2010 International Conference on Reconfigurable Computing

Modeling and formal control of partial dynamic reconfiguration

Sébastien Guillet*, Florent de Lamotte*, Éric Rutten**, Guy Gogniat*, Jean-Philippe Diguet*
* Lab-STICC, Université de Bretagne Sud, France
** INRIA Rhône-Alpes, France
This work is supported by the ANR FAMOUS project (ANR-09-SEGI-003)

***Abstract*—This paper introduces an approach for the safe design and modeling of dynamically reconfigurable FPGA based Systems-on-Chip. This approach is carried out in a design framework, GASPARD2, dedicated to high-performance embedded systems modeling using the OMG standard profile UML/MARTE. Information employed by the reconfiguration mechanism is identified to be extracted from MARTE models in order to synthesize a controller using a formal technique which significantly simplifies the correct design of reconfiguration control. This methodology is then demonstrated in a case study.**

I. INTRODUCTION

Complexity in Systems-on-Chip (SoC) has been increasing over the last few years, raising the need for proper design tools and methodologies. The Model Driven Architecture (MDA) intiative is successfully employed to solve various problems tied to SoC design, especially by hiding classic low level implementation details from the designer [5] [22] [2] [18].

The current study is based on the hypothesis that the need in terms of self reconfigurable embedded systems will increase and generalize for two reasons: i) Adaptation to the environment. Hardware modification to cope with uncontrollable environment is a key element in terms of dynamic optimization (failure, quality of service, energy consumption...). ii) System updates. An embedded system, in the same way as software, should be able to evolve during its existence in oder to follow the user needs (prototyping, updating and bug correcting).

In this context, this study will tackle two of the designer needs: i) Modeling. Self adaptive architectures are not a new concept. However their use in an industrial context is not generally observed. Performance, energy consumption and design complexity compared to unreconfigurable counterparts are often strong arguments against adoption. Reconfigurable architectures will not be exploited at their full potential unless an appropriate design methodology is provided. ii) Safe reconfiguration control and decision. Designing correctness is an important issue in critical SoCs, as a single error can lead to a financial, material and/or human disaster. Adding reconfiguration concepts in such SoCs increases the design complexity as adaptive systems need to be both secure and optimized to compromise between contradictory constraints (typically *Quality of Service* and *energy consumption* [3]). A correct by construction control generation, helping the designer in the reconfiguration decision implementation, is chosen in this study to answer that problem. So the contribution of this study is to show how a MARTE model from Gaspard2, containing information on how reconfigurations can be controlled, can be used in combination with control objectives to obtain a controller which guarantees a correct reconfiguration behavior.

This paper is organized as follows: Sections II, III and IV present the key concepts, tools, and related work on which this study is based, ie. High Level Modeling, Discrete Controller Synthesis, and the execution platform. Section V explains the transformation chain from high level to execution. Section VI shows the case study. Finally, in section VII, we draw some conclusions.

II. HIGH LEVEL MODELING

A. Model Driven Engineering (MDE)

This study follows a MDE approach for SoC Co-Design specification and development. It allows to use a component based model driven methodology to abstract and simplify the system specifications. The Object Management Group (OMG) has provided such approach in the form of the Modeling and Analysis of Real-Time and Embedded Systems (MARTE) Unified Modeling Language (UML) profile, gradually becoming a de-facto industry standard [7].

B. GASPARD2 *framework*

GASPARD2 [21] uses the MARTE profile to carry out, in a graphical manner, unified high level specifications of SoC applications and architectures along with their allocations and the eventual deployment to Intellectual Properties (IPs). It integrates a compilation chain to target different execution platforms by transforming the high level models into suitable code for SoC creation.

SoCs designed in the GASPARD2 framework relies on a repetitive Model of Computation (MoC) inspired by the industrial domain specific language Array-OL in which multidimensional structures are manipulated and describe both task and data parallelism [4]. An application in GASPARD2 is then specified as a component dependency acyclic graph, with respect to this MoC. Components (tasks) are connected via ports, each one having a shape (or dimension).

Related to Dynamic Partial Reconfiguration (DPR), control semantics have been introduced into GASPARD2 and MARTE to specify reconfiguration behavior at high modeling levels [24] [2]. These semantics follow the reactive mode automata formalism [14] [8]. Mode automata were chosen due to their compositional nature and are exploited to produce the main part of a reconfiguration controller. These semantics are being used in this study by a formal technique, named ”Discrete Controller Synthesis”, to generate the controller main part with respect to rules, instead of doing it manually.

III. Discrete Controller Synthesis (DCS)

A. Automaton model of reactive systems

One level of adaptive systems is related to events and states, defining execution modes or configurations of the system, with changes in the architecture and in the activation of components. Reactive languages based on finite state automata are widely used for these aspects, like StateCharts [11] or StateFlow in Matlab/Simulink or UML variants. Their underlying model, transition systems, is also the basic formalism for discrete control theory, which studies closed-loop control of discrete-event and logical aspects of systems [9].

Adaptive SoCs considered in this study are modeled as a set of components, each one having particular individual constraints. Managing these constraints is performed in a mixed imperative/declarative style where constraints intrinsic to one component are modeled by an automaton. Controller synthesis techniques and tools are used in order to combine the set of communicating automata (from components) with global constraints.

B. DCS and contracts for adaptation policy

DCS was originally defined in the framework of language theory, often called supervisory control of discrete event systems, and is related to game theory. It is one of the automated techniques that can exploit transition system models, and involves algorithms that explore symbolically the state space in a way like model-checking verification, with complexity issues and capacities of the same order. It consists in considering on the one hand, the set of possible behaviors of a discrete event system, where inputs are partitioned into uncontrollable and controllable ones. The uncontrollable inputs typically come from the system’s environment, while the values of the controllable inputs are given by the synthesized controller itself. On the other hand, it requires a specification of a control objective: a property typically concerning reachability or invariance of a state space subset. Such programming makes use of reconfiguration policy by logical contract. Namely, specifications with contracts amount to specify declaratively the control objective, and to have an automaton describing possible behaviors, rather than writing down the complete correct control solution. The basic case is that of contracts on logical properties i.e., involving only Boolean conditions on states and events.

Within the synchronous approach [20], DCS has been defined and implemented as a tool integrated with the synchronous languages: SIGALI [10]. It handles transition systems with the multi-event labels typical of the synchronous approach, and features weight functions mechanisms to introduce some quantitative information and perform optimal DCS. It has been applied to the generation of correct task

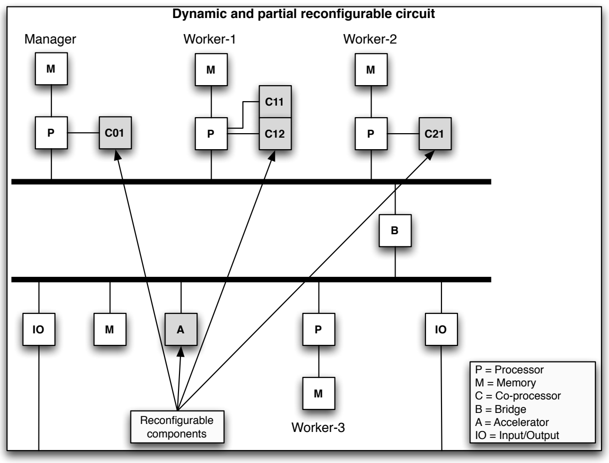


Figure 1. Reconfigurable multiprocessor architecture model

handlers, adaptive resource management [16], safe design of data-intensive embedded systems [15], and integrated in a synchronous language, named BZR.

C. The synchronous language BZR

BZR [19] [17] is used in the case study, and DCS is embedded in this user-friendly programming language. Models of the possible behaviors of the managed system are specified in terms of hierarchical parallel automata, and adaptation policies are specified in terms of contracts on invariance properties to be enforced. The automata part of a BZR program will be extracted from the MARTE specification, whereas the adaptation policy is directly specified in BZR as a set of boolean statements called the ”BZR contract”; one of which is shown in the case study. Compiling BZR yields a correct-by-construction controller (here in C), with the help of DCS, which can then be integrated into the execution platform of choice.

IV. Execution platform

Validation of the global approach relies on a clear definition of the hardware platform. Numerous experiences have been carried out in the domain of reconfigurable embedded systems. Eustache et al. [3] present a safe and efficient solution to manage asynchronous configurations of dynamically reconfigurable SoCs. In [6] Bomel et al. present a networked lightweight and partially reconfigurable platform assisted by a remote bitstreams server. The RAMPSoC [1] is an interesting project, where are addressed the necessity to find a trade-off between homogeneous and application-specific SoC, and a meet-in the middle design methodology to offer runtime configuration capabilities. The HDCRAM [12] approach addresses the architecture management to be inserted inside cognitive radio equipments and provides support for sensing and decision-making facilities.

As can be considered from these researches, typical Multiprocessor Systems on Programmable Chip are composed of

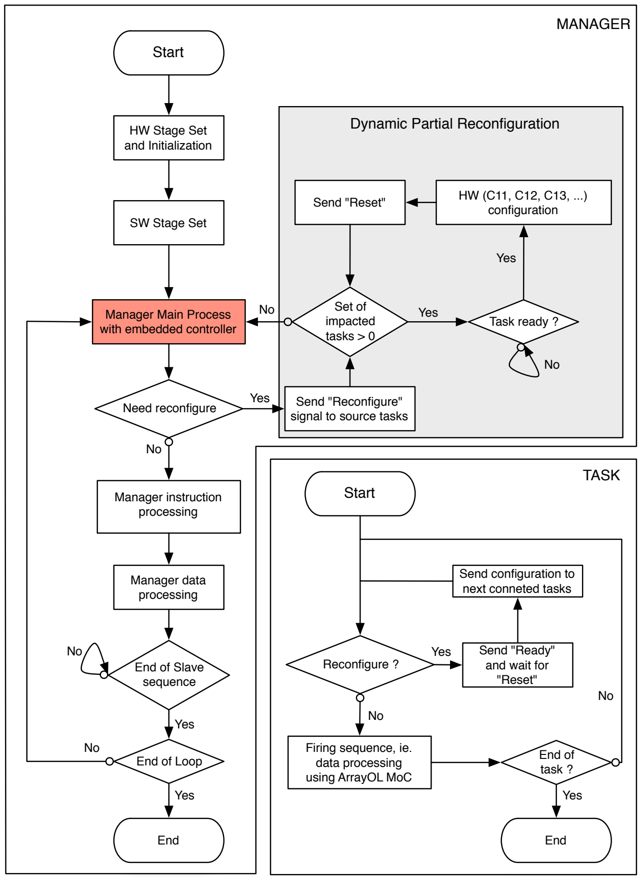

MANAGER
Start
HW Stage Set and Initialization
SW Stage Set
Manager Main Process with embedded controller
Need reconfigure
Manager instruction processing
Manager data processing
End of Slave sequence
End of Loop
End
Dynamic Partial Reconfiguration
Send "Reset"
HW (C11, C12, C13, ...) configuration
Set of impacted tasks > 0
Task ready ?
Send "Reconfigure" signal to source tasks
TASK
Start
Send configuration to next conneted tasks
Reconfigure ?
Send "Ready" and wait for "Reset"
Firing sequence, ie. data processing using ArrayOL MoC
End of task ?
End
Yes
No

Figure 2. Configuration processing flowchart

a set of embedded processors executing tasks, which communicate together to implement the system functionality (Figure 1). Specific IPs (Intellectual Property) can be considered to speed up the computation of intensive tasks. These IPs can be used as co-processors directly connected to processors or accelerators connected to internal buses. One of these kinds of models of platform is a reconfigurable one, named XPSoC, presented in [23]. This study is based on a similar model.

A. *XPSoC architecture*

The architecture model is composed of a manager processor (Microblaze from Xilinx) controlling some slave processors with a given number of co-processors and various other components such as shared memories, IP and peripherals. Basically, the manager is composed of a General Purpose Processor interfacing the system with its environment; it runs PetaLinux [13] and is mainly responsible for the coordination of the system, software and input/output management, and thus the controller obtained through DCS interacts with it.

B. *Dynamic reconfiguration process*

Figure 2 describes the reconfiguration flow used by the SoC supporting the current study. This flow is in accordance with the XPSoC one, and replaces the control part (previously entirely written by hand) by the code generated from BZR.

The SoC automatically loads its first hardware architecture (configuration) from non-volatile memory (e.g. Flash) after power-on, and then loads boot binaries for the manager (here the PetaLinux Operating System) and the slave(s) (here the example application boots waiting for its first configuration). The manager launches the main process, which is responsible for managing all the tasks (organized in component dependency acyclic graphs) and dynamic reconfiguration modules. This main process now embeds the controller generated from BZR, which means it sends events from the environment to the controller and lets it compute the values of the necessary controllable variables (in a maximally permissive manner). Decision part comes next, meaning that if more than one configuration are allowed by the controller, a choice must be set. Decision is not part of this study, but it has to be noted that whatever the quality of its implementation, decision is always safe with respect to control objectives because it can only choose a configuration in a set of secure ones given by the synthesized controller.

If the manager determines a new configuration, it sends a *Reconfigure* signal to the source tasks of impacted Workers. Then it downloads the next configuration stream to memory and performs hardware reconfiguration when it receives a *Ready* signal from an impacted task. This task then receives a *Reset* signal from the master when the new configuration is put in place. Upon receiving this *Reset* signal, the impacted task sends the *Reconfigure* signal to its next connected tasks in the acyclic graph. This signal embeds an ID, thus avoiding multiple same reconfigurations of the same task in a sequence.

V. Transformation Chain

A. *Design flow*

Figure 3 illustrates the contribution of this paper with respect to the GASPARD2 framework. Hardware and software are initially modeled, associated and deployed at a high abstraction level, conforming to the UML/MARTE profile. From the deployed MARTE model, the application is transformed into a hardware functionality (specifically a hardware accelerator or co-processor having multiple implementations) by the intermediate Register Transfer Level (RTL) representation, while the control model is transformed into a BZR program on which DCS is performed to provide the control code of a reconfiguration controller merged and exploited with respect to the XPSoC concepts.

B. *Control information extraction*

In the previously described MARTE models, the tasks of the system are associated with mode automata, for which each mode is associated to an implementation. As this study is concerned by reconfiguration control, the aim is to let the designer specify what can be controlled in the system by placing controllable variables on transitions from automata.

Additional refinement is provided to optimize control: quantitative attributes, or *weights*, representing non functional

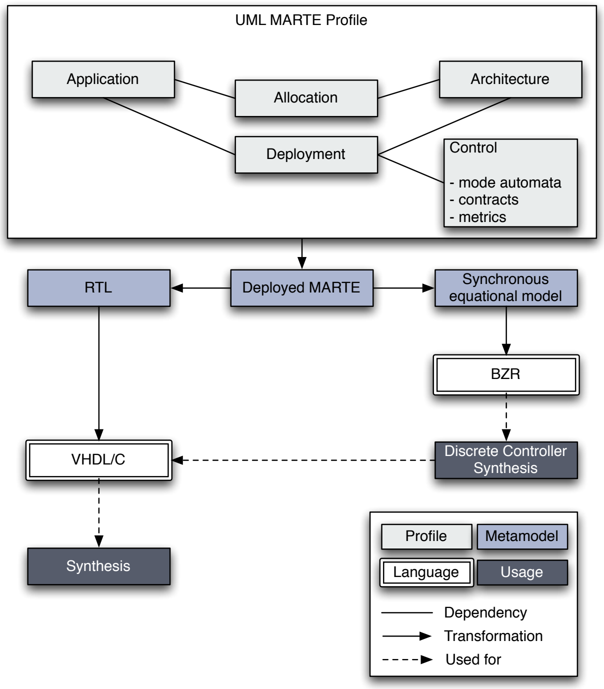


Figure 3. The GASPARD2 framework with integrated control aspects

properties (energy consumption, memory footprint, quality of service, etc.) can be associated to modes. The designer is supposed to set the same type of weights on every modes that need to be further compared. A type of weight defines a partial order and a binary operation which can be used to perform control (enforcing rules on these weights) and decision (deciding between remaining allowed implementations) based on rules (or *Contracts*) on modes or weights.

Three kinds of control information coming from a MARTE specification are thus identified: mode automata (with controllable variables), contracts and properties (with partial order and binary operation). Currently, only mode automata are officially supported in MARTE. The standard is being reworked in this project (ANR FAMOUS) to cope with further control needs. So for this study, only mode automata can really be extracted from a MARTE specification (contracts, for example, are still written directly in BZR).

VI. CASE STUDY

This section presents the control aspects modeling of a system composed of three tasks T1, T2 and T3, where T1 (source task) takes a picture from the environment and sends it to T2 which applies a filter on this image then sends it to T3 which displays it. Each task has multiple implementations. The choice of an implementation is restricted by a synthesized controller, obtained through DCS, enforcing a contract (reconfiguration policy) described further.

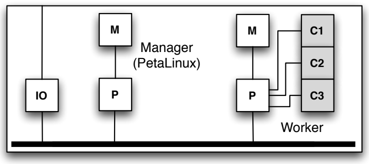


Figure 4. Platform of the case study

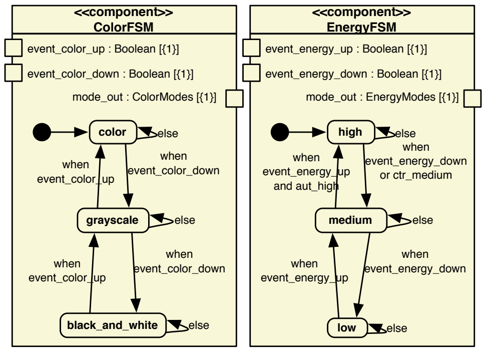

<<component>> ColorFSM
event_color_up : Boolean [{1}]
event_color_down : Boolean [{1}]
mode_out : ColorModes [{1}]
color
else
when event_color_up
when event_color_down
grayscale
else
when event_color_up
when event_color_down
black_and_white
else
<<component>> EnergyFSM
event_energy_up : Boolean [{1}]
event_energy_down : Boolean [{1}]
mode_out : EnergyModes [{1}]
high
else
when event_energy_up and aut_high
when event_energy_down or ctr_medium
medium
else
when event_energy_up
when event_energy_down
low
else

Figure 5. Color and Energy SMCs

A. Platform model

The platform model used in this study, shown in Figure 4, is an instance of the XPSoC model. In this case it is made of one manager processor and one slave processor being dynamically reconfigurable by means of three coprocessors C1, C2 and C3 respectively used by tasks T1, T2 and T3. The association between hardware and software components stays simple in this example. More complex associations are shown in [22].

B. Application model

With respect to the GASPARD2 underlying MoC, tasks can be executed in different ways. the first one is a pure software execution by the manager as a Linux thread. The second solution is a software execution of the thread on a slave. The third method consists in running the thread on a slave with a standard function implemented on coprocessor. The last possible implementation is when the whole thread can be executed by a dedicated hardware accelerator. For each version, a software binary file and/or a bitstream are/is required.

C. Behavior specification

State Machine Components (SMC) are used in GASPARD2 to adopt a state-based control, following the reactive mode automata formalism. In the current example, two SMC are defined to control two different aspects of the system: Color and Energy. As shown in Figure 5, the SMC named EnergyFSM has an interface that includes input Boolean ports (*event_energy_up*, *event_energy_down*) and an output mode port (*mode_out*). Values from input ports are dispatched to trigger transitions. A mode value specified in a state is conveyed through the *mode_out* port. Variables *ctr_medium* and

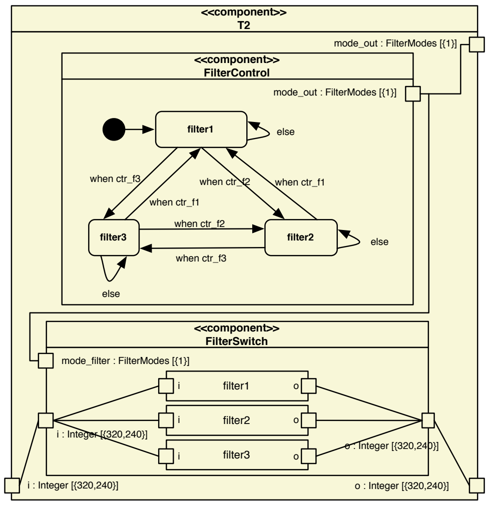
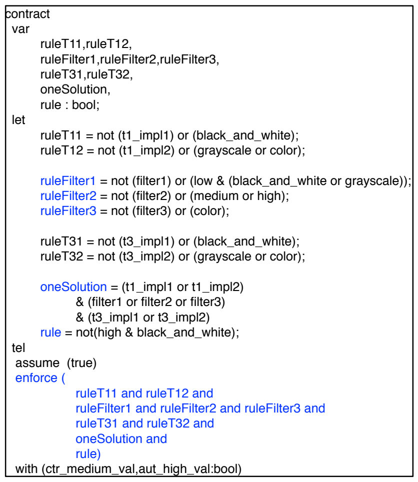


Figure 6. Modeswitch component connected with its default SMC



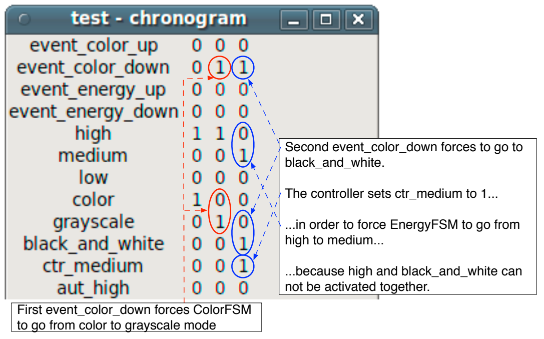
```
contract
 var
       ruleT11,ruleT12,
       ruleFilter1,ruleFilter2,ruleFilter3,
       ruleT31,ruleT32,
       oneSolution,
       rule : bool;
 let
       ruleT11 = not (t1_impl1) or (black_and_white);
       ruleT12 = not (t1_impl2) or (grayscale or color);

       ruleFilter1 = not (filter1) or (low & (black_and_white or grayscale));
       ruleFilter2 = not (filter2) or (medium or high);
       ruleFilter3 = not (filter3) or (color);

       ruleT31 = not (t3_impl1) or (black_and_white);
       ruleT32 = not (t3_impl2) or (grayscale or color);

       oneSolution = (t1_impl1 or t1_impl2)
               & (filter1 or filter2 or filter3)
               & (t3_impl1 or t3_impl2)
       rule = not(high & black_and_white);
 tel
  assume  (true)
  enforce (
               ruleT11 and ruleT12 and
               ruleFilter1 and ruleFilter2 and ruleFilter3 and
               ruleT31 and ruleT32 and
               oneSolution and
               rule)
  with (ctr_medium_val,aut_high_val:bool)
```

Figure 7. BZR contract



```
test - chronogram
     event_color_up     0 0 0
   event_color_down     0 1 1
    event_energy_up     0 0 0
  event_energy_down     0 0 0
               high     1 1 0
             medium     0 0 1
                low     0 0 0
              color     1 0 0
          grayscale     0 1 0
    black_and_white     0 0 1
         ctr_medium     0 0 1
           aut_high     0 0 0
```

Second event_color_down forces to go to black_and_white.

The controller sets ctr_medium to 1...

...in order to force EnergyFSM to go from high to medium...

...because high and black_and_white can not be activated together.

First event_color_down forces ColorFSM to go from color to grayscale mode

Figure 8. Simulation of the controlled system

aut_high are defined by the designer to give the DCS tool the necessary information on what can be controlled in the system. The synthesized controller will only be able to modify these variables which may have an impact on forcing or inhibiting a transition. A task under control may have several exclusive implementations named *running modes* which can be activated at run time by automata, where each state is associated with a mode. Mode Switch Components (MSC) define a switch function between different modes and are composed with SMC to represent the state/mode association. Figure 6 shows a default SMC connected a MSC for the task T2. Switching from a mode to an other without restriction for the controller is the default behavior at this level, which means that the SMC is like a complete graph where each vertex is a state and each edge is a controlled transition. This default SCM is automatically generated but for some reasons, a designer could want to reduce controllability from this point (for example to forbid a transition from a mode to an other).

D. Contract specification

The two SMC (ControlFSM and EnergyFSM) specified earlier are not connected to any MSC, but with the help of BZR contracts, their behavior will impact the implementations choices from the MSCs for the system's tasks.

Two implementations are given for the task T1, named T1_impl1 and T1_impl2; three for T2 (filter1, filter2 and filter3) and two for T3 (T3_impl1 and T3_impl2). Now let's say the filter1 is useful when the system is in Low state for the EnergyFSM SMC and in Black_and_white or Grayscale for the ColorFSM SMC. The corresponding invariance property is denoted *not(filter1) or (Low and (Black_and_white or Grayscale))*. For this property, the controller will constrain the values of the controllable variables in the SMC shown in Figure 6. Now let's introduce a constraint between ControlFSM and EnergyFSM. For example, it can be interesting to specify that the modes High (from Energy) and Black_and_white (from Color) are never active at the same instant. Such a state property is given by *not(High and Black_and_white)*. To finish, the designer wants to ensure the system can choose (at least) one implementation for each of its task for every legal state combination of Color and Energy SMC. This is checked by the following property: *(T1_impl1 or T1_impl2) and (filter1 or filter2 or filter3) and (T3_impl1 or T3_impl2)*. This property is true only when at least one mode is activated for each task. Actually, an important required property of state machines is determinism. Here it means that, given an instant, only one mode by SMC can be activated. But given the contract shown in Figure 7, one can see that when the system is in Medium and Color modes, filter2 and filter3 can be activated. In reality, only one of them will be activated but the controller will not decide. It will only constrain the necessary controllable variables (to enforce the contract) in a maximally permissive manner and let the designer implement an algorithm to decide for the values of the other controllable variables.

E. DCS application

The previous state machines and controllable variables are extracted from the MARTE specification and combined with the contract from Figure 7 to form a BZR program. This program is then compiled into synchronous equations to feed the SIGALI tool which performs DCS. At the end of its computation, SIGALI has found no problem and thus, it means that it has successfully produced a correct by construction controller enforcing the specified rules for every possible executions of the system. The prototype showing the integration of the controller within the execution platform is not yet ready. But for now, SIGALI comes with a tool that enables the simulation of a controller and it will be sufficient to give an idea about the final result. So the controller computed by SIGALI is extracted and simulated (Figure 8). It has to be noted that the simulator doesn't provide a smart decision when multiple choices are given by the controller (remaining controllable variables are set to zero by default) so we will only focus on Color and Energy SMC and controllable variables evolutions. The system is set in High and Grayscale modes by sending event_color_down. Then an other event_color_down is sent. The simulation tool shows that the ctr_medium is set to true, meaning that the controller has triggered the transition from High to Medium state, because staying in High is forbidden in combination with Black_and_white and there is no way for the controller to inhibit the transition from Grayscale to Black_and_white. This is (without surprise because the controller is formally correct) exactly the behavior expected for the example of this study. It should be noted that, if all this automated process had to be done by hand, which is the classical approach, the corresponding controller would be created as a last automata explicitly showing the synchronous composition of the others. To give an idea, if the system had 20 automata, each one having two states, the controller could necessitate (worst case) 2^{20} states to represent the synchronous composition. This (huge!) work should then be validated by model checking to formally certify the correctness of the controller. It can even be the case where such controller does not exist, and it is then even better when an automatic approach such as DCS can give a diagnostic about why it cannot exist, instead of testing solutions by hand.

VII. Conclusion and future works

A safe approach to design self-reconfigurable systems on chip by integrating rapid prototyping of adaptation policy into a MDE methodology has been introduced in this paper. It relies on information extraction (mode automata, controllable events, rules and properties) from a modified MARTE specification in order to generate the self-reconfiguration process by discrete controller synthesis. The DCS formal method is made usable by non-experts as it is embedded in a language (BZR) and compiler, itself embedded in the MDE flow. The aim of this approach is to enhance and facilitate the industrial adoption of the design flow introduced by the GASPARD2 framework to conceive self-reconfigurable SoC. The next steps will concern the implementation of a prototype, involving intensive signal processing, to demonstrate the methodology. This work is also at the heart of reconfiguration control validation right from the start of a reconfigurable embedded system design and is preliminary of future works is this domain.

References

[1] D. Gohringer; M. Hubner; V.Schatz; J. Becker. Runtime adaptive multiprocessor system-on-chip: Rampsoc. *IPDPS*, pages 1–7, April 2008.

[2] I.R. Quadri; S. Meftali; J.-L. Dekeyser. High level modeling of dynamic reconfigurable fpgas. *International Journal of Reconfigurable Computing*, 2009.

[3] Y. Eustache; J.-P. Diguet. Reconfiguration management in the context of rtos-based hw/sw embedded systems. *EURASIP Journal on Embedded Systems*, 2008.

[4] A. Demeure; Y. Del Gallo. An array approach for signal processing design. *Sophia-Antipolis Conf. on Micro-Electronics (SAME'98)*, 1998.

[5] G. Gogniat, J. Vidal, L. Ye, J. Crenne, S. Guillet, F. De Lamotte, J.-P. Diguet, and P. Bomel. Self-reconfigurable embedded systems: from modeling to implementation. *Engineering of Reconfigurable Systems and Algorithms, Las Vegas*, 2010.

[6] P. Bomel; J. Crenne; L. Ye; J.-P. Diguet; G. Gogniat. Ultra-fast downloading of partial bitstreams through ethernet. *ARCS*, 2009.

[7] OMG Group. Modeling and analysis of real-time and embedded systems (marte), www.omgmarte.org/, 2007.

[8] M. Pouzet J.-L. Colaço, G. Hamon. Mixing signals and modes in synchronous data-flow systems. *ACM International Conference on Embedded Software*, 10 2006.

[9] C. Cassandras; S. Lafortune. Introduction to discrete event systems. *Kluwer Acad. Publ.*, 1999.

[10] Hervé Marchand, Patricia Bournai, Michel Borgne, and Paul Guernic. Synthesis of discrete-event controllers based on the signal environment. *Discrete Event Dynamic Systems*, 10(4):325–346, Oct 2000.

[11] D. Harel; A. Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Meth.*, 1996.

[12] L. Godard; C. Moy; J. Palicot. An executable meta-model of a hierarchical and distributed architecture management for cognitive radio equipments. *Annals of Telecommunications*, 64(7):463–482, 2009.

[13] Petalinux. http://developer.petalogix.com/.

[14] F. Maraninchi; Y. Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Sci. Comput. Program.*, 46(3), 2003.

[15] A. Gamatie; H. Yu; G. Delaval; E. Rutten. A case study on controller synthesis for data-intensive embedded systems. *Embedded Software and Systems, 2009. ICESS '09. International Conference on*, pages 75 – 82, May 2009.

[16] G. Delaval; E. Rutten. Reactive model-based control of reconfiguration in the fractal component-based model. In *Component-Based Software Engineering*, 2010.

[17] G. Delaval; H. Marchand; E. Rutten. Contracts for modular discrete controller synthesis. In *Proc. of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems, LCTES*, 2010.

[18] O. Labbani; J.-L. Dekeyser; P. Boulet; É. Rutten. Introducing control in the gaspard2 data-parallel metamodel: Synchronous approach. *Int'l workshop on modeling and analysis of real-time and embedded systems*, 2005.

[19] S. Aboubekr; G. Delaval; E. Rutten. A programming language for adaptation control: Case study. In *APRES'09*, 2009.

[20] A. Benveniste; P. Caspi; S. A. Edwards; N. Halbwachs; P. Le Guernic; R. De Simone. The synchronous languages 12 years later. *PROCEEDINGS OF THE IEEE*, 91:64—83, Jan 2003.

[21] TheDaRTProject. Gaspard: Graphical array specification for parallel and distributed computing, http://www2.lifl.fr/west/gaspard/index.html, 2008.

[22] Jorgiano Vidal, Florent de Lamotte, Guy Gogniat, Philippe Soulard, and Jean-Philippe Diguet. A co-design approach for embedded system modeling and code generation with uml and marte. *DATE*, 2009.

[23] Linfeng Ye, Jean-Philippe Diguet, and Guy Gogniat. Modeling of reconfigurable mpsocs for on-demand computing. *GRETSI*, 2009.

[24] H. Yu. A marte-based reactive model for data-parallel intensive processing: Transformation toward the synchronous model. *Ph.D. dissertation, USTL/LIFL*, 2008.