Model-based design of correct controllers for dynamically reconfigurable architectures

Xin An, Eric Rutten, Jean-Philippe Diguet, Abdoulaye Gamatié

Model-based design of correct controllers for dynamically reconfigurable architectures

Xin An[1], Eric Rutten[2], Jean-Philippe Diguet[3], and Abdoulaye Gamatié[4]

[1]Hefei University of Technology, Hefei, China
[2]INRIA, Grenoble, France
[3]CNRS/Lab-STICC, Lorient / Brest, France
[4]CNRS/LIRMM, Montpellier, France

Abstract

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. In this paper, we propose a tool-supported formal method to automatically design a correct-by-construction control of the reconfiguration. By representing system behaviors with automata, we exploit automated algorithms to synthesize controllers that safely enforce reconfiguration strategies formulated as properties to be satisfied by control. 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 two case studies implemented on FPGAs.

1 Introduction

Dynamically reconfigurable hardware has been identified as a promising solution for the design of energy efficient [17] embedded systems. A common argument in favour of this kind of architecture is the specialisation of processing elements, that can be adapted to application functions in order to minimize the delay, the control cost and to improve data locality. Another key benefit is the hardware

* This work is supported by the French ANR project Famous.

Authors' addresses: Xin An, Hefei University of Technology, China; e-mail: xin.an.fr@gmail.com. Eric Rutten, Ctrl-A team, INRIA Grenoble, France; e-mail: eric.rutten@inria.fr. Jean-Philippe Diguet, CNRS/Lab-STICC, Lorient, France; e-mail: jean-philippe.diguet@univ-ubs.fr. Abdoulaye Gamatié, CNRS/LIRMM, Montpellier, France; e-mail : abdoulaye.gamatie@lirmm.fr.

reuse to minimise the area, and therefore the static power and cost. Further advantages such as hardware updates in long-life products and self-healing [29] capabilities are also often mentioned. In presence of context changes (e.g. environment or application functionality), self-adaptive technique can be applied as a solution to fully benefit from the runtime reconfigurability of a system. Dynamic Partial Reconfiguration (DPR) of FPGA is another accessible solution to implement and experiment reconfigurable hardware. It has been widely explored and detailed in literature. However, it appears that such solutions are not extensively exploited in practice for two main reasons: i) the design effort is extremely high and strongly depends on the available chip and tool versions; and ii) the simulation process, which is already complex for non-reconfigurable systems, is prohibitively large for reconfigurable architectures. So, new adequate methods are required to fully exploit the potential of dynamically reconfigurable and self-adaptive architectures. Here, we are proposing a design methodology for self-adaptive embedded systems. On the one hand, our approach considers reconfigurable architectures as implementation of execution platforms by exploiting their features. On the other hand, for the design of adaptation decision, it relies upon a formal method related to automata-based verification, and more originally by considering discrete controller synthesis. It is important to note that in this paper, synthesis is not used in the sense of hardware synthesis. It is rather considered under the meaning of controller synthesis as a formal operation on automata as explained in the sequel.

1.1 Reconfigurable Architecture Validation Problem

The validation of a reconfigurable architecture includes on the one hand, the separate validation of each possible configuration of the architecture, and on the other hand, the validation of each transition between pairs of configurations, since the behaviour of the system can depend on the memory content and the I/Os status modified from a configuration to another. Current validation approaches are based on simulation. For reconfigurable systems, the number of scenarios to deal with rapidly becomes untractable, making simulation less efficient for addressing the design correctness issue.

Verification is central in electronic system level design in order to ensure that a system implements its functionality in a correct, efficient and cost-effective manner [24]. This is particularly true for FPGA-based design [25]. A very popular verification approach is simulation, which can be either cycle-based or event-based. Compared to the latter, the former provides an extremely good visibility into designs for debug. But, it is potentially more compute-intensive and slower. Another important approach is design testing, which is very useful for large and complex system verification. Its quality depends quite on the size and relevance of used test benches. The last relevant approach is formal verification, either static such as model-checking and theorem proving to formally verify given design properties, or dynamic by combining simulation and static formal analysis. Here, dynamic techniques consider assertions and check their possible violations during simulation.

Dynamical reconfiguration requires to take decisions about the choice of new configurations, depending on occurring events in a system, on past events and sequences history, and on predictive knowledge about possible outcomes of reconfigurations. Such decision components are difficult to design because of the combinatorics of possible choices, the transversal constraints between them to be respected, and even more, the history aspects. Formal approaches to the design of reconfiguration controllers can provide tool-supported assistance to this difficult design. In embedded system domain, formal techniques have been designed largely for safe software design. The evolution of DPR systems makes them amenable to the same kind of techniques and models. Labelled Transition Systems (LTS) or automata are typical effective specification models that can be verified by model-checking.

1.2 Objective and Contribution of this Paper

The present work aims to deal with the correct dynamical management of reconfigurations. It advocates the design of control loops addressed by Control Theory, which covers in general both continuous and discrete systems. Here, only the latter is considered. The class of Discrete Event Systems is modeled using Petri nets or automata. Based on notions of supervisory control, automated techniques have been defined for Discrete Controller Synthesis (DCS).

The contribution of this paper is to exploit these advantages of DCS for designing controllers to manage reconfigurable architectures with practical implementation on dynamically partially reconfigurable FPGAs. This enables us to answer the problems mentioned earlier in this paper by:

1) automatically generating the code of controllers to be implemented on a processor in charge of the reconfiguration management, considering that the reconfigurable architecture can be generated with recent model-based design flow [36],

2) adopting our approach where automated generation is guaranteed correct by the synthesis algorithm, instead of simulation for verification.

This paper relies on previous preliminary results [3] [4] and brings new improvements along the following directions: i) it extends the reconfiguration management methodology by describing the modeling concepts, including their generation procedure, at all system levels (architecture, application, etc.) for applying the DCS technique; ii) it draws a global design flow applied to two real and concretely developed case studies with run-time image processing and reconfiguration to demonstrate clearly how our advocated DCS technique can be applied; and iii) quantitative evaluations and analysis regarding the scalability of our advocated approach are reported.

Outline In the remainder of this paper, we first discuss some related work in Section 2. Then, we define the class of reconfigurable architecture that we target, as well as their reconfiguration policies in Section 3. We introduce the modeling and DCS concepts used through this paper in Section 4. These concepts are considered in Section 5 for building the general behavioral model of the

target class of architectures, and for formally specifying their associated control objectives. The resulting general design flow of our approach is presented in Section 6. Its validation is illustrated in Section 7 via some case studies involving concrete FPGA platform. Some discussion of the proposed solution regarding the obtained results is addressed in Section 8. Finally, concluding remarks and perspectives are provided in Section 9.

2 Related Work

In literature, most of the existing approaches dealing with the management of reconfigurable embedded systems target the run-time scheduling of application tasks onto a reconfigurable architecture (e.g., [26] and [12]) or an architecture including a reconfigurable fabric (e.g., [27]). Since these approaches perform scheduling analysis on-line, they thus usually resort to heuristic algorithms to generate fast and lightweight solutions and are able to deal with the scheduling of applications that are unknown a priori. However, such approaches cannot guarantee optimal solutions and/or strict system constraints due to unknown situations, and are usually validated by (limited) simulations.

Beyond usual simulation techniques [5] [14], formal methods provide attractive verification techniques that are applicable to reconfigurable embedded system designs. In [32], authors present a typical study addressing the correctness of reconfigurable cores, such as a 64-bit adder and a 8-bit counter. They consider a formalisation based on propositional logic and integer arithmetic. They use a theorem-prover at run-time to check whether the dynamically calculated circuits are correct. Their solution applies mainly to circuits that take seconds to verify. Other approaches [7] [20] suggest model-checking techniques for the verification of FPGAs and dynamically reconfigurable embedded systems in general. However, none of these solutions deals with the correct design of the reconfiguration control.

The reconfiguration management in DPR technologies is usually addressed by considering manual encoding and analysis, which is tedious and error-prone [13]. With the foreseeable increase in complexity of such technologies nowadays, automatic techniques appear more suitable in order to better solve the limitations related to the manual approach. Other existing approaches dedicated to self-management of adaptive or reconfigurable systems use for instance heuristics and machine learning techniques. In [33], a system built on a reconfigurable architecture exploits self-adaptivity. It adopts application heartbeats as monitoring framework, and a heuristic mechanism to switch between different configurations. Self-management in the form of self-healing that exploits FPGAs is also proposed in other studies [29] [18].

Regarding design infrastructures, an architectural proposal [22] provides a slot-based organization of a reconfigurable hardware as well as an elaborate communication framework with good reconfiguration support. In [37], an adaptive system is implemented on FPGA by means of a programming model and environment for the development of reconfigurable multiprocessor architectures.

Beyond the previous aspects, reconfiguration control is one major issue for adequate system behaviors. In [21], authors discuss some approaches applying standard control techniques such as Proportional Integral and Derivative (PID) controller or Petri nets-based control. The same kind of control has been also used for processor and bandwidth allocation in servers [19]. A close-loop control has been applied in [10] to select hardware/software configurations on an FPGA with a configuration control based on a data-flow model and diffusion mechanisms. We note that such a solution relies on heuristics and empirical laws that prevent instability and select the suitable configurations. In [30], a design flow is proposed from high level models to automatic code generation, for the implementation of reconfigurable FPGA based systems-on-chip. The system control is modeled manually and integrated into the flow.

Compared to the above reconfiguration control techniques, a major advantage of the discrete control approach considered in this paper is the enabled formal correctness. In addition, the advocated controllers are generated automatically at design time. To the best of our knowledge, there is no existing work addressing the automatic generation of correct reconfiguration controllers, starting from design to implementation, for FPGA systems. An earlier work [15] explored only simple models for tasks, and invariance control objectives which concern state exclusions. Here, we have a extended model structure with application, tasks and architecture, and we use reachability and optimal control beyond invariance. Optimal control concerns the optimization of costs or weights associated with states and/or transitions. By enforcing optimal control, e.g., minimizing the system Worst Case Execution Time (WCET), our approach is thus also able to guarantee performance.

3 DPR Control Problem

We present informally the considered class of systems with an illustrative example.

3.1 Hardware Architecture Model

We consider a multiprocessor architecture implemented on a reconfigurable device which is composed of a general purpose processor $A0$ (e.g., ARM core), and a reconfigurable area (e.g. FPGA-like with power management capabilites) divided into n reconfigurable tiles. Figure 1 a) shows an illustrative example of four reconfigurable 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). A fixed dual port memory buffer is associated with each tile, which means that at most two tasks can simultaneously access data stored in the shared memory. Reconfigurable tiles can be combined and configured to implement and execute tasks by loading predefined bitstreams, such as tiles A1 and A2 of Figure 1 a).

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.

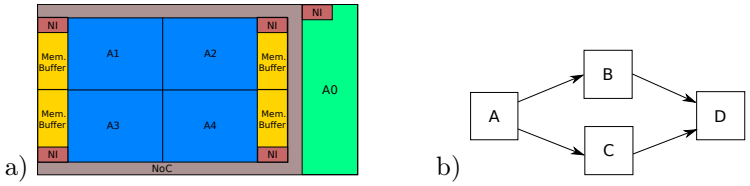

Figure 1: a) architecture structure, b) a DAG application specification.

3.2 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. Note that the chosen graph-based representation is seen as a generic representation that enables to describe a large number of applications. It is also a useful abstraction level for dealing with the safe control of tasks by using formal techniques. The coarse grain tasks considered at this abstraction level avoid the burden of associated unnecessary low level details for defining a suitable solution to the problem. There exist a number of works that show how such a task graph can be derived from an initial application specification described in programing languages such as C [35] or some high-level specification languages such as UML MARTE [16]. Dealing with such transformations is out of the scope of this paper. Figure 1 b) shows an illustrative example consisting of four tasks: A, B, C and D.

In our framework, unless otherwise specified, 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 uncontrollable. The way of *delaying* and *executing* tasks is controlled by a runtime manager designed to achieve system objectives.

3.3 Task Implementations

Given a hardware architecture, a task can be implemented in various ways characterised by various parameters of interest, such as the set of used reconfigurable tiles (rs), worst case execution time (WCET) (wt), reconfiguration time (rt), and power peak pp. For instance, task A may have the two following implementations:
▷Implementation 1: $rs_1 = \{A1\}$, $wt_1 = 190$, $rt_1 = 10$, $pp_1 = 180$;
▷Implementation 2: $rs_2 = \{A3, A4\}$, $wt_2 = 85$, $rt_2 = 15$, $pp_2 = 250$;

Table 1 gives the implementations and profiled characteristics of tasks A, B, C, D. Among the possible task implementations, a run-time manager is in charge of choosing the suitable implementations at run-time according to system objectives.

Table 1: Profiled task implementation characteristics for the working example.

	Implementations (tiles set, WCET, reconfig. time, power peak)		
Tasks	Implementation 1	Implementation 2	Implementation 3
A	$(\{A1\}, 190, 10, 180)$	$(\{A3, A4\}, 85, 15, 250)$	-
B	$(\{A2\}, 430, 20, 120)$	$(\{A1, A2\}, 275, 25, 160)$	$(\{A1, A2, A3, A4\}, 120, 30, 400)$ -
C	$(\{A3\}, 225, 15, 100)$	$(\{A3, A4\}, 80, 20, 250)$	-
D	$(\{A1\}, 238, 12, 200)$	$(\{A1, A2\}, 85, 15, 350)$	$(\{A1, A2, A3, A4\}, 50, 20, 450)$

3.4 System Reconfiguration

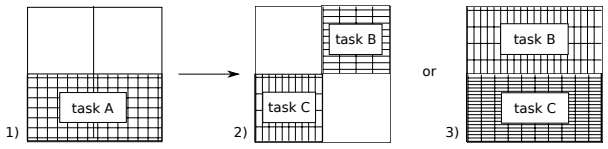

Figure 2: Configurations and reconfigurations.

Figure 2 shows three system configuration examples. In configuration 1, task A is running on tiles A3 and A4 while tiles A1 and A2 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 1 b), 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 circuit surface and therefore consumes more power. On the contrary, when the battery level is high, configuration 3 would be chosen if the user expects a better performance.

3.5 System Objectives

System objectives define the system functional and non-functional requirements. This section gives the objectives considered in this work paper for a general dynamically reconfigurable architecture with power management capabilities. It categorises them as logical and optimal control objectives. Generally speaking, logical objectives concern exclusions, whereas optimal objectives concern weights and costs.

Considered logical control objectives are as follows:

1. resource usage constraint: exclusive uses of reconfigurable tiles $A1$-$A4$;
2. dual accesses to the shared memory: by at most two functions running in parallel;
3. energy reduction constraint: switch tiles to
   (a) sleep mode when executing no task;
   (b) active mode when needed;
4. reachability: DAG execution can always finish once started;
5. power peak of hardware platform is constrained w.r.t battery levels;

Optimal control objectives of interest are as follows:

6. minimise power peak of hardware platform;
7. minimise WCET of DAG executions;
8. minimise worst case energy consumption of system executions.

These objectives will be formalized further in Section 5 in terms of the formalisms recalls next in Section 4. Some of them will be used and validated in Section 7.

4 Modeling Formalism and DCS

We apply the formal technique of DCS to address the control problem of a dynamically reconfigurable architecture. Before presenting DCS, we firstly introduce the automata based synchronous modeling formalism, which is adopted in the paper to describe the control problem. We adopt such a modeling formalism because 1) automata based modeling formalisms are quite natural to model system reconfiguration behaviors, 2) the mathematical foundation of the synchronous model favors system formal analysis, and 3) there exist DCS tools that can exploit synchronous parallel automata.

4.1 Modeling Formalism

We adopt the formal framework defined in details elsewhere [1] [9] for the automata definition.

Definition 1 (Automaton) *An automaton is a tuple* $S =< \mathcal{Q}, q_0, \mathcal{I}, \mathcal{O}, \mathcal{T} >$*:*

- $\mathcal{Q}$ *is a finite set of states;*
- $q_0 \in \mathcal{Q}$ *is the initial state of* S*;*
- $\mathcal{I}$ *is a finite set of input events;*
- $\mathcal{O}$ *is a finite set of output events;*
- $\mathcal{T}$ *is the transition relation that is a subset of* $\mathcal{Q} \times Bool(\mathcal{I}) \times \mathcal{O}^* \times \mathcal{Q}$*, such that* $Bool(\mathcal{I})$ *is the set of Boolean expressions of* $\mathcal{I}$ *and* $\mathcal{O}^*$ *is the power set of* $\mathcal{O}$*.*

Each transition, denoted by $q \xrightarrow{g/a} q'$, has a *label* of the form g/a, where *guard* $g \in Bool(\mathcal{I})$ must be true for the transition to be taken, and *action* $a \in \mathcal{O}^*$ is a conjunction of output events, emitted when the transition is taken. State q is the *source* of the transition, and state q' is the *destination*. A *path* is a sequence of transitions denoted by $p = q_i \xrightarrow{g_i/a_i} q_{i+1} \xrightarrow{g_{i+1}/a_{i+1}} ... \xrightarrow{g_{i+k-1}/a_{i+k-1}} q_{i+k}$, where $\forall j, i \leq j \leq i+k-1, \exists (q_j, g_j, a_j, q_{j+1}) \in \mathcal{T}$.

The composition of two automata put in parallel is the *synchronous composition*, denoted by $||$. Given two automata $S_i =< \mathcal{Q}_i, q_{i,0}, \mathcal{I}_i, \mathcal{O}_i, \mathcal{T}_i >, i = 1, 2$, with $\mathcal{Q}_1 \cap \mathcal{Q}_2 = \emptyset$, their *composition* is defined as follows: $S_1||S_2 =< \mathcal{Q}_1 \times \mathcal{Q}_2, (q_{1,0}, q_{2,0}), \mathcal{I}_1 \cup \mathcal{I}_2, \mathcal{O}_1 \cup \mathcal{O}_2, \mathcal{T} >$, where $\mathcal{T} = \{(q_1, q_2) \xrightarrow{g_1 \wedge g_2 / a_1 \wedge a_2} (q_1', q_2') | q_1 \xrightarrow{g_1/a_1} q_1' \in \mathcal{T}_1, q_2 \xrightarrow{g_2/a_2} q_2' \in \mathcal{T}_2, g_1 \wedge g_2 \wedge a_1 \wedge a_2\}$. Composed state (q_1, q_2) is called a *macro state*, where q_1 and q_2 are its two *component states*.

The *encapsulation* operation, defined in [1], is used to enforce the synchronization between two composed automata by means of a variable which is an input on one side, and an output on the other side. Let $S =< \mathcal{Q}, q_0, \mathcal{I}, \mathcal{O}, \mathcal{T} >$ be an automaton, and $\Gamma \subseteq \mathcal{I} \cup \mathcal{O}$ be a set of inputs and outputs of S. The *encapsulation* of S w.r.t. Γ is the automaton $S \backslash \Gamma =< \mathcal{Q}, q_0, \mathcal{I} \backslash \Gamma, \mathcal{O} \backslash \Gamma, \mathcal{T}' >$ where $\mathcal{T}'$ is defined by $(q \xrightarrow{g/a} q' \in \mathcal{T}) \bigwedge (g^+ \bigcap \Gamma \subseteq \mathcal{O}) \bigwedge (g^- \bigcap \Gamma \bigcap \mathcal{O} = \emptyset) \Leftrightarrow (q, \exists \Gamma . g, \mathcal{O} \backslash \Gamma, q') \in \mathcal{T}'$. g^+ is the set of variables that appear as positive elements in the monomial g, i.e., $g^+ = \{x \in g | (x \bigwedge g) = g\}$. g^- is the set of variables that appear as negative elements in the monomial g, i.e., $g^- = \{x \in g | \neg (x \bigwedge g) = g\}$. Figure 4.1 gives an example of using encapsulation to enforce the synchronization of two automata A and B that are composed by a synchronous composition through variable b.

The automata states can be associated with *weights*, characterising quantitative features. We define a cost function $C : \mathcal{Q} \rightarrow N$ to map each state of an LTS to a positive integer value. Costs can also be defined on execution paths across an LTS. For instance, a cost function of path p can be the sum of all the

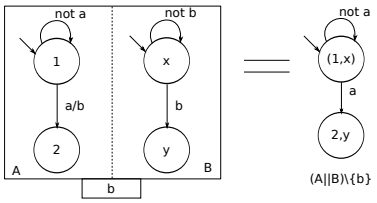

Figure 3: An example using encapsulation to enforce the synchronization of two composed automata.

costs of its traversed states. When composing LTS's, the cost values w.r.t. the resulting global states/transitions can be defined on the basis of the local costs as their sum or the maximal/minimal value.

Based on the above definition of automata, and other automata-based modeling formalisms presented in this section, formal analysis and verification techniques, such as model checking, discrete controller synthesis, can be applied. In this work, we adopt the *discrete controller synthesis* technique, which is presented in the next section.

4.2 Discrete Controller Synthesis

DCS, introduced in 1980s [31], was proposed to deal with the control and coordination problems of discrete event systems. A *discrete event system* (DES) [31] is a discrete-state, event-driven dynamic system that evolves in accordance with the occurrences of discrete events at possibly irregular intervals. An event, for example, may correspond to the invoke or completion of a task, the failure or frequency switch of a processor. Such systems arise in various domains of our daily life, such as manufacturing, transport, automotive, embedded systems, healthcare. These applications have their own design requirements, and require control and coordination to ensure their desired behavior.

The main advantage of the theory is that it separates the concept of open loop dynamics (i.e., the DES) from feedback control, and allows the autonomic analysis and control of DESs w.r.t. a given specification of control objectives.

DCS is an operation that applies on a DES presented as e.g., an automaton as defined in Section 4.1. In order to control a DES, i.e., enable a controller to influence the evolution of the DES behavior, the occurrences of certain events are under control. The set of events XY of a DES is thus partitioned into two subsets: Y_{uc} and Y_c, representing respectively the *uncontrollable* and *controllable* event sets. Figure 4 shows the principle of DCS. It is applied with a given control *objective*: a property that has to be enforced by control. The objective is expressed in terms of the system's outputs X. The controller denoted by C is obtained automatically from a system model S and an objective, both specified by a user, via appropriate synthesis algorithms. The synthesis algorithms, which are related to model checking techniques, automatically compute, by exploring the system state space, a constraint on controllable variables Y_c, i.e., the

controller. Its purpose is to constrain the values of controllable variables Y_c, in function of outputs X and uncontrollable inputs Y_{uc}, such that all remaining behaviors satisfy the given objective.

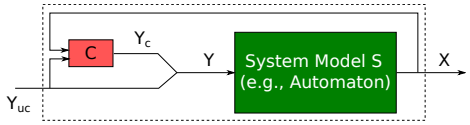

Figure 4: Principle of discrete controller synthesis

There can be several controllers that meet the same control objective. In the extreme case, a controller can forbid any state transition in order to avoid the invalid states. This is apparently not desirable for target systems. We are interested in *maximally permissive* controllers, which ensure the largest possible set of correct behaviors of the original uncontrolled system.

More generally, advantages of DCS are: high-level specification with declarative control policies; automated synthesis of correct-by-design/construction controllers; and optimality in the sense of maximal permissivity (minimally constraining controller).

In this work, we use DCS operations corresponding to different algorithms to synthesize controllers [23] for: invariance w.r.t. a subset of states, reachability of a subset of states, one-step optimization of a cost on the next state, and path optimization of a cost of the bounded path to a target state [9]. Such operations have been implemented in the Sigali DCS tool [23]. In particular, we use the user-friendly tool BZR* [8], whose compilation involves the Sigali tool to perform DCS. It employs the automata modeling formalism of Section 4.1 to describe the target system behavior, and a dedicated construct *contract* to specify the control objectives to be enforced in a declarative style. The compilation of BZR will automatically synthesize a controller enforcing the specified objectives. This controller is then re-injected automatically into the initial BZR program so that an executable program can be generated (in C or Java) for execution. This executable code is used in the experiments described in Section 7.

As for other formal verification techniques such as model-checking, where complexity is in the worst case polynomial in the size of the state space to handle, DCS is also concerned by the scalability issue (which will be discussed in Section 8). However, compared to them, the main advantage of the DCS is that it is more constructive and is able to produce a maximally permissive and correct solution, while other formal techniques such as model-checking require a possibly error-prone and over-constraining manual encoding phase before a tedious verification phase [11].

*`http://bzr.inria.fr`

5 Modeling Reconfiguration Management Computation 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 execution of the resulting controller.

5.1 Architecture Behaviour

The architecture consists of a processor $A0$, and n reconfigurable tiles $\{A1,..., An\}$ and a battery (see Figure 1 a), where $n = 4$). 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.

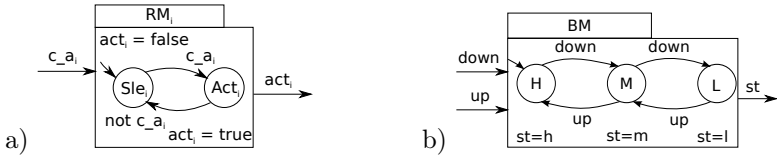

Figure 5: Models RM_i for tile Ai, and BM for battery.

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*.

It can be observed that the architecture behavior including the behaviors of reconfigurable tiles and battery can be described systematically, and could be specified with some high level specification languages given some syntactic sugar. As an example, a systematic way to generate such automata of Figure 5 from the UML profile MARTE [28] can be found in [16].

5.2 Application Behaviour

The software application is described as a DAG, which specifies the tasks to be executed and their execution sequences and parallelism. We capture its behaviour by defining a *scheduler automaton* representing all possible execution scenarios. It does so by keeping tracking of application execution states and emitting the *start* requests of tasks in reaction to the task *finish notifications*.

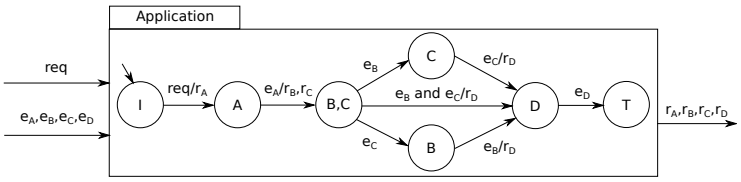

Figure 6: Automaton capturing application DAG execution behaviours.

5.2.1 Informal Description

Figure 6 shows the scheduler automaton of the application DAG in Figure 1 b). It starts the execution of the application by emitting event r_A, which requests the *start* of task A, upon receipt of application request event *req* in the idle state I. Upon receipt of e_A notifying the finish or *end* of A's execution, events r_B and r_C are emitted together to request the execution of tasks B and C in parallel. Task D is not requested until the execution of both B and C is finished, denoted by events e_B and e_C. It reaches the final state T, implying the end of the application DAG execution, upon receipt of e_D.

In fact, Figure 6 models the execution behavior of an iteration of the example task graph. Our approach can also deal with pipelined executions of streaming applications described by data-flow models such as Synchronous Data Flow Graphs (SDFGs). For SDFGs, we can build execution models similarly by: 1) identifying events that fire task or actor computations when their input tokens are ready, and those that notify ends of computations with corresponding output tokens produced, and 2) representing all the relevant states on which control should be based.

Such a scheduler automaton can be constructed algorithmically from a DAG described application. In the following section, we describe systematically how to obtain such a scheduler automaton from a DAG described application.

5.2.2 Scheduler Automaton Derivation

As shown in the example of Figure 6, the derived scheduler automaton from an application DAG captures the dynamic execution behavior of the application. Its states represent the tasks that are being executed. They are denoted and labeled by the names of these tasks. It has an initial state I, i.e., the idle state, which means the application has not been invoked, and an end state T, which means that the application has finished its execution. The automaton input events are the task end events $e_1, e_2, ..., e_i, ...$ and the application request event *req*, while its output events are the task request events $r_1, r_2, ..., r_i, ...$ Its transitions are of the form g/a, where g is a firing condition, and a is an action. A firing condition is a boolean expression of input events, and an action is a conjunction of output events. *Note*: 1) we suppose the application is only invoked once. If it is allowed to be repeatedly invoked, the end state would be

the same to the initial state. 2) if the graph has a task that has more than one instance, the instances are then seen as different tasks.

Algorithm 1 illustrates how to construct the scheduler automaton for a DAG. It derives the automaton from initial state I to end state T by exploring the state space of the application execution w.r.t. the DAG.

- *Inputs:* a directed, acyclic task graph $< \mathbb{T}, \mathbb{C} >$, where $\mathbb{T}$ and $\mathbb{C}$ represent respectively the set of tasks and the set of edges.
- *Local variables and functions* used in the algorithm:
  - s or $nextState$: a state, with element $taskSet$ representing the set of tasks *associated* to the state (i.e., the tasks executing in the state);
  - drawState(s): a function that draws state s, labeled by $s.taskSet$;
  - drawTrans($source, sink, transition\ label$): a function that draws a transition from state $source$ to state $sink$ guarded by $transition\ label$;
  - $drawnStates$: the set of states that have been drawn out;
  - $stateQueue$: a FIFO queue, keeping track of the states to be processed, with function $popup()$ to return and delete the first state element, and function $add(s)$ to add state s to the end of the queue;
  - $t_i.prec$: the set of tasks that immediately precede task t_i;
  - $readyTaskSet$: the set of tasks that are enabled to execute;
  - tc: a set of tasks, or a task combination;
  - powerSet($a\ set\ of\ tasks$) returns the power set of the $set\ of\ tasks$ without $\emptyset$;
  - traversed(s) returns the set of states traversed by some path from state I to state s (states I and s included) w.r.t. the current drawn automaton, with element $taskSet$ to return the union of the tasks associated with all the states of the set.

Lines 1 to 8 deal with **Phase 1**, i.e., the drawing of the initial state I and the initialization of local variables. At line 1, the initial state, i.e., idle state I is drawn denoted by $drawState(I)$. The set of drawn states $drawnStates$ is thus initialized to $\{I\}$ at line 2. State queue $stateQueue$ stores the states that have been drawn but not processed. It is initialized to have element I at line 3. Variable $readyTaskSet$ represents the set of tasks that are *enabled* to execute once some event happens. A task is *enabled* if all its precedent tasks have finished their executions. Lines 4 to 8 set $readyTaskSet$ to the set of tasks that have no precedent tasks, as such tasks can be executed immediately once the application is invoked/requested denoted by the receipt of event req.

Lines 9 to 44 deal with **Phase 2**, i.e., the sequential processing of the states stored in $stateQueue$. The processing of a state concerns drawing its immediate following states and the corresponding transitions, and adding new drawn states in the queue. The automaton derivation process finishes when the queue becomes empty. Three types of states are distinguished and processed accordingly.

ALGORITHM 1: Scheduler Automaton Derivation

```
   // Phase 1: initialization.
 1 drawState(I);
 2 drawnStates = {I};
 3 stateQueue = stateQueue.add(I);
 4 forall the t_i ∈ 𝕋 do
 5     if t_i.prec = ∅ then
 6         readyTaskSet = readyTaskSet ⋃ t_i;
 7     end
 8 end
   // Phase 2: processing of the states in stateQueue.
 9 while stateQueue! = ∅ do
10     s = stateQueue.popup();
       // Case 1: s be the initial state.
11     if s = I then
12         nextState.taskSet = readyTaskSet;
13         drawState(nextState);
14         drawTrans(I, nextState, req/r_readyTaskSet);
15         drawnStates = drawnStates ⋃ nextState;
16         stateQueue.add(nextState);
17     end
       // Case 2: s be the end state.
18     else if s = T then
19         continue;
20     end
       // Case 3: s be neither the initial state or the end state.
21     else
22         forall the tc ∈ powerSet(s.taskSet) do
23             readyTaskSet = ∅;
24             forall the t_i ∈ 𝕋− traversed(s).taskSet do
25                 if t_i.prec ⊆ (traversed(s).taskSet - s.taskSet) ⋃ tc then
26                     readyTaskSet = readyTaskSet ⋃ t_i;
27                 end
28             end
29             nextState.taskSet = readyTaskSet ⋃ (s.taskSet - tc);
30             if nextState.taskSet = ∅ then
31                 nextState = T;
32             end
33             if nextState ∈ drawnStates then
34                 drawTrans(s, nextState, e_tc);
35             end
36             else
37                 drawState(nextState);
38                 drawTrans(s, nextState, e_tc);
39                 drawnStates = drawnStates ⋃ nextState;
40                 stateQueue.add(nextState);
41             end
42         end
43     end
44 end
```

They are initial state I, end state T and the rest. Due to space limitation, their detailed explanations are omitted here. We refer the readers to Section 5.3 of [2] for more details.

5.3 Task Execution Behaviour

Before executing a task on a reconfigurable architecture, the task implementation (i.e. a bitstream in case of FPGA) should be loaded to reconfigure the corresponding tiles if required. The reconfiguration operations inevitably involve some overheads regarding e.g., time and energy. The worst case that can be imagined is that a reconfiguration operation is always required before a task is executed. In this case, reconfiguration operation and task execution can be treated as a whole. In general, however, whether a reconfiguration operation is required or not before a task is executed depends on the run-time situation, i.e., whether the corresponding task implementation is already configured.In this case, the reconfiguration operation and task execution are independent, and should be distinguished and treated accordingly. In the following, we describe the modeling in consideration of the first case (i.e., the worst case), and refer the readers to [4] for the modeling of the second case.

In the worst case, a task implementation is always loaded before being executed. These two consecutive operations are thus combined and treated as one executing operation. In consideration of the four control points of task executions (see Section 3.2), the execution behaviour of task A associated with two implementations (see Section 3.3) can be modelled as Figure 7. 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 choice points: being delayed and executed. From initial state I_A, upon the receipt of *start* request r_A, task A goes to either:

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

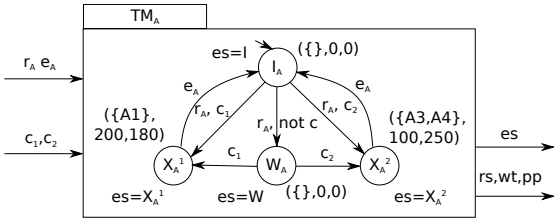

Figure 7: Execution behavior model TM_A of task A considering the worst case.

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 finish or end notification event e_A is received, the automaton goes back to *idle* state I_A. Output es represents its execution state. Similar to the architecture behavior in Section 5.1, the task execution behavior can also be described systematically, and be specified with some high level specification languages. Therefore, the automaton model for task execution can also be generated systematically from some high level specification languages as in [16].

Local execution costs. The reconfiguration and execution costs of different task implementations are different. As task reconfiguration operation and execution are combined as a whole, their costs thus need to be combined as well. Therefore, three cost parameters are considered here (see Section 3.3). 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), $wt \in \mathbb{N}$ (the sum of a reconfiguration time value and a WCET value) and $pp \in \mathbb{N}$ (a power peak). The costs associated with *executing* states are the values associated with their corresponding implementations. For *idle* and *wait* states, $rs = \emptyset, wt = 0, pp = 0$. Figure 7 gives the complete model of task A.

5.4 Global System Behaviour Model

5.4.1 *Global behavior.*

The parallel composition of the control models for reconfigurable tiles RM_1-RM_4, battery BM and tasks TM_A-TM_D, plus scheduler Sdl comprises the system model:

$$\mathcal{S} = RM_1||...||RM_4||BM||TM_A||...||TM_D||Sdl$$

with initial state $q_0 = (Sle_1, ..., Sle_4, H, I_A, ..., I_D, I)$. It represents all the possible system execution behaviours in the absence of control (i.e., a run-time manager is not yet integrated). Each execution behaviour corresponds to a *complete path*, which starts from initial state q_0 and reaches one of the *final states*:

$$Q_f = (q(RM_1), ..., q(RM_4), q(BM), I_A, ..., I_D, T),$$

where $q(Id)$ denotes an arbitrary state of automaton Id.

5.4.2 *Global costs.*

The costs defined locally in each task execution model need to be combined into global costs. A system state q is a composition of local states (denoted by $q_1, ..., q_n$), and we define its cost from the local ones as follows:

- used resources: union of values for local states: $rs(q) = \bigcup rs(q_i), 1 \leq i \leq n$;

- worst case execution time: this indicates how much time the system takes at most in this current state. It is thus defined as the minimal WCET of all executing tasks in this state, i.e., $wt(q) = min(wt(q_i), wt(q_i) \neq 0, 1 \leq i \leq n)$; Otherwise, if no task is executing in the state, i.e., $\forall 1 \leq i \leq n, wt(q_i) = 0$, $wt(q) = 0$;
- power peak: sum of values for local states, i.e., $pp(q) = \sum(pp(q_i), 1 \leq i \leq n)$;
- worst case energy consumption: the product of the worst case execution time and power peak of the system state, i.e., $we(q) = pp(q) * wt(q)$.

Now, we need to define costs associated with paths so as to capture the characteristics of system execution behaviours. Given path $p = q_i \rightarrow q_{i+1} \rightarrow ... \rightarrow q_{i+k}$, and costs associated with system states, we define costs on path p as follows:

- WCET: sum of WCETs on states on the path, i.e., $wt(p) = \sum wt(q_j), i \leq j \leq i + k$;
- power peak: maximum on states along the path: $pp(p) = max(pp(q_j), i \leq j \leq i + k)$;
- worst case energy consumption: the sum of the worst case energy consumptions on the states along the path, i.e., $we(p) = \sum we(q_j), i \leq j \leq i + k$.

5.5 System Objectives

Based on the formal model above, we formalize the reconfiguration policies of Section 3.5. The two types of system objectives: logical and optimal, are described in terms of the states and the costs defined on the states or paths of the model.

Logical control objectives. For any system state q, we want to enforce the following:

(1) exclusive uses of reconfigurable tiles by tasks: $\forall q_i, q_j \in q, i \neq j, rs(q_i) \bigcap rs(q_j) = \emptyset$;

(2) dual accesses to shared memory, i.e., at most two tasks access at the same time:
$\sum v_i \leq 2$, s.t. $v_i = \begin{cases} 1 & q_i \in X_i \\ 0 & \text{otherwise} \end{cases}$, where X_i represents the set of executing states of corresponding task;

(3.a) switch tile Ai to sleep when executing no task: $\nexists q_j \in q, Ai \in rs(q_j) \Rightarrow act_i = false$;

(3.b) switch tile Ai to active when executing task(s): $\exists q_j \in q, Ai \in rs(q_j) \Rightarrow act_i = true$;

(4) reachability: Q_f is always reachable.

(5) battery-level constrained power peak (given threshold values P_0, P_1, P_2): $pp(q) < P_0$ (resp. P_1 and P_2) when battery level is high (resp. medium and low).

Optimal control objectives. They can be classified into two types: one-step optimal and optimal control on path objectives. We use pseudo functions max and min in the following to represent the maximisation and minimisation objectives, respectively.

One-step optimal objectives. One-step optimal objectives aim to minimise or maximise costs associated with states and/or transitions in a single step [23]. Objective 6 of Section 3.5 belongs to this type.

(6) minimise power peak pp in the next states of state q: $min(pp, q)$.

Optimal control on path objectives. They aim to drive the system from the current state to the target states Q_f at the best cost [9], as in 7 and 8.

(7) minimise remaining WCET wt from state q: $min(wt, q, Q_f)$;

(8) minimise remaining energy consumption we from q: $min(we, q, Q_f)$.

These models can be encoded in BZR to generate automatically a controller satisfying the defined system objectives. Besides, the BZR compiler also allows the designers to simulate their designed models, which would be shown in Section 6.1. Implementing such models in two real cases studies would be presented in Section 7.

6 Design Flow

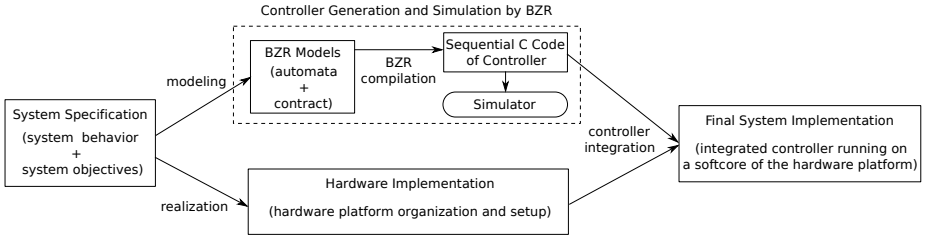

Figure 8: Our design flow.

In this section, we present our design flow for self-adaptive embedded systems from their *system specifications* towards *final system implementations* on reconfigurable architectures (see Figure 8), where the upper branch, i.e., *controller generation and simulation by BZR*, deals with the system reconfiguration management. It models the system reconfiguration control problem by *BZR models*, and performs the BZR compilation to derive automatically a controller in C

code. Along with the generated C code of controller, BZR also generates some other executable C codes to allow the users to use an associated simulator to perform simulations. The other branch deals with the *hardware implementation*, which selects and organises the hardware components according to system specification so as to realize the system functionality. The *final system implementation* is derived by integrating the generated controller on the hardware implementation, i.e., by the *controller integration* process. To be more specific, the final implementation implements the generated controller as a software task running on a soft core (i.e., $A0$) of the hardware implementation.

In the paper, we have focused on the modeling of the reconfiguration control problem (as illustrated in Section 3) by BZR-style models (as in Section 5). In the rest of this Section, we describe briefly the *controller generation and simulation* which includes the description of the generated *C code of controller*, *controller integration* and a typical experimental setup which describes a typical *hardware implementation*.

6.1 Controller Generation and Simulation

As shown in Figure 8, by feeding *BZR models* to the BZR compiler, it produces a controller (in C code) satisfying the defined system objectives. This code is synthesised in another executable C code, which can be compiled for the embedded processor on the target hardware architecture. This C code is structured of two functions: a **reset** function sys_reset to initialize system state variables, and a **step** function sys_step, which performs system state transitions according to the values of system uncontrollable inputs and states, and the computed values of controllable variables. Two additional C files named `main.c` and `main.h` are also generated by the compiler for simulation purpose. All these generated C codes can be fed to the graphical display tool sim2chro (from the *Verimag* research center[†]) associated with BZR to perform simulations of the controlled system. This enables the designers to validate and adjust their designs at the early stage before going to *final system implementation*.

Figure 9 shows a simulation scenario of the models in Section 5 for which the 5 logical control objectives of Section 5.5 are illustrated (see Table 1 for the implementation characteristics of the tasks). At instant 3, as labeled 1 in the figure, variables a_onA3 and a_onA4 become true, which implies that the second implementation of A which uses tiles $A3$ and $A4$ is chosen by the manager. At the same instant, tiles $A3$ and $A4$ are switched to the active mode, i.e., $act3, act4$ become true, which corresponds to objective 3.b). At instant 9, as shown by label 2 in the figure, task C finishes its execution by releasing tiles $A3$ and $A4$, i.e., c_onA3 and c_onA4 become false. At the same instant, tiles $A3$ and $A4$ are switched to the sleep mode, i.e., $act3, act4$ become false. This corresponds to objective 3.a). As shown in label 3, the system power peak pp is always less than 300, even though battery level is high. This is because that, firstly, the tasks cannot change its implementation once executed, and

[†] `http://www-verimag.imag.fr/`

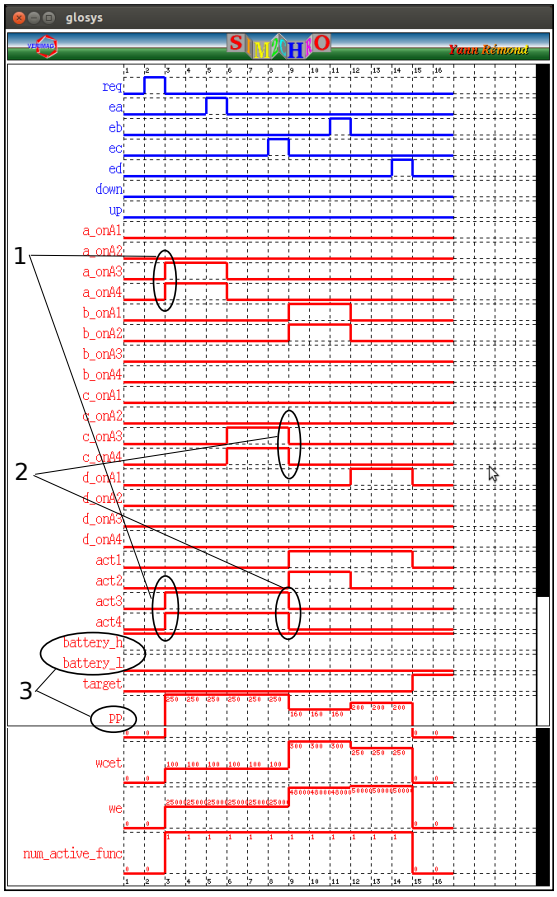

Figure 9: A simulation scenario.

secondly, *down* and *up* events are uncontrollable. The power peak value is thus always kept under 300 to avoid the system goes to an invalid state where a task uses an implementation with a power peak bigger than the value that the lower level allows, i.e., 300, and the battery level goes low before it finishes. The exclusive usages of all the tiles (i.e., objective 1)) can also be seen from the figure, e.g, for tile A1, the variables $t\_onA1, t = \{a, b, c, d\}$ do not have value true at the same time during the simulation. The variable num_active_func representing the number of active tasks is always 1 during simulation, which means that objective 2) is met. The objective 4) is also met as variable $target$, which represents whether the end state T is reached, becomes true at instant 15.

6.2 Controller Integration

With the C code of controller generated by BZR and described in Section 6.2, we can integrate it (represented by box *Controller* in Fig. 10) with the system hardware implementation by using the glue code (right box of Figure 10) which consists of two parts. The initialization part initializes system state variables by invoking **reset** function $sys\_reset()$, and starts the processing of data (e.g. video stream processed by FPGA reconfigurable tiles) by $processing\_start()$. Then, an infinite loop, which performs the following steps: (1) $processing\_control()$ monitors the data processing and checks the timing or conditions to be respected before the reconfiguration controller can be invoked, e.g. *wait the arrival of a new frame of type I* or simply *wait 10 ms* ; (2) $get\_yuc()$ collects the uncontrollable input values from the running system ; (3) sys_step takes as input the values of uncontrollable variables (denoted by Y_{uc}) and the system state variables (denoted by X) and computes the values of the defined controllable variables and consequently the new state variables (X) ; (4) $configure\_hw\_sw(X)$ performs reconfiguration by interpreting the computed values of output variables as system (reconfiguration) actions, it loads the right bitreams from a remote server or from a Flash memory and invokes the ICAP driver to execute the FPGA reconfiguration.

Finally, the above written code together with the C code of controller generated by BZR is deployed on the CPU managing the reconfigurable hardware, e.g., Microblaze.

6.3 A Typical Experimental Setup

We consider an ML605 board from Xilinx as our hardware execution platform. It includes a Virtex-6 FPGA (XC6VLX240T), several I/O interfaces like switches, buttons, Compact Flash reader, and an external 512MB DDR3 memory. An Avnet extension card (DVI I/O FMC Module) with 2 HDMI connectors (In and Out) has been plugged onto the platform so that it can receive and send video streams through the connectors.

Figure 11 illustrates the global structure of our implementation. We have divided the FPGA surface into two regions: static and reconfigurable regions.

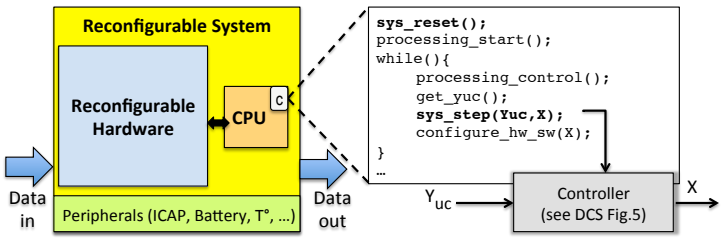

Figure 10: Controller integration with system implementation.

Nine independent reconfigurable tiles are specified in the reconfigurable region. The reconfigurable tiles are in charge of the executions of reconfigurable tasks. The MB is synthesised on the static region of the FPGA (like *A*0 in Figure 1 a)). It executes two main system tasks: *the computed controller* and the management of the configuration bitstreams. The latter task involves the control of related peripherals (i.e., Compact Flash memory, I/O interrupts, DDR3, ICAP) through corresponding implemented controllers. The external DDR 3 memory is used to buffer the input data, e.g., frame pixel data of video streams, and store the software executable, typically *the computed controller*, to be launched by the MB. We use a compact Flash card to store the bitstreams of different reconfigurable task implementations on each reconfigurable tile. The C code of *the computed controller* is deployed on the MB as an infinite loop. It is invoked whenever the MB is interrupted. Two additional interrupt controllers (GPIO switches and GPIO buttons) are added for the platform to generate interrupts. They monitor the states of the buttons and switches, and generate interrupts when these states change. Once the controller is invoked, it is able to read the system states and computes out a new configuration for the nine reconfigurable tiles. The MB then selects the appropriate bitstreams from the Compact Card, and sends them to the ICAP to reconfigure the associated reconfigurable tiles.

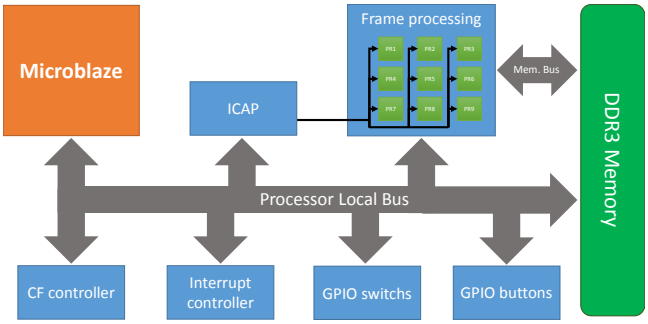

Figure 11: Global structure of the implementation

7 Case Studies

We describe two experimental case studies, to demonstrate the previous control models on real FPGAs, and focus on the modeling and controller generation aspect.

7.1 Case Study I: A Video Processing System

7.1.1 Case Study Description

We consider a video processing system to be implemented on a FPGA board, so that the partial reconfigurations of a FPGA controlled by a synthesized controller can be tested and visualised. The processing system (see Figure 12) consists of a camera that captures images to be processed on the FPGA, a dispatcher that feeds 9 reconfigurable tiles, a compositor aggregating pixels, and a screen displaying the processed images. Each captured image is divided into 9 areas, which are processed in parallel by 9 processing elements dynamically configured in the 9 tiles (as we had four in Figure 1 a)). In this way, when a tile is reconfigured, one can see it on the screen. We consider three filtering algorithms (namely red, green and blue ones) that can be implemented on each reconfigurable tile to process images. When configured to process the same image, they have different performance values regarding some characteristics such as power peak, execution time. In the study, we suppose the power peaks of each tile for running the red, blue and green filters are 3, 2 and 1.

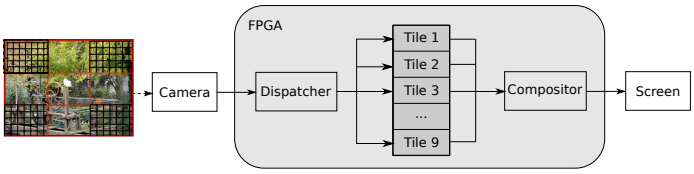

Figure 12: The video processing system case study. Each processed image is divided into 9 areas for processing, with those covered by grids called corner areas, and the rest ones called cross areas.

We then introduce events that will induce state transitions and so reconfigurations. First the processing system can work at two different modes: *high* and *low*, controlled by the user through a switch on the platform. The user can also demand the use of the red filters to process the four corner areas of images by means of another switch. Apart from the user demands, the system also needs to respect the following three rules. The four corner areas of the images to be displayed are of the same color, the five cross areas are of the same color, and the color of the four corner areas is different from the color of the five cross areas ; the global power peaks of the platform are bounded by 30 (respectively 20) in the *high* (respectively *low*) mode ; minimizing the power peaks of the next states. A run-time manager is thus required to configure each reconfigurable

tile of the FPGA by using one of the three algorithms to filter images in the way satisfying the aforementioned requirements.

7.1.2 System Modeling and Controller Generation

We model the system reconfiguration behavior by using *synchronous parallel automata* as in Section 5, and DCS is then performed to generate a controller by using BZR. Once the system gets started (modeled as the emission of event s), the controller should decide on the system initial state and configure the nine reconfigurable tiles of the FPGA accordingly. The behaviors of the two switches, denoted by $ModeSwitch$ and $CornerColorSwitch$, are captured by two boolean variables ms and gr respectively, with value *true* means switch on.

The reconfiguration behavior of the system is captured by a three-state automaton (Figure 13 a)), with Boolean input ms capturing $ModeSwitch$, and Boolean output h representing whether it is in mode $high$. Initially in idle state I, once it is started by s, it goes to either High or Low depending on ms.

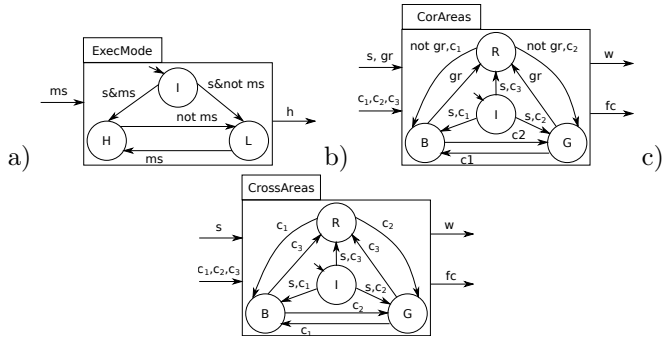

Figure 13: The model of system execution mode behavior $ExecMode$, and the models for choosing algorithms for processing the four corner areas $CorAreas$ and the four cross areas $CrossAreas$

As the colors of the four corner areas are required to be the same, they always need the same filtering algorithm. We thus use one single automaton (see Figure 13 b)) to model their choices among the three filtering algorithms. Boolean inputs s and gr represent respectively whether the system gets started or not, and whether the user has switched on or off the corner color switch, while outputs $fc \in \{corR, corB, corG, corI\}$ and $w \in \{12, 8, 4, 0\}$ represent respectively the current state and the weight associated with the state. At the beginning, it is in state I. Once the system gets started, i.e., event s is received, it goes to state R, G or B, meaning that the red, green or blue filtering algorithm is used for processing the four corner areas of images, depending on the values of controllable variables $c1, c2, c3$. As running the red filter in a reconfigurable tile has cost 3, and R represents that all the four corner areas run the red filter, we associated state R with cost 12. The same to the costs of states G and B. The automaton goes to state R upon the reception of event gr (i.e., the user

switches on $CornerColorSwitch$), when it is in states G or B. The rest of the transitions (e.g., between G and B) are managed by the controller by evaluating the values of controllable variables $c1, c2, c3$ according to system requirements.

The modeling for choosing filters for processing the five cross areas is done similarly. The main difference is that the user now has no control over the usage of some filter for processing the four cross areas, i.e., the choice among the filters are made by the controller through controllable variables $c1, c2, c3$. Figure 13 c) shows the model.

At last, all the aforementioned models are composed to derive the global system behavior. We then encode them and the control rules in BZR and employ BZR to automatically synthesize a controller satisfying the control rules. It generated the C code of controller (with overall size 77.8 kB) within 5 sec (see Table 2).

7.2 Case Study II: ASmart Camera Object Detection System

7.2.1 Case Study Description

We have used an advanced industrial conveyor simulator [6] and a use case where parcels can be conveyed from one location to another. Compared with the previous case, the camera is disconnected and the HDMI output of the PC running the simulator is connected instead to the board video input.

The object detection system detects the moving objects on the conveyor belts, characterizes the moving objects in terms of speed, size, color and moving direction, and makes task implementation choice decisions according to the characteristics of the objects. Figure 14 describes how it works. A camera captures the video frames (*Acquisition task*) and sends them to the detection algorithm, which is implemented on a FPGA (represented by the grey rectangle). The detection algorithm is specified as a data-flow application where circles represent tasks, and arrows represent the communication channels. Numbers are labelled on channels to represent the corresponding numbers of input and output data tokens. Typically, a data token is one pixel, or an integer. Rectangles represent buffers with numbers denoted above to represent buffer sizes. Each video frame is of $N \times M = 1280 \times 720$ pixels, and each pixel is 32 bits. After acquisition the frame is duplicated and sent to tasks *Cleaning* and *Filtering*. *Cleaning* firstly applies erosion and dilation filters and then compares pixel-by-pixel the current frame with the previous one. Then task *Labelisation* identifies the possible object movements according to the comparison results of pixel values and get the coordinates of the object rectangle: top-left (x, y) position, height (h) and width (w). Task *Filtering* filters the frame before task OSD (On Screen Display) can be applied.

With the results of task *Labellisation*, the four following tasks denoted by dotted circles are used to compute area, direction, speed and acceleration of moving objects:

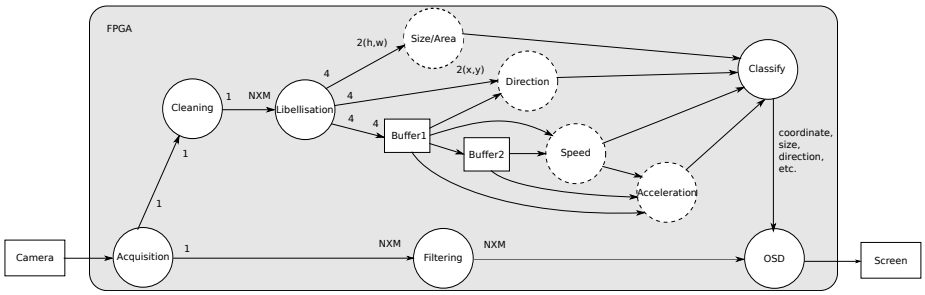

Figure 14: The object detection system case study.

- Task *Size/Area* multiplies the resulting height and width to get the object area.
- Task *Direction* computes direction by comparing current and previous positions.
- Task *Speed* computes the speed of objects by using the two previous positions.
- Task *Acceleration* computes the acceleration by using previous speed and positions.

The task *Classify* takes the analysis results of the four precedent tasks, and classify them accordingly. The analysis results of tasks *speed*, *size* and *acceleration* are classified into one of the three levels: low, medium and high. The result of *direction* is classified into one of the four directions: north, west, east and south. As a result, task *Classify* produces three events *esz, esp, eac* to represent the categories (i.e., high, medium or low) of the size, speed and acceleration of the moving object, and event *edi* to represent its direction: north, west, east or south. Task *OSD (On Screen Display)* displays a rectangle surrounding the moving object, with input data from the two branches.

In this example we consider a reconfigurable architecture composed of 9 tiles that can execute the four tasks (size, speed, direction, acceleration) with three configurations w.r.t. different precisions (QoS). The three configuration high (H), medium (M) and low (L) require 3,2 and 1 tiles respectively. It means that 4 out of 9 tiles are used if all tasks are running with a low resolution but all the tasks cannot run simultaneously with a high resolution. In particular, we suppose that, depending on the moving direction of the detected object, the four reconfigurable tasks are given corresponding preferences (modeled by weights) to use high QoS implementations. We consider the following reconfiguration constraints: the number of available tiles is fixed by 9 ; if no object is detected, low precision implementations will be used for all tasks ; if speed is high, there is no need to use high precision for size ; Optimizing overall QoS: weighted function $QoS = \sum w_i * QoS(t_i)$, where weight values w_i for tasks t_i depends on

the moving direction detected object ; Configuration bitstream imposes adjacent tiles (2 or 3) in vertical or horizontal direction for resolution medium and high.

7.2.2 System Modeling and Controller Generation

Each task has three implementations corresponding to three precision levels. Figure 15 a) models the implementation model. The choices between them are controllable by variables $c1, c2, c3$. Outputs lp, hp represent which implementation is used, and integer outputs qos and t_num represent respectively the QoS and number of used tiles of the current implementation.

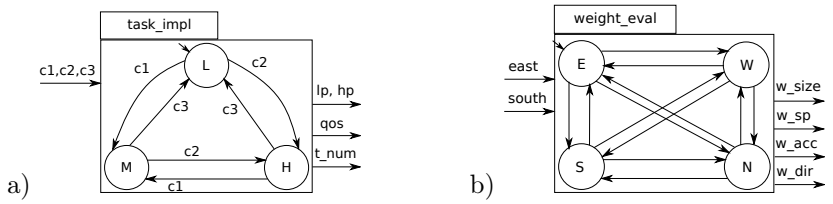

Figure 15: The task implementation model $task\_impl$, and the weight evaluation model $weight\_eval$.

Depending on the moving direction of the detected object, the four reconfigurable tasks are given corresponding preferences (modeled by weights) to use high QoS implementations. Figure 15 b) models the weight evaluation model. It has four states E, W, S and N corresponding to the four directions. The boolean inputs $east, south$ are used to represent the moving directions. The integer outputs w_size, etc represent the correspondingly evaluated weights for the four tasks. We then encode the models and control objectives in BZR and employ BZR to automatically synthesize a controller satisfying the control rules. By feeding the resulting program to the BZR compiler, it generated the C code of controller (with overall size 298.4 kilobytes) within 25 seconds (see Table 2). In both case studies, the partial bitstreams are relatively small (about 50KB) and can be configured in less than 0.5ms with a 100MHz ICAP clock. The maximum camera resolution is 1080x1920 and the frame rate is 60fps, which means a period of 16.6ms. The complete bitstream of a project is about 1.6MB.

Table 2: Summary of the two case studies. The experiments are performed on a computer with Intel(R) Core(TM)2 Duo CPU of 2.33GHz and a 3.8Gb main memory.

case study	number of states	synthesis time (secs)	C code size (kilobytes)
I	48	4	77.8
II	324	21	298.4

8 Discussion on Scalability

The major concern of our approach is the scalability issue, which is common to other formal techniques like model-checking. We have carried out extensive experiments to evaluate the scalability of our framework. Table 3 shows our experimental results to compute the controllers. It gives the time costs for different DCS operations corresponding to different system objectives w.r.t. different system models and state space sizes. The state space size of each system model is computed by simply multiplying state space sizes of its composed automata. The size of synthesized controllers varies from 50Kb (objective 2 on model 4:(2,3,2,3)) to 28Mb (objective 7 on model 6:(1^6)). We have started our experiments from the task graph of Figure 1 b). We then refine B to 3 tasks so as to increase the system model to 6 tasks, and at last, refine C to 3 tasks as well to address a 8 task model, as shown in Figure 16. We use the notation

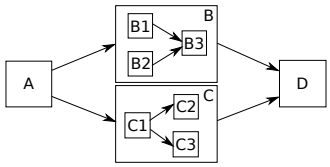

Figure 16: The refined task graph for experiments.

$n : (m_1, ...m_n)$ to represent the models, where n denotes the number of tasks, and m_i the number of possible implementations of task i. Besides, we use m^k to represents k consecutive m's. E.g., $4 : (4^4)$ denotes $4 : (4, 4, 4, 4)$. All experiments are performed on a computer with a Intel(R) Core(TM)2 Duo CPU of 2.33GHz and a 3.8Gb main memory.

Table 3: The time costs for DCS operations corresponding to different target objectives w.r.t. different system models and state space sizes. $n : (m_1, ...m_n)$ denotes a model of n functions, with m_i denoting the number of possible implementations of function Fi. $*$ Objectives 7 and 8 are the same kind of operation (objective 8 is thus omitted here).

system model & **state space size** / target objectives	4:(2,3,2,3) **241,920**	4:(4^4) **806,736**	6:(1^6) **5,898,240**	6:(3,1,1,2,1,3) **16,588,800**	6:(3,2^4,3) **32,400,000**	8:(1^8) **566,231,040**	8(3^3,2^5) **5,832,000,000**
1) exclusive usage of A_1-A_4	0.29sec	0.65sec	0.16sec	1.16sec	8.20sec	1.10sec	16min1sec
2) dual access to memory	0.12sec	0.49sec	0.10sec	0.69sec	1.50sec	1.29sec	23.05sec
3.a) switch to active mode	0.74sec	2.28sec	0.46sec	1.88sec	1min19sec	1.30sec	27min58sec
3.b) switch to sleep mode	0.76sec	2.01sec	0.22sec	1.74sec	2min11sec	0.90sec	41min29sec
5) battery-level constrained power peak	0.89sec	2.23sec	0.68sec	4.18sec	21.18sec	2.21sec	49min24sec
4) reachability:	1.78sec	3.48sec	4.74sec	17.33sec	2min	25.16sec	3hr34min
6) minimize p.p. in next states	3min54sec	3hr18min	29min45sec	*stopped*	*stopped*	*stopped*	*stopped*
7)* minimize remaining WCET:	9min17sec	2hr43min	21min19sec	*stopped*	*stopped*	*stopped*	*stopped*

In our experiments, the DCS of invariance constraints, i.e., objectives 1-3 and 5 are applied directly to the original system model. On the basis of the resulting controller, the optimal and reachability ones are then performed. The objectives about invariance and reachability appear promising, while optimal ones are explosive. An interesting point observed is that the time cost is not always increasing as state space size grows. System models consisting of more tasks but less possible implementations could have less synthesis times, e.g., DCS operations for $6{:}(1^6)$ model take less times than these for $4{:}(4^4)$ model. These observations can be explained as follows by the nature of the DCS algorithms corresponding to different control objectives.

The invariance objectives aim to make a subset E of system states invariant. The synthesis algorithm explores the system states from the initial state(s) and their transitions to return a controllable system such that the controllable transitions 1) leading to states that are not in E are inhibited, and 2) leading to states from where a sequence of uncontrollable transitions that can lead to states not in E are inhibited. The computational cost of this DCS operation thus depends not only on the system state space, but also the size of target state set E, and the number and the guard type of transitions associated with these states. Since more implementations of tasks mean more choices/transitions to explore, this explains why the computational cost w.r.t. these objectives for more tasks with less implementations can be less.

The rechability objective aims to make a subset E of system states always reachable from current states. Its corresponding algorithm thus explores the system states from the initial state(s) and their transitions to return a controllable system such that the controllable transitions entering subsets of states E' from where E is not reachable are disabled. Compared to the synthesis algorithm for invariance objectives, it generally takes more time as it needs to firstly explore states and transitions to search for E' and secondly explore the rest states and their transitions to compute the values of controllable variables. This can explain why the time costs w.r.t. the rechability objective are more than those corresponding to invariance objectives.

The optimal control objectives aim to optimize the costs or weights defined on the states and/or transitions of system automaton models. Their time costs get much higher compared the two aforementioned objective types, as their DCS algorithms perform not only the state and transition exploration, but also cost computations and comparisons. To improve the scalability of our approach and address systems of more tasks, especially when optimal control objectives are enforced, one can on the one hand employ more powerful PCs and spend more time, and one the other hand, improve the efficiency of employed synthesis tools and employ modular DCS presented in [8]. The main idea of the modular DCS is to break the system into subsystems by structuring task graphs into hierarchical sub-graphs, and perform local DCS for each subsystem before performing the global DCS for the whole system.

9 Conclusion and perspectives

We described the management of dynamically partially reconfigurable FPGAs, where formal guarantees are given on the behavior of the reconfigurable system in terms of reachable state space. Our contribution consists of a tool-supported method to design safe controllers for dynamically reconfigurable architectures, and its experimental validation on two case studies using a FPGA board. Our approach is to formalize the behaviors of the DPR FPGAs as automata, following a modeling methodology, distinguishing the different levels of hardware architecture, task implementation and application software. We formulate the reconfiguration policy as properties on the state space of the model, and the reconfiguration control as a Discrete Controller Synthesis problem. The BZR language and compiler is used to implement the models, solve the control problems and generate executable C code.

Concerning architecture aspects, this formal approach allows to design complex self-adaptive SoC that are correct-by-construction. This point is crucial to avoid intractable scenario-based simulations. Our formal approach paves the way to the safe use of future reconfigurable architectures with efficient power gating capabilities (ex. MTJ-based FPGA [34]), that reconfiguration can be exploited to finely adjust power consumption to application requirements. Perspectives are in different directions: concerning formal modeling and control, the exploitation of modular compilation and DCS can improve the scalability of the approach on large systems, provided they can be structured hierarchically. The extension of DCS to logico-numeric aspects is being integrated in BZR and can support some quantitative aspects of systems, and more elaborate control objectives. Finally, the automatic generation of hardware implementation corresponding to the controllers built from our approach is an interesting perspective. For this purpose, the backend of BZR compiler needs to be extended for generating VHDL programs and FPGA bitstreams.

References

[1] K. Altisen, A. Clodic, F. Maraninchi, and E. Rutten. Using controller-synthesis techniques to build property-enforcing layers. In *Proc. 12th European Conference on Programming*, ESOP'03, pages 174–188, 2003.

[2] X. An. *High Level Design and Control of Adaptive MPSoCs*. PhD thesis, U. Grenoble, 2013.

[3] X. An, E. Rutten, J-P. Diguet, N. le Griguer, and A. Gamatié. Autonomic management of dynamically partially reconfigurable fpga architectures using discrete control. In *Proc. 10th Int. Conf. Autonomic Computing (ICAC'13)*, pages 59–63, june 2013.

[4] X. An, E. Rutten, J-P. Diguet, N. le Griguer, and A. Gamatié. Discrete control for reconfigurable fpga-based embedded systems. In *Proc. 4th IFAC Workshop on DCDS*, pages 151–156, 2013.

[5] J. Aylward, C. H. Crawford, K. Inoue, S. Lekuch, K. Müller, M. Nutter, H. Penner, K. Schleupen, and J. Xenidis. Reconfigurable systems and flexible programming for hardware design, verification and software enablement for system-on-a-chip architectures. In *Proc. Conf. on Reconfigurable Computing and FPGAs (ReConFig'11)*, pages 351–356, 2011.

[6] R. Bévan, J-L. Lallican, W. Allègre, and P. Berruet. The simsed framework for modelling and simulation of transitic systems under uncertain environment. In *9th Int. Industrial Simulation Conf.*, pages 11–17, 2011.

[7] O. Dahmoune and R. de B. Johnston. Applying model-checking to post-silicon-verification: Bridging the specification-realisation gap. In *Proc. Conf. on ReConFig*, pages 73–78, 2010.

[8] G. Delaval, H. Marchand, and E. Rutten. Contracts for modular discrete controller synthesis. In *Conf. on Languages, Compilers, and Tools for Embedded Systems*, pages 57–66, 2010.

[9] E. Dumitrescu, A. Girault, H. Marchand, and E. Rutten. Multicriteria optimal discrete controller synthesis for fault-tolerant tasks. In *Workshop on Discrete Event Systems*, pages 366–373, Sept. 2010.

[10] Y. Eustache and J.-P. Diguet. Specification and os-based implementation of self-adaptive, hardware/software embedded systems. In *Proc. 6th Int. Conf. on CODES/ISSS*, pages 67–72, 2008.

[11] A. Gamatié, H. YU, G. Delaval, and E. Rutten. A case study on controller synthesis for data-intensive embedded systems. In *Proc. 6th IEEE Int. Conf. on Embedded Software and Systems, ICESS'09*, pages 75–82, 2009.

[12] F. Ghaffari, M. Auguin, M. Abid, and M.B. Ben Jemaa. Dynamic and on-line design space exploration for reconfigurable architectures. In *Transactions on High-Performance Embedded Architectures and Compilers I*, volume 4050, pages 179–193. 2007.

[13] D. Gohringer, M.Hubner, V.Schatz, and J.Becker. Runtime adaptive multiprocessor system-on-chip: RAMPSoC. In *Symp. on Parallel & Distributed Processing*, pages 1–7, April 2008.

[14] L. Gong and O. Diessel. Modeling dynamically reconfigurable systems for simulation-based functional verification. In *19th Int. Symp. on Field-Programmable Custom Computing Machines*, pages 9–16, 2011.

[15] S. Guillet, F. de Lamotte, N. Le Griguer, E. Rutten, G. Gogniat, and J-P. Diguet. Designing formal reconfiguration control using uml/marte. In *Proc. Int. Conf. on ReCoSoC*, pages 1–8, 2012.

[16] S. Guillet, F. de Lamotte, N. le Griguer, E. Rutten, G. Gogniat, and J.-P. Diguet. Extending uml/marte to support discrete controller synthesis, application to reconfigurable systems-on-chip modeling. *ACM Trans. Reconfigurable Technol. Syst.*, 7(3):27:1–27:17, 2014.

[17] H. Hinkelmann, P. Zipf, and M. Glesner. Design and evaluation of an energy-efficient dynamically reconfigurable architecture for wireless sensor nodes. In *FPL Conf.*, pages 359–366, 2009.

[18] S. Jovanović, C. Tanougast, and S. Weber. A new self-managing hardware design approach for fpga-based reconfigurable systems. In *Reconfigurable Computing: Architectures, Tools and Applications*, volume 4943 of *Lecture Notes in Computer Science*, pages 160–171. 2008.

[19] C. Lu, J.A. Stankovic, S.H. Son, and G. Tao. Feedback control real-time scheduling: Framework, modeling and algorithms. *Real-Time Systems Journal*, 23(1/2):85–126, 2002.

[20] F. Madlener, J. Weingart, and S. A. Huss. Verification of dynamically reconfigurable embedded systems by model transformation rules. In *Int. Conf. on CODES+ISSS*, pages 33–40, 2010.

[21] M. Maggio, H. Hoffmann, A. V. Papadopoulos, J. Panerati, M. D. Santambrogio, A. Agarwal, and A. Leva. Comparison of decision-making strategies for self-optimization in autonomic computing systems. *ACM Trans. Auton. Adapt. Syst.*, 7(4):36:1–36:32, 2012.

[22] M. Majer, J. Teich, A. Ahmadinia, and C. Bobda. The erlangen slot machine: A dynamically reconfigurable fpga-based computer. *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 47:15–31, 2007.

[23] H. Marchand and M. Samaan. Incremental design of a power transformer station controller using a controller synthesis methodology. *IEEE Trans. on Soft. Eng.*, 26(8):729 –741, 2000.

[24] G. Martin, B. Bailey, and A. Piziali. *ESL design and verification: a prescription for electronic system level methodology.* Morgan Kaufmann, 2010.

[25] C. Maxfield. *The design warrior's guide to FPGAs: devices, tools and flows.* Elsevier, 2004.

[26] Juanjo Noguera and Rosa M. Badia. Multitasking on reconfigurable architectures: Microarchitecture support and dynamic scheduling. *ACM Trans. Embed. Comput. Syst.*, 3(2):385–406, 2004.

[27] V. Nollet, P. Avasare, H. Eeckhaut, D. Verkest, and H. Corporaal. Run-time management of a mpsoc containing fpga fabric tiles. *IEEE Trans. VLSI Systems*, 16(1):24 –33, 2008.

[28] Object Management Group. A UML profile for MARTE, 2013.

[29] K. Paulsson, M. Hubner, and J. Becker. Strategies to on-line failure recovery in self-adaptive systems based on dynamic and partial reconfiguration. In *Conf. Adap. Hardware and Systems*, pages 288–291, 2006.

[30] I. R. Quadri, H. Yu, A. Gamatié, E. Rutten, S. Meftali, and J-L. Dekeyser. Targeting reconfigurable fpga based socs using the uml marte profile: from high abstraction levels to code generation. *Int. J. of Embedded Systems*, 4(3/4):204–224, 2010.

[31] P.J. Ramadge and W.M. Wonham. The control of discrete event systems. *Proc. of the IEEE*, 77(1):81–98, 1989.

[32] S. Singh and C. J. Lillieroth. Formal verification of reconfigurable cores. In *FCCM*, pages 25–32, 1999.

[33] F. Sironi, M. Triverio, H. Hoffmann, M. Maggio, and M.D. Santambrogio. Self-aware adaptation in FPGA-based systems. In *Field Programmable Logic and Applications*, pages 187–192, 2010.

[34] D. Suzuki, N Natsui, A Mochizuki, S Miura, H. Honjo, K. Kinoshita, H. Sato, S. Ikeda, T. Endoh, H. Ohno, and T. Hanyu. Fabrication of a magnetic tunnel junction-based 240-tile nonvolatile field-programmable gate array chip skipping wasted write operations for greedy power-reduced logic applications. *IEICE Electronics Express*, 10(23):20130772, 2013.

[35] K.S. Vallerio and N.K. Jha. Task graph extraction for embedded system synthesis. In *Proc.16th Int. Conf. on VLSI Design*, pages 480–486, 2003.

[36] J. Vidal, F. De Lamotte, G. Gogniat, P. Soulard, and J-P. Diguet. A co-design approach for embedded system modeling and code generation with uml and marte. In *DATE*, pages 226–231, 2009.

[37] L. Ye, J-P Diguet, and G. Gogniat. Rapid application development on multi-processor reconfigurable systems. In *Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 285–290, 2010.