Designing formal reconfiguration control using UML/MARTE

Sébastien Guillet*, Florent de Lamotte*, Nicolas Le Griguer*, Éric Rutten**, Guy Gogniat*, Jean-Philippe Diguet*
*Lab-STICC, France, {sebastien.guillet,florent.lamotte,nicolas.le-griguer,jean-philippe.diguet,guy.gogniat}@univ-ubs.fr
**LIG / INRIA Rhône-Alpes, France, eric.rutten@inria.fr

***Abstract*—This paper presents the first framework to design and synthesize a formal controller managing dynamic reconfiguration, using a Model Driven Engineering methodology base on an extension of UML/MARTE. The implementation technique highlights the combination of hard configuration constraints using weights (*control part*) – ensured statically and fulfilled by the managed system at runtime – and soft constraints (*decision part*) which, given a set of correct and accessible configurations, choose one of them. An application model of an image processing application is presented, then transformed and synthesized to be executed on a Xilinx platform to show how the controller, executed on a Microblaze, manages the hardware reconfigurations.** [1]

I. Introduction

The control of dynamicity in reconfigurable System-on-Chip (SoC) is one of the biggest challenges facing designers of such systems. And the increasing complexity of embedded system designs in general make it even more difficult to design a safe system, which is critical as they tend to be ubiquitious. This increasing complexity calls for both formal methods and high-level specification formalisms with automated transformations towards lower-level descriptions. In this context, the present study offers a modeling approach to enforce reconfiguration constraints in an integrated and automated solution. Specifically, a UML/MARTE [2] model – which is used to specify real time embedded systems – augmented with control information is transformed into a synchronous specification, named BZR, to be synthesized by a formal tool (SIGALI) which performs Discrete Controller Synthesis (DCS). DCS is an automated technique which finds (if it exists) a controller of a system, given its reconfiguration behavior, its controllability and its constraints that should be enforced.

Figure 1 shows the execution model targeted by this approach. Suppose that a system contains a global execution loop, which starts by taking events from the environment. Then these events get processed by a task (*Reconfiguration controller*) which choses the system's configuration. Finally, this configuration order gets dispatched through the system's tasks following its model of computation, and another iteration of the loop can start again. Many systems follow this kind of execution model. For example, this is the case for many image processing applications which perform image transformations and receives events from the environment to adapt their behavior. If a system can be represented using this execution model, then the proposition of this paper can help to design and formally obtain its *Reconfiguration controller* task.

The whole proposition, cf. figure 2, is integrated using a UML modeller, Papyrus, from which MARTE models augmented with control information are specified. These models are transformed into a synchronous program in BZR language, from which a formal method, Discrete Controller Synthesis, can be performed to obtain a correct-by-construction controller that can be integrated into a Xilinx platform project.

After presenting the related work and tools, MARTE control elements are presented, then MARTE transformations are detailed and finally, a controller is synthesized from the mapped formal representation and integrated to be executed on the platform showing a reconfigurable image processing application.

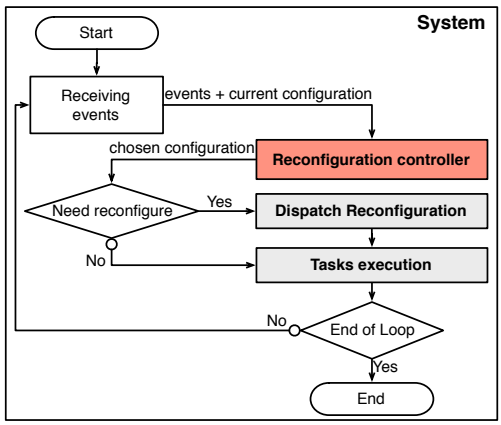

Figure 1. Configuration processing flowchart

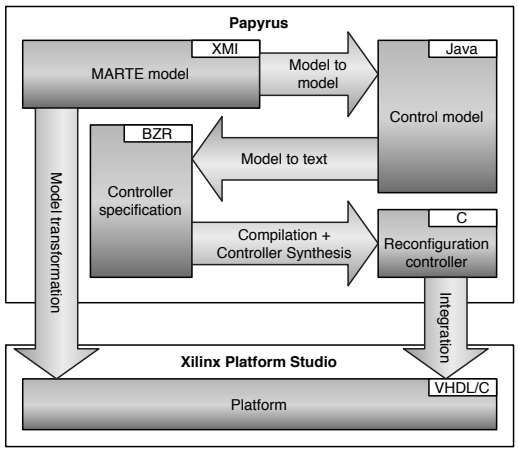

Figure 2. MDE flow

II. Related work

Many research works contribute to the domain of reconfigurable embedded systems, some of them use continuous control techniques. This study is especially interested in those which

[1]This work is supported by the ANR FAMOUS project (ANR-09-SEGI-003)
[2]http://www.omg.org/spec/MARTE/1.1/

care about the closed-loop management of reconfiguration. In [9], an FPGA-based PID motion control system that dynamically adapts the behavior of a robot is presented. Several designs can be swapped, and tradeoff between them are evaluated in terms of area, speed or power consumption. It has to be noted that functional correctness of all the designs is verified by experiment, and not by a formal method. To assure the correctness of the execution of embedded systems, analysis verification and control methods are needed. These methods are often based on model checking, for example in [1] authors use such technique for migration (reconfiguration) of algorithms from hardware to software. Another approach is based on theorem proving, such as [12] that presents a framework for description and verification of parametrized hardware libraries with layout information (explicit symbolic coordinates, neighboring placement). The correctness of the generated layout is established by proof in higher order logic using Isabelle theorem prover. What these studies have in common is that each time, the control system for reconfiguration must be entirely specified by the designer, so that it can be verified or proven afterwards. But a technique from discrete control theory, discussed in the next section, allows for the control design only by giving its constraints. A closer approach to the current proposition is [5] where a formal control technique, based on Discrete Controller Synthesis, is used to control the communications between system-on-chip components by filtering their inputs. But the technique is only applicable in a class of applications in hardware design where input filtering can be safely achieved.

The considered reconfigurable SoCs are a specialization of autonomic computing systems [8], which adapt and reconfigure themselves through the presence of a feedback loop. This loop takes inputs from the environment (e.g. sensors), updates a representation (e.g. Petri nets, automata) of the system under control, and decides to reconfigure the system if necessary. Several works [3] [4] chose to describe these loops in terms of Discrete Controller Synthesis (DCS) problems. It consists in considering on the one hand, the set of possible behaviors of a discrete event system [10], where variables are partitioned into uncontrollable and controllable ones. The uncontrollable variables typically come from the system's environment (i.e. "inputs"), while the values of the controllable variables 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 [14], DCS has been defined and implemented as a tool integrated with the synchronous languages: *Sigali* [11]. 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 handlers, adaptive resource management [4], reconfigurable component-based systems [2], and integrated in a synchronous language, named BZR [13].

In [7], BZR is used to synthesize a controller to manage partial and dynamic reconfiguration. This approach gives the opportunity to the designer to only specify the constraints that must be ensured at runtime so that a corresponding controller, if it exists, can be computed. The controller is *maximally permissive*, it means that every possible decision are presented at each control step, for example several configurations can be allowed in a step, so a decision system must be implemented to chose only one of them. The current study is an improvement of this work on several aspects: it shows how to integrate this decision module, it adds weights constraints, and the concept can be generated using high level models.

III. MARTE CONTROL MODEL

Existing MARTE elements are used to specify the reconfiguration behavior, but low level details about a complete controller specification ready for Discrete Controller Synthesis can not yet be generated because of the lack of an associated semantic in MARTE. We now go into details and show both standard and non standard model elements mandatory for a complete control specification.

In MARTE, Modes represent alternative operational states of a system or component. Mode transition models dynamic operational behavior that represents switching between configurations and changes in components' internal characteristics. Modes and mode transitions are defined in UML StateMachine diagrams, which are stereotyped as *ModeBehavior*. Each mode is represented by a UML State stereotyped as *Mode*, and transitions, describing how modes are linked together, are represented by UML Transitions stereotyped as *ModeTransition* (cf. figure 3).

The MARTE standard provides a mode specific component topology. Such a mode abstraction is an explicitly defined configuration of sub-components, connections, flows, end-to-end flows as well as property values. It is represented by a UML Composite structure specialized by the *Configuration* stereotype. It contains a *Mode* attribute which makes reference to a specific mode state, declared in a *ModeBehavior* (cf. figure 4). When specifying a MARTE *Configuration*, the designer should be able to attach specific properties characterizing the configuration so that he can specify control constraints about them. These properties are named *Weights* and define a value, associated to a type.

Types of weights should then be defined in the MARTE specification. The required metadata is a unique string for each type, so that at transformation time, these names can be connected to a real object type, defining 1) how to combine weights 2) how to compare weights and 3) the neutral weight value to set if none is specified for a configuration. *Inputs* used by *ModeBehaviors* should also be specified and typed in the MARTE model, so that they can be extracted for the *step* function of the controller, which aggregates the set of *Inputs* for all ModeBehaviors. Specific variables, named *Controllables* and assimilable to control points, should also be defined in the MARTE specification, so that they can be used in *ModeBehaviors*. Their values are set by the controller so that temporal properties remain True for all possible executions. Temporal properties, named *Contract* and

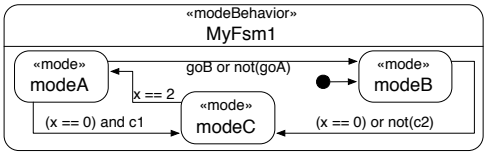

Figure 3. MARTE modeBehavior stereotype

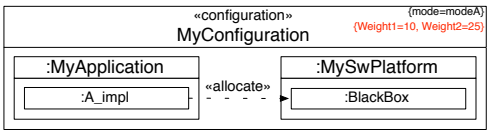

Figure 4. MARTE configuration stereotype with weights proposition

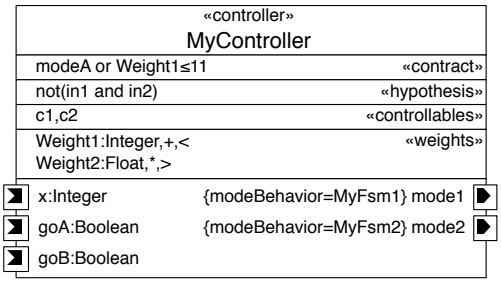

Figure 5. Proposition for a MARTE controller stereotype

Hypothesis are also new element propositions for the MARTE standard. They both define boolean equations. At runtime the hypothesis is supposed to be *true*; and respectively, the contract is guaranteed to be *true* thanks to a controller which will further be obtained through Discrete Controller Synthesis (only if it succeeds, else it means that the contract cannot be fulfilled). *Controllables*, *Types of weights*, *Contract* and *Hypothesis* have been defined in a MARTE extension stereotype, appropriately named *Controller* (cf. figure 5). This controller also shows explicitly the *modeBehavior* it should manage: these can be infered from the stereotype of its *output* ports (mode1, mode2), the *Inputs* ones (x, goA, goB) being processed by the controller and dispatched to the *modeBehaviors*.

IV. MARTE TO SYNCHRONOUS REPRESENTATION

From a MARTE model, containing the previously described standard and non standard elements, several transformations are performed to obtain a Controller specification ready for Discrete Controller Synthesis. This section presents the mapping between such a MARTE model to a BZR program. The synchronous background related to BZR is first recalled, then an example of a MARTE model with control elements is shown with its targeted transformation result, and finally the transformations are formalized.

A. Synchronous background

Definition 1. *Labeled Transition System (LTS):*

A LTS is a tuple $S = \langle \mathcal{Q}, q_0, \mathcal{I}, \mathcal{O}, \mathcal{T} \rangle$ where $\mathcal{Q}$ is is a finite set of states, q_0 is the initial state of S, $\mathcal{I}$ is a finite set of input events (produced by the environment), $\mathcal{O}$ is a finite set of output events (emitted towards the environment), and $\mathcal{T}$ is the transition relation, that is a subset of $\mathcal{Q} \times \text{Bool}(\mathcal{I}) \times \mathcal{O}^* \times \mathcal{Q}$, where $\text{Bool}(\mathcal{I})$ is the set of boolean expressions of $\mathcal{I}$. If we denote by $\mathcal{B}$ the set $\{\text{true}, \text{false}\}$, then a guard $g \in \text{Bool}(\mathcal{I})$ can be equivalently seen as a function from $2^{\mathcal{I}}$ into $\mathcal{B}$.

Each transition has a label of the form g/a, where $g \in \text{Bool}(\mathcal{I})$ must be true for the transition to be taken (g is the guard of the transition), and where $a \in \mathcal{O}^*$ is a conjunction of outputs that are emitted when the transition is taken (a is the action of the transition). State q is the source of the transition (q, g, a, q'), and state q' is the destination. A transition (q, g, a, q') will be graphically represented by $(q \xrightarrow{g,a} q')$.

The composition operator of two LTS put in parallel is the synchronous product, noted $||$, and a characteristic feature of the synchronous languages. The synchronous product is commutative and associative. Formally: $\langle \mathcal{Q}_1, q_{0,1}, \mathcal{I}_1, \mathcal{O}_1, \mathcal{T}_1 \rangle || \langle \mathcal{Q}_2, q_{0,2}, \mathcal{I}_2, \mathcal{O}_2, \mathcal{T}_2 \rangle = \langle \mathcal{Q}_1 \times \mathcal{Q}_2, (q_{0,1}, q_{0,2}), \mathcal{I}_1 \cup \mathcal{I}_2, \mathcal{O}_1 \cup \mathcal{O}_2, \mathcal{T} \rangle$ with $\mathcal{T} = \{((q_1, q_2) \xrightarrow{(\bigwedge_{k=1}^{n} g_k)/(\bigwedge_{k=1}^{n} a_k)} (q'_1, q'_2)) | (q_k \xrightarrow{g_k/a_k} q'_k) \in \mathcal{T}'_k, (q_k, q'_k) \in q \times q'\}$.

Here (q_1, q_2) is called a *macro-state*, where q_1 and q_2 are its two *component states*. A macro-state containing one component state for every LTS synchronously composed in a system S is called a *configuration* of S.

Definition 2. *Discrete Controller Synthesis (DCS) on LTS:*

A system S is specified as a LTS, more precisely as the result of the synchronous composition of several LTS. $\mathcal{F}$ is the objective that the controlled system must fulfill, and $\mathcal{H}$ is the behavior hypothesis on the inputs of S. The controller C obtained with DCS achieves this objective by restraining the transitions of S, that is, by disabling those that would jeopardize the objective $\mathcal{F}$, considering hypothesis $\mathcal{H}$. Both $\mathcal{F}$ and $\mathcal{H}$ are expressed as boolean equations. The set $\mathcal{I}$ of inputs of S is partitioned into two subsets: the set $\mathcal{I}_C$ of controllable inputs and the set $\mathcal{I}_U$ of uncontrollable inputs. Formally, $\mathcal{I} = \mathcal{I}_C \cup \mathcal{I}_U$ and $\mathcal{I}_C \cap \mathcal{I}_U = \emptyset$. As a consequence, a transition guard $g \in \text{Bool}(\mathcal{I}_C \cup \mathcal{I}_U)$ can be seen as a function from $2^{\mathcal{I}_C} \times 2^{\mathcal{I}_U}$ into $\mathcal{B}$. A transition is controllable *if and only if* (iff) there exists at least one valuation of the controllable inputs such that its guard is false; otherwise it is uncontrollable. Formally, a transition $(q, g, a, q') \in \mathcal{T}$ is controllable iff $\exists X \in 2^{\mathcal{I}_C}$ such that $\forall Y \in 2^{\mathcal{I}_U}$, we have $g(X, Y) = false$. In the framework of this document, the following function $S_c = \textit{make\_invariant}(S, E)$ from SIGALI is used to synthesize the controlled system $S_c = S \cap C$ where E is any subset of states of S, possibly specified itself as a predicate on states (or *control objective*) $\mathcal{F}$ and predicate on inputs (or *hypothesis*) $\mathcal{H}$. The function *make_invariant* synthesizes and returns a controllable system S_c, if it exists, such that the controllable transitions leading to states $q_i \notin E$ are inhibited, as well as those leading to states from where a sequence of uncontrollable transitions can lead to such states $q_i \notin E$. If DCS fails, it means that a controller of S does not exist for objective $\mathcal{F}$ and hypothesis $\mathcal{H}$. In this context, the present proposition relies on the use of DCS to synthesise a controller C which makes invariant a safe set of states E in a LTS-based system where E is

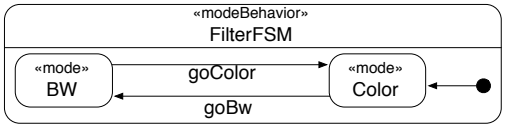

Figure 6. Filter automaton

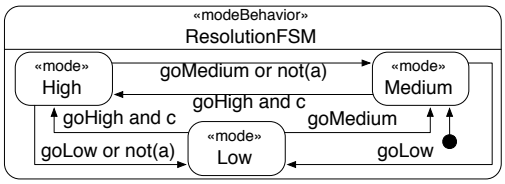

Figure 7. Resolution automaton with controllables variable a and c

inferred by boolean equations defining a control objective and an hypothesis on the inputs. The controller C given by DCS is said to be *maximally permissive* wich means that it doesn't set values of controllable inputs which can be either true or false while still compliant with the control objective. Actually, the BZR compiler defaults these variables to *true* but this type of decision is too arbitrary and the current proposition – which relies on BZR– also proposes a way to integrate a custom decision module, defined by a function interface in C, which can be implemented by the designer. This way, when the controller states that more than one configuration is accessible, this decision module can safely chose one of them to optimize the transitions choices inside E. The actual implementation of such a module goes beyond the scope of this paper so it will be assimilated to a simple random choice.

Definition 3. *Potential transition:*

A transition is potential if the current state is its source and at least one authorized values combination for the controllable variables (if there are any) associated to the uncontrollable input values allows its guard to be evaluated to *true*.

Definition 4. *System equivalence:*

Two systems S and S' sharing the same states and the same initial state are equivalent *with respect to* (wrt) an objective $\mathcal{F}$ and an hypothesis $\mathcal{H}$ they have in common iff for all correct execution wrt $\mathcal{H}$:

- objective $\mathcal{F}$ is guaranteed in both S and S';
- every potential transition for each step of execution in a system is also potential in the other.

Two equivalent systems only differ on the final potential transition choice at each step. This choice being always correct wrt $\mathcal{F}$ and $\mathcal{H}$.

Definition 5. *State and configuration accessibility:*

A state of a LTS is accessible if at least one transition going to it from the current state is potential (which includes by default the case where this state is the current state and no outgoing transition is potential). A configuration, or macro-state of a system, is accessible if each state of its composition is accessible.

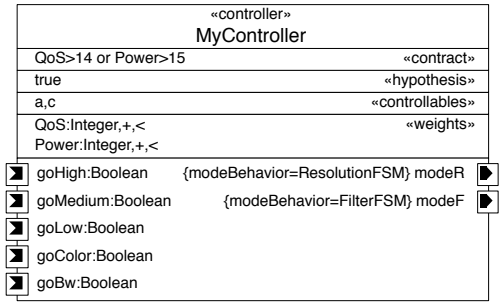

Figure 8. Controller definition

B. Model and transformation example

So as to better understand the equations (coming in the next section) defining the actual transformations, a MARTE example is shown with its transformation into BZR. The example is based on a reconfigurable image processing application. The considered system is composed of two reconfigurable tasks, FilterTask and ResolutionTask, respectively filtering and resizing images from a video stream. Each task has several implementations (IPs), respectively Color, Bw and High, Medium, Low. These implementations are abstracted as MARTE *Configurations* (defined in figure 4), in order to give them the following weight values for two defined weight types named QoS and Power [3]:

	Color	Bw	High	Medium	Low
QoS	5	8	5	10	11
Power	8	7	8	20	17

When a control step is triggered (e.g. by pushing a button), the system receives five events from its environment (eg. from dip switches): *goHigh*, *goMedium*, *goLow*, *goColor* and *goBw*, which are used to take transitions from an implementation to another in both tasks. For example, upon the reception of goMedium, the system is supposed to go from whatever implementation of ResolutionTask to the Medium implementation. The reconfiguration behaviors of FilterTask and ResolutionTask are specified by the MARTE modeBehaviors, shown respectively in figures 6 and 7, each mode being connected to a MARTE configuration as a property. Figure 8 shows a controller which receives several uncontrollable boolean inputs (*goHigh*, *goMedium*, *goLow*, *goColor* and *goBW*) and several controllable boolean inputs (a and c). It will dispatch them to 6 and 7 and output their respective active modes. The hypothesis $\mathcal{H}$ is set to true, meaning that each controllable input combination is correct, and the control objective $\mathcal{F}$ (contract) states that the total weights combination of *QoS* and *Power* for each mode of a configuration should always remain respectively higher than 14 and 15 units for all possible execution/reconfiguration. The weights compositions are defined in the controller specification which states that QoS and Power are Integers and they can be composed using addition on Integers. They can also be compared using the "lower than" operation on Integers, which can be usefull when implementing a decision module. If it exists, a controller obtained through DCS is able to keep the system in states where $\mathcal{F}$ remains *true* for all possible executions by setting the correct values of

[3]Such values could come from profiling or simulation tools for example

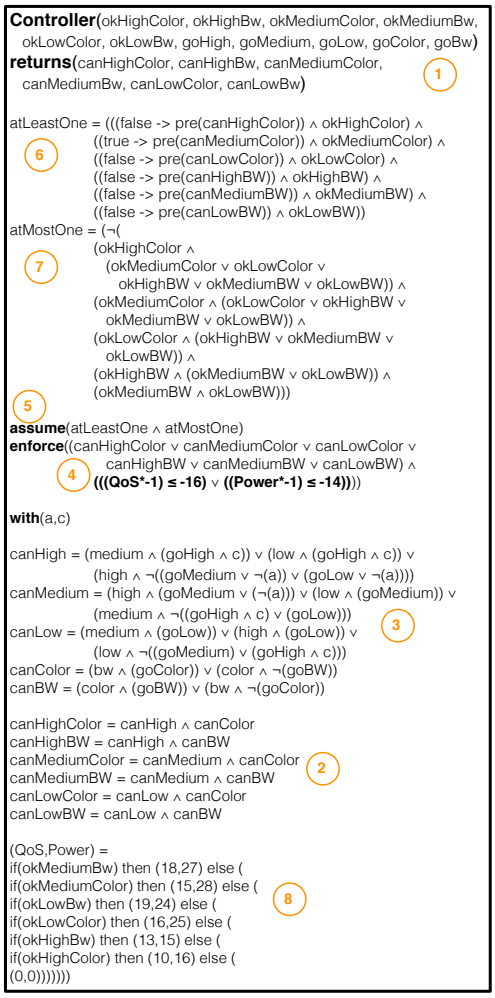

Figure 9. Target transformation to BZR. Circled numbers are references to equations.

a and c which are boolean variables defined as "controllable". These variables are defined in the "controllables" section of the controller specification. They are used to respectively inhibit transitions to High (if c is set to False) and force transitions going out of High (if a is set to False). So in this specification, if nothing has to be controlled to fulfill the control objective, a and c remain *true* and no transition is forced or inhibited.

Figure 9 shows a transformation result of this MARTE specification into BZR using the equations given in the next part. Each circled number in this figure is a reference to a corresponding equation. To avoid redundancies, the transformation of the two modeBehaviors from figures 6 and 7 is shown in figure 11, which gives their graphical representation. This transformation is also explained in the following equations.

V. Decision integration and weight constraints

As can be seen from definition 1, a direct mapping from a MARTE ModeBehavior is trivial: a Mode is a state, a ModeTransition is a transition and a ModeBehavior is a LTS; inputs of the LTS comes from both the *inputs* and *controllables* sections of the controller proposition in MARTE; finally the control objective and hypothesis comes respectively from the *contract* and *hypothesis* sections of the controller specification. But this simple mapping into a LTS with a contract and an hypothesis is not satifying as it does not take advantage of the transition choice when several transitions from a state are possible (ie. when multiple configurations are allowed by the controller). This is why this part shows how to transform such a LTS into one that can be connected to a decision module which can choose between several reconfigurations propositions.

Let $S = \langle \mathcal{Q}, q_0, \mathcal{I}, \mathcal{O}, \mathcal{T} \rangle$ be a system defined as a LTS or a synchronous composition of several LTS, and $\mathcal{F}$ its control objective guaranteed wrt an hypothesis $\mathcal{H}$. The objective is to build a system $S' = \langle \mathcal{Q}, q_0, \mathcal{I}', \mathcal{O}', \mathcal{T}' \rangle$, equivalent to S wrt $\mathcal{F}$ and $\mathcal{H}$ (cf. definition 4), integrating a decision system which makes a choice on accessible configurations at each step.

A. Modifying inputs and outputs

S' encapsulates both the inputs and outputs of S and extends them in order to take into account the configurations accessibility as output and the choice of an accessible configuration as input. Let n be the number of configurations given by all possible combinations of the states of the LTS of S, and m be the number of LTS of S. Let $1 \leq i \leq n$ and $1 \leq j \leq m$. Let $\mathcal{D}$ and $\mathcal{P}$ be two boolean sets, with card($\mathcal{D}$) = card($\mathcal{P}$) = n, where $d_i \in \mathcal{D}$ corresponds to the choice given by the decision system where only one configuration is chosen at each step, formally: $\forall d_i \in \mathcal{D}, \{d_i = false | i \neq x, 1 \leq x \leq n\}$, x is the chosen configuration identifier (id), and $p_i \in \mathcal{P}$ reflects the accessibility of the configuration for which the id is i. At least one configuration must be accessible so formally $\exists p_i \in \mathcal{P}, p_i = true$. The inputs and outputs $\mathcal{I}'$ and $\mathcal{O}'$ of $\mathcal{S}'$ are defined as:

$$\begin{aligned} \mathcal{I}' &= \mathcal{I} \cup \mathcal{D}, \text{with } \mathcal{I} \cap \mathcal{D} = \emptyset \\ \mathcal{O}' &= \mathcal{O} \cup \mathcal{P}, \text{with } \mathcal{O} \cap \mathcal{P} = \emptyset \end{aligned} \quad (1)$$

$\mathcal{I}$ being equal to $\mathcal{I}_C \cup \mathcal{I}_U$, the controllables $\mathcal{I}_C$ can be seen in the *with* part of figure 9. Each configuration accessibility boolean p_i is defined by an equation evaluating the potentiality of all transitions going to each state of the configuration i. This equation is true if all concerned states are accessible. So let $p_i \in \mathcal{P}$, and s_{i_j} be a boolean reflecting the accessibility of a state j of the configuration i. p_i is defined by the following equation:

$$p_i = \bigwedge_{j=1}^{m} s_{i_j} \quad (2)$$

with each s_{i_j} defined by:

$$\begin{aligned} s_{i_j} &= \bigvee \{(g \wedge b_q) | (q \xrightarrow{g/a} q') \in \mathcal{T}\} \\ b_q &= \begin{cases} true & \text{if } q \text{ is the current state} \\ false & \text{otherwise} \end{cases} \end{aligned} \quad (3)$$

To ensure that, at each step, at least one accessible configuration exists wrt $\mathcal{F}$, this property has to be added as a control objective so that DCS can enforce it. Thus, the control objective $\mathcal{F}'$ of $\mathcal{S}'$ is specified by:

$$\mathcal{F}' = \bigvee_{i=1}^{n} p_i \wedge \mathcal{F}, p_i \in \mathcal{P} \quad (4)$$

It has to be noted that if $\mathcal{F}'$ is enforced, then $\mathcal{F}$ is also enforced implicitly for $\mathcal{S}'$, which is important because $\mathcal{S}'$ is supposed to be equivalent to $\mathcal{S}$ wrt $\mathcal{F}$.

B. Correct decision as input

Now that a way to output the next accessible configurations has been specified, processing the final configuration choice (coming as input from a decision module) should be defined. The decision system is seen as a black box from the point of view of DCS, only its inputs ($\mathcal{P}$) and outputs ($\mathcal{D}$) are known. However it is necessary to specify some behavior hypothesis from this system in order to give the following formal information to DCS:

- at least one accessible configuration, given in the previous step, should be given as input (at least one boolean of $\mathcal{D}$ is true);
- at most one configuration is chosen (at most one boolean of $\mathcal{D}$ is true).

The hypothesis $\mathcal{H}'$ of $\mathcal{S}'$ extends the hypothesis $\mathcal{H}$ of $\mathcal{S}$ in order to indicate the previous properties. The *pre* operator, as it is defined in synchronous languages, allows here to refer to the values of configuration accessibility (values of $\mathcal{P}$) at previous step. Formally, let *atLeastOne* and *atMostOne* be two equations defining respectively 1) the fact that at least one accessible configuration is chosen and 2) at most one configuration is given:

$$\mathcal{H}' = \mathcal{H} \wedge atLeastOne \wedge atMostOne \tag{5}$$

Thus, $\mathcal{H}$, *atLeastOne* and *atMostOne* should always be true. Let z be the identifier of the initial configuration, $1 \leq z \leq n$, *atLeastOne* is specified by:

$$\begin{aligned} atLeastOne \quad = \quad & ((true \rightarrow pre(p_z)) \wedge d_z) \vee \\ & (\bigvee((false \rightarrow pre(p_i)) \wedge d_i)), i \neq z \end{aligned} \tag{6}$$

which means that at first step, the initial input given by decision sets d_z to *true*, then for the next steps d_i or d_z should be *true* only when *pre*(p_i) or *pre*(p_z) is *true*. And *atMostOne* is given by:

$$atMostOne = \neg(\bigvee_{x=1}^{n-1}(d_x \wedge \bigvee_{y=x+1}^{n} d_y)) \tag{7}$$

C. Modifying the LTS transitions

In order to finalize a step execution, it is necessary to modify the LTS transitions of $\mathcal{S}'$, so that they react only on occurrence of inputs coming from the decision system (inputs from the specification of $\mathcal{S}$ are entirely processed by the previous equations defining $\mathcal{P}$). The LTS should also be modified in order to output, besides their original outputs, a set of boolean values, named $\mathcal{B}$, allowing to identify their current state, which is needed to evaluate b_q in the specification of a state accessibility.

Let S_k and S'_k (with $1 \leq k \leq m$) be respectively an LTS of S and a transformation of S_k as an LTS of S'. Let $\mathcal{B}_k$ be a set of boolean variables and $n_k = card(\mathcal{B}_k) = card(\mathcal{Q}_k)$. The transformation of each $\mathcal{S}_k$ is specified by:

$$\forall S_k = \langle \mathcal{Q}_k, q_{0_k}, \mathcal{I}_k, \mathcal{O}_k, \mathcal{T}_k \rangle, S'_k = \langle \mathcal{Q}_k, q_{0_k}, \mathcal{D}, \mathcal{O'}_k, \mathcal{T'}_k \rangle$$

with $\mathcal{O'}_k = \mathcal{O}_k \cup \mathcal{B}_k$, $\mathcal{T'}_k \subset (\mathcal{Q} \xrightarrow{Bool(\mathcal{I})/\mathcal{O'}^*_k} \mathcal{Q})$, $\mathcal{O'}^*_k$ being a conjunction of $\mathcal{O'}_k$, and $\mathcal{T'}_k$ being defined by:

$$\mathcal{T'}_k = \{(q_k \xrightarrow{\{\bigvee_{i=1}^{n} d_i \in \mathcal{D} | q_i \in \mathcal{Q}, q'_k \in q_i\}/(a, \bigcup_{j=1}^{n_k} b_{q_{k_j}})} q'_k) | (q_k \xrightarrow{g/a} q'_k) \in \mathcal{T}_k\},$$

$$b_{q_{k_j}} = \begin{cases} true & \text{if } q_{k_j} = q'_k, q_{k_j} \in \mathcal{Q}_k \\ false & \text{otherwise} \end{cases}$$

The result of this transformation is shown on the transitions of the automata from figure 11, which now reacts only on decision inputs (and not original inputs *eg.* $goHigh$, $goColor$, etc.). Finally, in accordance with the synchronous composition principle, transitions $\mathcal{T}'$ of $\mathcal{S}'$ are determined by:

$$\mathcal{T}' = \{(q \xrightarrow{(\bigwedge_{k=1}^{n} g_k)/(\bigwedge_{k=1}^{n} a_k)} q') | (q_k \xrightarrow{g_k/a_k} q'_k) \in \mathcal{T'}_k, (q_k, q'_k) \in q \times q'\}$$

This transformation proposition has shown the way to instrument a equivalent version S' of S wrt definition 4. This version allows the designer to implement its own decision system without interfering with the control objectives as long as it complies with the control interface which is:

- a set of booleans $\mathcal{P}$ for configuration accessibility as input;
- a set of booleans $\mathcal{D}$, containing one and only one final choice (one boolean set to *true*) as output among accessible configurations given at previous step (so there is a direct correspondence between $\mathcal{D}$ and $\mathcal{P}$).

D. Weights computation

To complete the transformations of a MARTE specification into BZR, weights combinations (ie. weights of all configurations) have to be computed. Types of weights appear in the specification, but the actual way to combine and compare them is a technical detail which is implemented directly in the transformations. This way, the designer can use the already implemented types and operations or implement its own by extending the type system as long as they follow the following rules. Let W be the set of types of weights. Each type $w_f \in W$ is associated to a partially ordered group which consists of a group defined by a set V_f together with a binary operation $\star_f$ on this set, a neutral element $\emptyset_f$ (so that when a weight value is not given, the neutral element is used), and a partial order relation $\preceq_f$ on this group. The objective being to compute the combination of the weights values v_{f_k} of each mode q_k of a configuration (for all configurations) so that global weights constraints can be set, variables v_f for each weight type w_f are defined in S' by the following equations:

$$\forall w_f \in \mathcal{W}, v_f = \{\bigstar_{f}^{m}_{k=1} v_{f_k} \mid b_{q_k}\}, \tag{8}$$

$$b_{q_k} = \begin{cases} true & \text{if } q_k \in q', (q \xrightarrow{g/a} q') \in \mathcal{T} \\ false & \text{otherwise} \end{cases}$$

Computation of these global weights values happens offline during the model transformations, so every v_f variable is statically defined. Currently, BZR only propose the order relation "*lower or equal*" on Integers and Floats, so every type and operations defined in the transformations type system must be mappable to pure Float or Integers, and weight constraints (as can be shown in point 4 of figure 9) can only use "*lower or equal*" on these mappings. However, $\star_f$ operations are actually useful when implementing the decision module to make optimizations, as it becomes possible to compare configurations with them. With these mechanisms, a designer can define MARTE models with the control extension to specify a reconfiguration behavior based on weights and states, and can connect a system-specific

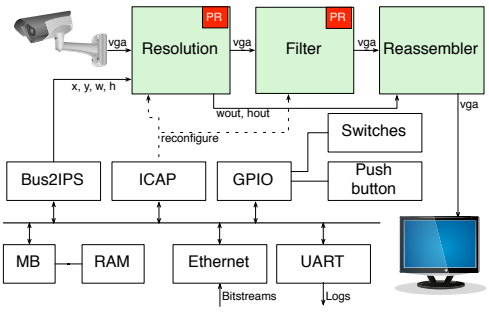

Figure 10. Target platform

decision component. The next section shows the execution of the given example.

VI. Experiment

The generated BZR from figure 9 is compiled and DCS – happening during compilation by calling SIGALI – succeeds, which means that a controller has been found and its code in C is given. The decision module itself is generated by default as a random choice, and all this C code is then deployed on a concrete FPGA-based application system which conforms to the execution model presented in figure 1. The platform is a XUPV5 board which is based on a Xilinx Virtex V FPGA and provides two video ports that we use to capture images from a camera and display them to a screen. The global architecture of the system is presented in figure 10.

A. Architecture

This architecture is composed of two parts. An operative part, which is the video pipeline, implementing the application, and a command part consisting in a microprocessor architecture on which runs the controller. On the video pipeline, the two hardware tasks have been implemented on partially reconfigurable (PR) zones. The first zone receives the resolution IP (High, Medium or Low) and the second receives the filter which either provides a grayscale or color image (respectively with BW and Color IPs). First zone is directly fed by the data from the camera. For the display, a third IP, which is named reassembler, has been developed. This processor writes the current frame on the framebuffer of the video output, ensuring good synchronization. Since it depends on the resolution used – the image size can vary – it adds padding to the image according to a parameter of the resolution IP.

The microprocessor architecture is based on a Microblaze configured through EDK. Control of the video pipeline is done through two channels. The first one is a Bus2IPS IP, used to send parameters to the IPs in the pipeline, in this application, we can change the position and size of the input image in the video stream. The second one is through the ICAP, which can change the configuration of a PR zone. An Ethernet controller is used to download the bitstream from a server [6]. Finally, a GPIO has been synthesized to send events to the controller. To sum up, the controller program reads the request from the GPIO, chooses a configuration for the PR zones, downloads the bitstreams and apply them through the ICAP.

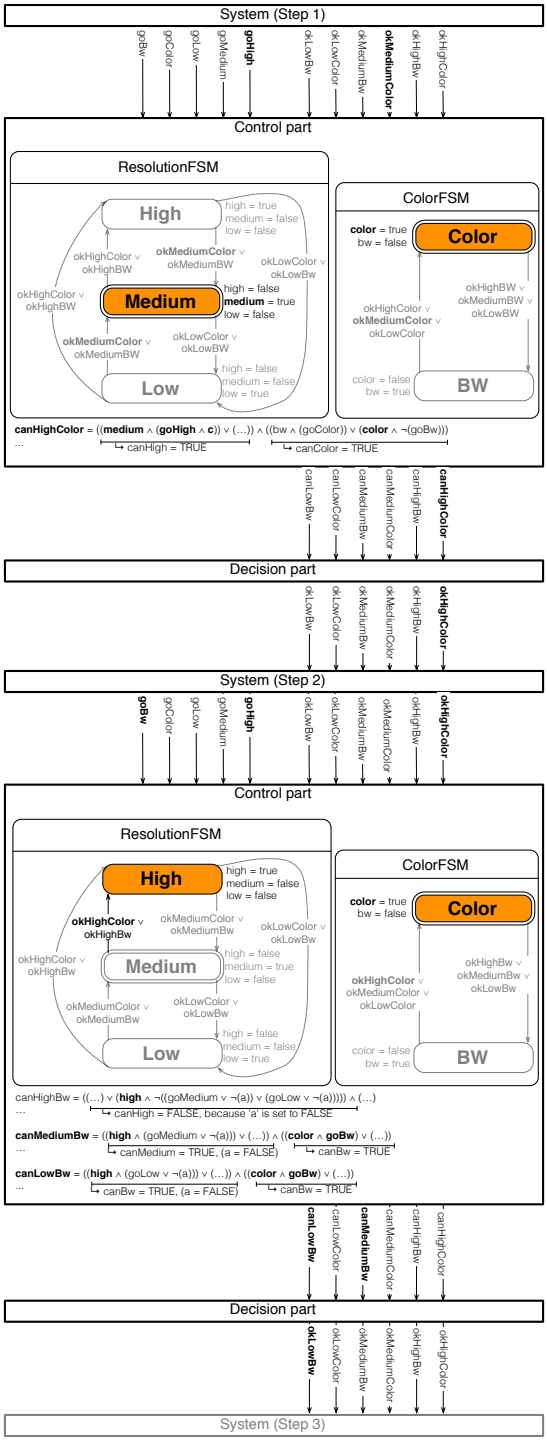

Figure 11. Two first execution steps

B. Execution

Controller steps are triggered by a push button, and the event values are taken from the states of switches, so five switches are dedicated to the events goHigh, goMedium, goLow, goColor and goBw. Two first execution steps of the controller are shown in figure 11 (timeline going from top to bottom).

It should be noted that, to really comply with the MARTE controller specification, the decision module should output mode values the system instead of a boolean for each configuration. Actually a module (also generated) has this role but is not represented here for space reason.

On the first one, the system gives the initial values of the decision which reflects the initial configuration; here okMediumColor is set to true which is correct with respect to the fact that the atLeastOne equation (cf. point 2 of figure 9) requires okMediumColor to be True at the first step, and only this variable is True, which respects the hypothesis defined by the atMostOne equation (cf. point 3 of figure 9). The system also gives the inputs coming from the environment (i.e. the switches values), here only goHigh is arbitrarily set to True.

Given these inputs, the controller takes no transition yet (because it is already in configuration [Medium, Color]) and computes the other equations dedicated to the outputs. From these other equations, only one is True, canHighColor, because given the current configuration and only goHigh from the environment being True, and also because the controller keeps the default values of the c controllable variable to True, the only accessible configuration for the next step is [High, Color]. Given this only possible configuration, the decision process doesn't do much for the present step and just provides back a True value for okHighColor as the identifier of the chosen configuration for the next step.

This choice is then given as a reconfiguration order, performed by the Microblaze which downloads the bitstream High from the bitstream server and reconfigures the downscaler after the current image from the VGA stream is processed.

A second control step can then be prepared by setting new switch states (for example to set goHigh and goBw to True) and pressing the push button. The system sets the new inputs of the controller: the current configuration given by the decision process in the previous step (okHighColor), and the current inputs from the environment given by the switch states.

Given these inputs, the controller takes transitions to the configuration [High, Color] and computes the output equations. But this time, should not go to [High, Bw] even if the inputs goHigh and goBw are True, because it would jeopardize the control objective: indeed, combining the weights of High and Bw provides a QoS of 13 units $(5+8)$ and a Power of 15 units $(8+7)$. The controller is prepared for this situation and autonomously sets the controllable variable a to False, which will force to go out of the state High in the next step. So canHighColor is evaluated to False because a is set to False, which has the border effect of setting canMediumBw and canLowBw to True.

Now given these two possibilities provided by the controller, a decision must be set to choose between them. Here, the decision system retains (remember, this choice is seen as random) the configuration [Low, Bw] by setting okLowBw (and only this variable) to True to the system. And finally, the system propagates the new configuration by downloading and setting the partial bitstreams Low and Bw.

From this example, we understand that the next steps will continue to follow this execution pattern of providing both the current configuration and the switch values to the controller, which will compute the available configurations and let the decision process inform the system about the configuration choice, which performs reconfigurations.

VII. Conclusion and perspectives

This paper presented a way to specify a reconfiguration controller for a DPR SoC in MARTE and synthesize it using the Discrete Controller Synthesis formal technique. To the best of our knowledge, this is the first experiment that introduces DCS to secure configuration transitions in the context of reconfigurable computing.

As usual when speaking about state combination, this approach has its limits due to the state-explosion problem. However, this is still an automated and formal solution to a problem currently solved manually. Controllers obtained through this methodology are guaranteed to always provide a correct configuration to the system, with respect to constraints specified by the designer, for every possible execution, thus freeing the designer to test the system on this critical aspect. This methodology is integrated in a conception flow from where the reconfiguration controller can be designed and transformed into an executable one. The MARTE-based profile used in this study provides a clear definition of control and configurations. The MARTE standard is currently reworked to integrate these modeling and transformation concepts, so that a clear semantic can be used by designers to create standard and consistent models.

As a perspective for the BZR compiler, the integration of the decision module into the control step, instead of being externally defined, would also be great improvement.

References

[1] M. Borgatti, A. Fedeli, U. Rossi, J.-L. Lambert, I. Moussa, F. Fummi, C. Marconcini, and G. Pravadelli. A verification methodology for reconfigurable systems. *Microprocessor Test and Verification, International Workshop on*, 0:85–90, 2004.

[2] Tayeb Bouhadiba, Quentin Sabah, Gwenaël Delaval, and Éric Rutten. Synchronous control of reconfiguration in fractal component-based systems – a case study. *EMSOFT*, 2011.

[3] Fabienne Boyer, Gwenaël Delaval, Noël de Palma, Olivier Gruber, and Eric Rutten. Discrete supervisory control application to computing systems administration. In *INCOM*, 2012.

[4] Gwenaël Delaval and Éric Rutten. Reactive model-based control of reconfiguration in the fractal component-based model. *CBSE*, 2010.

[5] E Dumitrescu, M Ren, L Pietrac, and E Niel. A supervisor implementation approach in discrete controller synthesis. *Emerging Technologies and Factory Automation, 2008. ETFA 2008. IEEE International Conference on*, pages 1433 – 1440, 2008.

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

[7] S. Guillet, F. De Lamotte, É. Rutten, G. Gogniat, and J.-P. Diguet. Modeling and formal control of partial dynamic reconfiguration. *ReConFig*, 2010.

[8] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 2003.

[9] W. Zaho; B. H. Kim. Fpga implementation of closed-loop control system for small-scale robot. *ICAR*, 2005.

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

[11] 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.

[12] O. Pell. Verification of fpga layout generators in higher-order logic. *Journal of automated reasoning*, 2006.

[13] 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.

[14] 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.