are prevented from occuring. The manager does not choose the software version since the latter has a `wcet` greater than the maximum threshold.

This example illustrates that in practice the precise formulation of objectives can be made in several successive improvements during a design.

VI. Conclusions and perspectives

We propose a generic modeling and control approach for Dynamically Partially Reconfigurable FPGA hardware architectures, based on Discrete Event Systems and their supervisory control. Particularly, we exploit logico-numerical control in order for controllers to take into account changing integer values of performance and quality requirements, conditional objectives for complex specifications, as well as modular DCS supported by Heptagon/BZR for scalability of design space exploration. We apply our approach in an implemented case study.

Perspectives are in making our generic models useable by specialists of the target systems and applications, rather than discrete control theory, under the form of a Domain Specific Language generating models automatically. Enriching the control approach can be done e.g., by considering adaptive discrete control, when objectives change due to the application or environment.

References

[1] X. An, E. Rutten, J.-P. Diguet, and A. Gamatié. Model-based design of correct controllers for dynamically reconfigurable architectures. *ACM Transactions on Embedded Computing Systems (TECS)*, 15(3), June 2016.

[2] N. Berthier, F. Alvares, H. Marchand, G. Delaval, and E. Rutten. Logico-numerical control for software components reconfiguration. In *2017 IEEE Conference on Control Technology and Applications (CCTA)*, pages 1599–1606, Aug 2017.

[3] N. Berthier and H. Marchand. Discrete Controller Synthesis for Infinite State Systems with ReaX. In *IEEE Int. Workshop on Discrete Event Systems*, Cachan, France, May 2014.

[4] E. Chen, V. G. Lesau, D. Sabaz, L. Shannon, and W. A. Gruver. Fpga framework for agent systems using dynamic partial reconfiguration. In *Proc. 5th Int. Conf. Industrial Applications of Holonic and Multi-agent Systems for Manufacturing*, HoloMAS'11, 2011.

[5] G. Delaval, S. M. Gueye, and E. Rutten. Distributed execution of modular discrete controllers for data center management. In *Proc. 5th IFAC workshop DCDS'15*, 2015.

[6] G. Delaval, E. Rutten, and H. Marchand. Integrating discrete controller synthesis into a reactive programming language compiler. *Discrete Event Dynamic Systems*, 23(4), Dec. 2013.

[7] F. Fons, M. Fons, E. Cantó, and M. López. Real-time embedded systems powered by fpga dynamic partial self-reconfiguration: A case study oriented to biometric recognition applications. *J. Real-Time Image Process.*, 8(3), Sept. 2013.

[8] N. Halbwachs. Synchronous programming of reactive systems, a tutorial and commented bibliography. In *Tenth Int. Conf. on Computer-Aided Verification, CAV'98*, June 1998.

[9] D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4), Oct 1996.

[10] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36:41–50, January 2003.

[11] S. Mak Karé Gueye, É. Rutten, and J.-P. Diguet. Autonomic management of missions and reconfigurations in FPGA-based embedded system. In *NASA/ESA Conf. on Adaptive Hardware and Systems (AHS)*, Pasadena, USA, July 2017.

[12] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. Ros: an open-source robot operating system. In *ICRA WS Open Source Software*, 2009.

[13] P. Ramadge and W. Wonham. On the supervisory control of discrete event systems. *Proc. IEEE*, 77(1), Jan. 1989.

[14] M. D. Santambrogio. From reconfigurable architectures to self-adaptive autonomic systems. In *Int. Conf. Computational Science and Engineering, CSE '09*, volume 2, Aug 2009.

[15] A. N. Sylla, M. Louvel, E. Rutten, and G. Delaval. Design framework for reliable multiple autonomic loops in smart environments. In *Int. Conf. Cloud and Autonomic Computing (ICCAC)*, 2017.