Extending UML/MARTE to Support Discrete Controller Synthesis, Application to Reconfigurable Systems-on-Chip Modeling

SÉBASTIEN GUILLET, FLORENT DE LAMOTTE, and NICOLAS LE GRIGUER, Lab-STICC
ÉRIC RUTTEN, INRIA
GUY GOGNIAT and JEAN-PHILIPPE DIGUET, Lab-STICC

This article presents the first framework to design and synthesize a formal controller managing dynamic reconfiguration, using a model-driven engineering methodology based 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 system at runtime—and soft constraints (*decision part*) that, 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.

General Terms: Design, Reliability, Management

Additional Key Words and Phrases: Model-driven engineering, UML/MARTE, discrete controller synthesis, synchronous language, reactive systems, feedback loop, BZR, formal method

ACM Reference Format:
Sébastien Guillet, Florent de Lamotte, Nicolas Le Griguer, Éric Rutten, Guy Gogniat, and Jean-Philippe Diguet. 2014. Extending UML/MARTE to support discrete controller synthesis, application to reconfigurable systems-on-chip modeling. ACM Trans. Reconfig. Technol. Syst. 7, 3, Article 27 (August 2014), 17 pages.
DOI: http://dx.doi.org/10.1145/2629628

1. INTRODUCTION

New dynamically reconfigurable architectures represent a promising opportunity to improve the sensitive issues of heat dissipation and energy efficiency in future Systems-on-Chip (SoC). However, the control of the dynamicity of such systems is one of the biggest challenges with which designers must cope. Moreover, the increasing complexity of SoC makes even more difficult the design of safe embedded systems, which is critical as they tend to be ubiquitous. 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[1] model—that is used to specify real-time embedded

[1]http://www.omg.org/spec/MARTE/1.1/.

This work is supported by the ANR research project FAMOUS. This project is funded by the French National Research Agency under grant ANR-09-SEGI-003.
Authors' addresses: S. Guillet, F. de LaMotte, and N. le Griguer, Lab-STICC, Rue de Saint-Maude, 56100 Lorient, France; E. Rutten, INRIA Rhone-Alpes, UMR 5217-Laboratoire LIG-Maison Jean Kuntzmann-110 av. de la Chimie-Domaine Universitaire de Saint-Martin d' Heres-BP 53-38041 Grenoble cedex 9-France; G. Gogniat and J.-P. Diguet (corresponding author), Lab-STICC, Rue de Saint-Maude, 56100 Lorient, France; email: jean-philippe.diguet@univ-ubs.fr.

DOI: http://dx.doi.org/10.1145/2629628

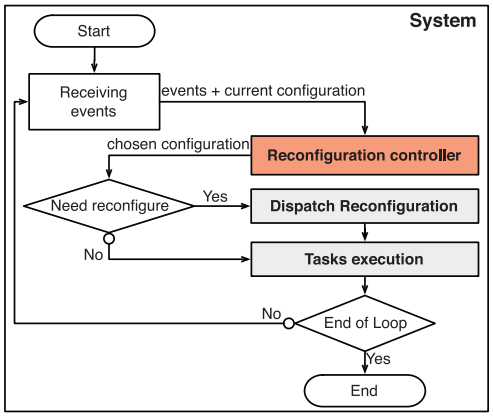

Fig. 1. Configuration processing flowchart.

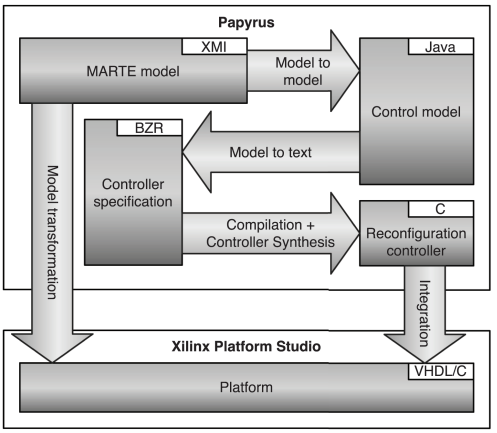

Fig. 2. MDE flow.

systems—is first enhanced with control information and then transformed into a synchronous specification, named BZR, to be synthesized by a formal tool (SIGALI). This latter sebsequently performs Discrete Controller Synthesis (DCS) [Ramadge and Wonham 1989].

Figure 1 shows the execution model targeted by this approach. Suppose that a system contains a global execution loop that starts by taking events from the environment. Then these events get processed by a task (*reconfiguration controller*) that chooses 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 that perform image transformations and receive events from the environment to adapt their behavior. If a system can be represented using this execution model, then the proposition of this work can help to design and formally obtain its *reconfiguration controller* task.

The whole proposition is described in Figure 2 as it is integrated in a UML modeller named Papyrus. This tool is used to specify MARTE models augmented with new control information. The justification of MARTE as the entry point for modeling goes beyond the scope of this contribution. Even if the following methodology is based on MARTE, the modeling language in itself could easily be replaced by another one, as long as the new one also has the ability to define automata and synchronous boolean equations. However, the interested reader could refer to Gamatié et al. [2011] in order to read more about the ability of MARTE to model embedded systems.

Augmented MARTE models are automatically transformed into a synchronous program (in the BZR language), which becomes a defined-by-constraint behavior model. This BZR program is a formal specification on which DCS performs an exhaustive state space exploration to build a correct-by-construction executable controller (in C) that forces the model to remain in a correct set of states with respect to the predefined constraints. This controller can then be integrated into a Xilinx platform project.

It should be noted that the aim of this work is not to replace High-Level Synthesis (HLS)-based methodologies; instead it can be very complementary. Kundu et al. [2011] especially shows the complementarity of HLS and formal verification techniques like model checking. HLS can be seen as stepwise transformation of a high-level design into an RTL design, starting by capturing the behavioral description in an intermediate representation, usually a control dataflow graph. Thereafter, HLS is usually divided

into several subtasks to solve technical problems, typically: allocation, which consists of determining the number of resources that need to be allocated to synthesize the hardware circuit; scheduling, to determine the timestep or the clock cycle in which each operation of the design is executed; resource selection, to determine the resource to execute an operation from several resources of different types and areas and timings; binding and optimization, to map an operation to functional units, variables to registers, and data/control transfers to interconnection components; and control synthesis, which generates a control unit (usually a final state machine) that implements the schedule. The HLS domain has been widely explored and mature HLS tools have emerged such as the Spark framework [Dossis 2011; Gupta et al. 2004] that uses formal transformations to perform HLS.

Now, if a system can be proven technically correct using such methodologies, does it still satisfy a given functional property? This key question is usually tackled by model checking: the properties to be verified are formulated using a formal language and then a model of the system is given and the formal lang-age tests automatically whether this model satisfies the properties for all its possible executions. While model checking requires the definition of a complete system (its behavior) for a given property to be verified, DCS only requires a partial definition based on a notion of controllability, that is, a behavioral specification using controllable variables whose values must be set by a program named the controller. DCS actually goes beyond checking a property: it automatically provides the code of the maximal permissive controller (if it exists) whose role is to constrain the system only when necessary (i.e., to fulfill this property) by appropriately valuating its various controllable variables at runtime. So, in this work, the various technical problems usually solved by HLS are abstracted soas to focus only on functional properties to enforce regarding the compositions of reconfigurable hardware tasks.

This article is organized as follows: 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.

2. RELATED WORK

Much research contributes to the reconfigurable embedded systems domain. Some, like Diguet et al. [2011], use continuous control techniques. This study is especially interested in those that care about the closed-loop management of reconfiguration.

In Zhao et al. [2005], an FPGA-based PID motion control system that dynamically adapts the behavior of a robot is presented. Several designs can be swapped and tradeoffs 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 Borgatti et al. [2004] the authors use such techniques for migration (reconfiguration) of algorithms from hardware to software. Another approach is based on theorem proving, such as Pell [2006] 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 the 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 Dumitrescu et al. [2008] where a formal control technique, based on DCS, 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, and it does not target dynamic reconfiguration problems but communication.

The considered reconfigurable SoCs are a specialization of autonomic computing systems [Kephart and Chess 2003] that 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.

Describing such a loop can be done in terms of a DCS problem. It consists in considering, on the one hand, the set of possible behaviors of a discrete event system [Cassandras and Lafortune 2006], 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 a programming makes use of reconfiguration policy by logical contract. In other words, specifications with contracts amount to specifying declaratively the control objective, and 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, that is, involving only boolean conditions on states and events.

Within the synchronous approach [Benveniste et al. 2003], DCS has been defined and implemented as a tool integrated with the synchronous languages, namely, SIGALI [Marchand et al. 2000]. It handles transition systems with the multi-event labels typical of the synchronous approach, and features weight function mechanisms to introduce some quantitative information and perform optimal DCS.

It has been integrated in a synchronous language named BZR [Delaval et al. 2010]. This language includes a DCS usage from SIGALI within its compilation. BZR is used in this work and its compilation yields a correct-by-construction controller (here in C language) that is then integrated into the reconfigurable platform.

In Guillet et al. [2012], we showed a design flow going from a hypothetical extended MARTE model where control information is extracted and transformed into a BZR program, which is then compiled to produce reconfiguration controllers. MARTE was used over other modeling standards mainly because it is based on UML, is lightweight, and provides higher abstraction levels. More details about the abilities of MARTE in SoC modeling and advantages compared to the other standards and profiles can be found in Quadri [2010]. This article is an improvement over this last work, as it shows both the complete design methodology based on MARTE and also how MARTE has been extended to support control specification for DCS.

3. MARTE CONTROL MODEL

This section is focused on control specification only. It justifies an extension, named RecoMARTE, of the MARTE metamodel for further control synthesis operations.

Let's suppose we have the following models represented in MARTE, from which we will only consider an abstract view (to focus on control aspects): *application model*, *architecture model,* and *allocation model*. The application model is a task graph, each task being a MARTE RtUnit that contains execution semantics (such as input consumption and output production at each call). The architecture model is a set of components (MARTE HwResource) communicating together through ports. And, finally, the allocation represents the mapping of tasks on hardware components (e.g., a task is executed

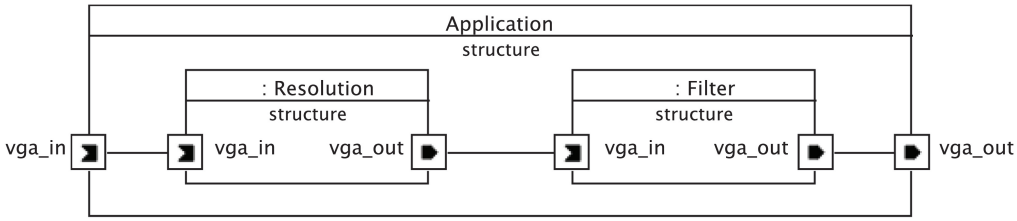

Fig. 3. Application task graph.

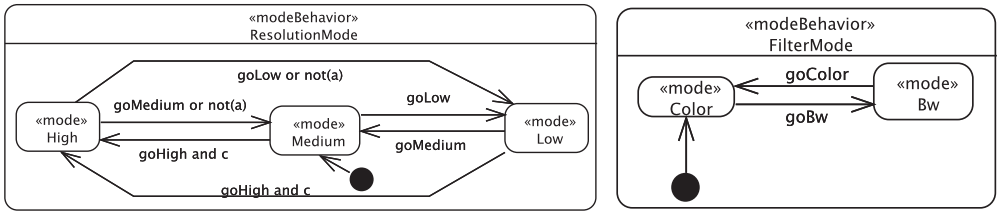

Fig. 4. MARTE ModeBehaviors.

on a processor, meaning that it is a software task; or a task is allocated to a hardware component identified as a black box, meaning that it is a hardware task).

Let's take an example based on a reconfigurable image processing application. The considered system is composed of two reconfigurable hardware tasks, namely filter and resolution, filtering and resizing images from a video stream, respectively (refer to Figure 3). Each task has several implementations (IPs), respectively color, bw, and high, medium, and low. These implementations are abstracted as MARTE *configurations*, which are UML composite structures representing configuration aspects of the system. Such a component is an explicitly defined configuration of subcomponents, connections, flows, end-to-end flows, as well as property values, etc., that must be instantiated when the configuration is *active*.

Each MARTE configuration is linked to a *mode* using its *mode* property. A mode is itself defined in a ModeBehavior, which is basically a UML state machine (states connected by transitions using boolean equations as triggers) declaring an interface (events as inputs and an active mode as output). Figure 4 illustrates their state machine view and Figure 7 shows their component view. For more information about the semantics being used by ModeBehaviors in this methodology, the reader can refer to André et al. [2007] that, among other things, describes how timed events can be used in MARTE for behavior modeling (e.g., state machine modeling) to reflect the synchronous semantics of SyncCharts [André 1996]. When a mode (state) is activated, the configuration defined in a MARTE configuration linked to this mode must be instantiated, which is a way to represent dynamicity in MARTE. Figure 5 shows the case for Configuration HighMode: an implementation named *high* is instanciated on the resolution task when *high* mode is activated.[2]

When specifying a MARTE *configuration*, the designer is able to attach specific non-functional properties—or *weights*—(e.g., *energy consumption*, *quality of service*, etc.) characterizing the configuration using MARTE NFPs, which are basically typed and valued attributes given to a configuration. However, in order to verify constraints using these values, one must be able to combine and compare them, but such a mechanism

[2]The internal representation is just an abstraction here; this information is not needed to solve the control problem.

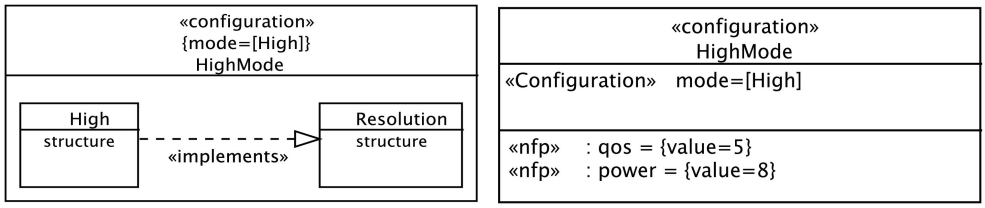

Fig. 5. MARTE configuration HighMode, component view (left) and structural view (right).

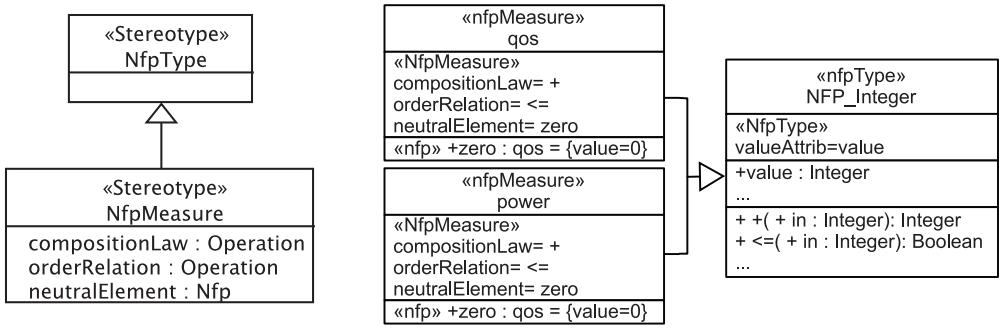

Fig. 6. MARTE extension of NFP concepts (left) and usage example (right).

does not exist in MARTE; thus an extension of the NFP concepts is proposed and illustrated in Figure 6, which shows the extension and its usage in a model to define two weight types (QoS and power). A valuation of these weights for the MARTE configuration HighMode is shown in the structural view of Figure 5. An NfpMeasure extends the notion of NfpType allowing the designer to attach appropriate operations to combine values of a certain type of NfpMeasure, to compare them, and to insert a default value when none has been defined for a given configuration.

Using this NfpMeasure concept, let's say the designer gives the following values[3] of QoS and power for these MARTE configurations:

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

Applying verification techniques to ensure temporal properties regarding several ModeBehaviors—like state invariance[4] or state reachability[5]—implies that these ModeBehaviors can be synchronized, that is, their inputs and outputs are given and emitted at the same discrete instants, respectively. One of these verification techniques, named DCS, also requires that some of their inputs are *controllable*, meaning that their values are given only by a *resolver* that is the function obtained through the application of DCS on such a model. The aim of this function is to set correct values of these controllable variables at each discrete instant so that—combined with all inputs—the

[3]Such values could come from profiling or simulation tools, for example.
[4]The system can be forced to stay in states that comply with the ensured temporal properties.
[5]A state is reachable if a correct path to it (each step of this path being compliant with the ensured temporal properties) exists from the initial state.

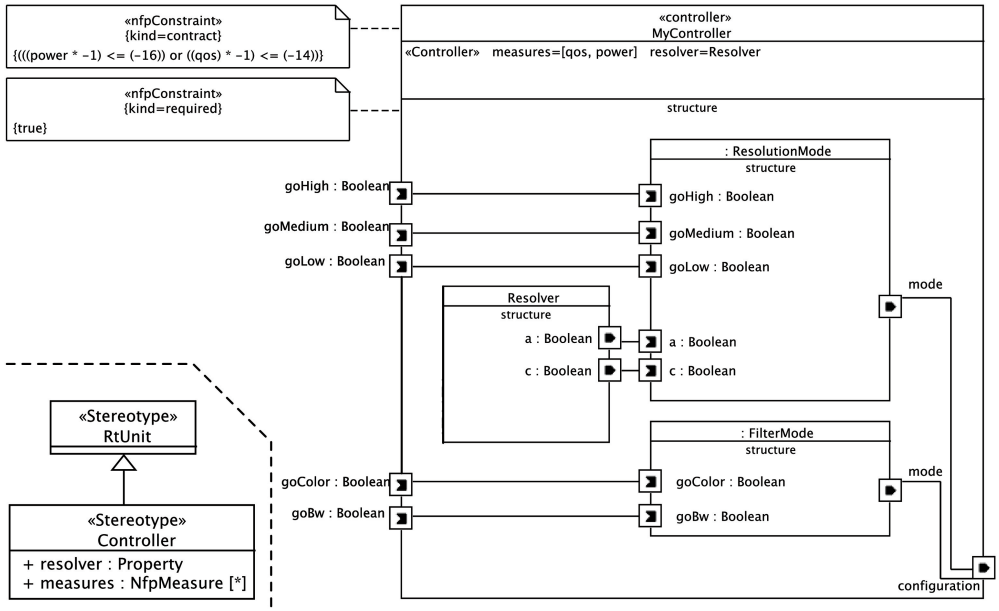

Fig. 7. MARTE RtUnit extension (lower left) and usage example for a controller definition.

system cannot take a transition into a state that would not comply with the specified temporal properties.

Synchronizing ModeBehaviors can be represented in MARTE using an RtUnit that encapsulates them and presents their inputs/outputs on its own interface (synchronizing them by definition). Adding temporal properties to such an RtUnit can still be done in standard using MARTE NfpConstraints that can be of two types: (1) *contract*, representing a boolean constraint to enforce (i.e., to keep *true*) for all executions of the RtUnit, or (2) *required*, representing a boolean constraint supposed to be ensured (i.e., *true*) when using the RtUnit. Identifying controllable variables can be done using the following methodology: the designer adds an internal component into the RtUnit to control and creates a connection from an output port of this component to a ModeBehavior for each controllable variable. This internal component knows both the current state of its encapsulating RtUnit and its input values at every step: it is the *resolver* function that the designer is looking for.

To complete the model for DCS, one must identify which component is the resolver inside a RtUnit to control, as well as the NfpMeasures to take into account if some of them are used in related NFP constraints (temporal properties to assume or to enforce). This cannot be done using the standard. Figure 7 shows the metamodel extension of the RtUnit component, named *controller*, and its usage for a controller definition.

The controller RtUnit (MyController) depicted in this last figure shows that, when it is triggered (e.g., by pushing a button), it receives five events from its environment (e.g., from dip switches): *goHigh*, *goMedium*, *goLow*, *goColor*, and *goBw* that are used to take transitions from an implementation to another in both tasks resolution and filter, whose reconfiguration behaviors are given by the ModeBehaviors ResolutionMode and FilterMode. Controllable events, a and c, are also produced each time the RtUnit is triggered, coming from an internal component named *resolver* that is the actual resolver function interface.

Two NFP constraints are attached to the controller RtUnit, one required and one of type contract, respectively referred as $\mathcal{H}$ and $\mathcal{F}$ in the next section. The hypothesis $\mathcal{H}$ is set to true, meaning that each controllable variable 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 16 units for all possible executions/reconfigurations. These combinations can be computed automatically using the information given in the NfpMeasures definitions: *QoS* and *power* can both be composed and compared using, respectively, the addition and the *lower-than or equal-to* operations on integers.

If it exists, a *resolver* obtained through DCS is able to keep the system in states where $\mathcal{F}$ remains *true* (given the fact that $\mathcal{H}$ is ensured to be true) for all possible executions by setting the correct values of a and c. They are used here 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. The next section shows the transformation of such an extended MARTE model into a synchronous representation based on the BZR language on which DCS will be applied.

4. MARTE TO SYNCHRONOUS REPRESENTATION

From a MARTE model using the previously described extension, several transformations are performed to obtain a controller specification (in BZR) 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.

4.1. Synchronous Background

Definition 4.1 (*Labeled Transition System (LTS)*). An LTS is a tuple $S = \langle \mathcal{Q}, q_0, \mathcal{I}, \mathcal{O}, \mathcal{T} \rangle$, where $\mathcal{Q}$ 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 {true, 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, we have $\langle \mathcal{Q}_1, q_{0,1}, \mathcal{I}_1, \mathcal{O}_1, \mathcal{T}_1 \rangle || \langle \mathcal{Q}_2, q_{0,2}, \mathcal{I}_2, \mathcal{O}_2, \mathcal{T}_2 \rangle = \langle \mathcal{Q}_1 \times \mathcal{Q}_2, (q_{0,1}, q_{0,2}), \mathcal{I}_1 \cup \mathcal{I}_2, \mathcal{O}_1 \cup \mathcal{O}_2, \mathcal{T} \rangle$ with $\mathcal{T} = \{((q_1, q_2) \xrightarrow{(\bigwedge_{k=1}^{n} g_k)/(\bigwedge_{k=1}^{n} a_k)} (q'_1, q'_2)) | (q_k \xrightarrow{g_k/a_k} q'_k) \in \mathcal{T}'_k, (q_k, q'_k) \in q \times q'\}$. Note that this synchronous composition is the simplified one presented in Altisen et al. [2003] and supposes that g and a do not share any variable that would be permitted in synchronous languages like Esterel.

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

Definition 4.2 (*Discrete Controller Synthesis (DCS) on LTS*). A system S is specified as an LTS, more precisely as the result of the synchronous composition of several LTS. $\mathcal{F}$ is the objective that the controlled system must fulfill while $\mathcal{H}$ 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, namely, the set $\mathcal{I}_C$ of controllable variables 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 variables 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 $S_c = make\_invariant(S, E)$ from SIGALI [Marchand et al. 2000] is used to synthesize (i.e., *compute by inference*) 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, this 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 synthesize a controller C that makes invariant a safe set of states E in an LTS-based system where E is inferred by boolean equations defining a control objective and a hypothesis on the inputs. The controller C given by DCS is said to be *maximally permissive*, meaning that it doesn't set values of controllable variables that 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 (that relies on BZR) also proposes a way to integrate a custom decision module, defined by a function interface in C that can be implemented by the designer. This way, when the controller states that more than one configuration is accessible, this decision module can safely choose one of them to optimize the transition choices inside E. The actual implementation of such a module goes beyond the scope of this article so will be assimilated to a simple random choice.

Definition 4.3 (*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 4.4 (*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 a 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'; and
—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 $\mathcal{H}$.

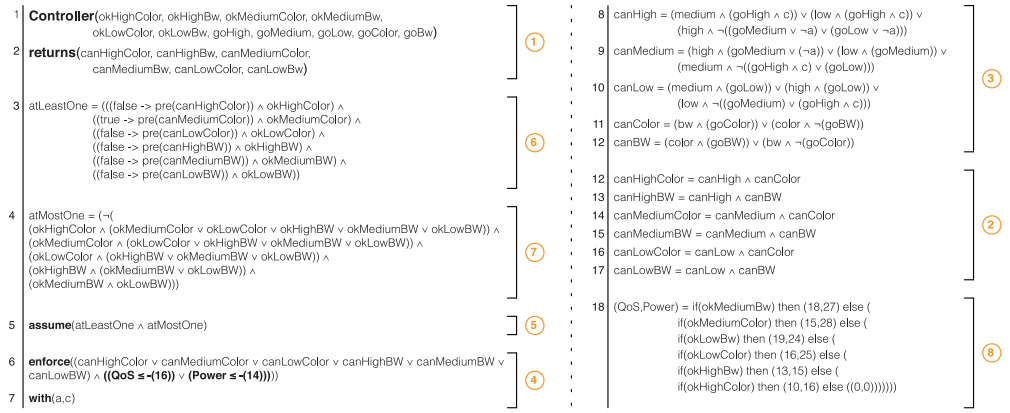

Fig. 8. Target transformation to BZR. Circled numbers are references to equations.

Definition 4.5 (*State and Configuration Accessibility*). A state of an 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 macrostate of a system, is accessible if each state of its composition is accessible.

5. DECISION INTEGRATION AND WEIGHT CONSTRAINTS

Soas to better understand the next equations, Figure 8 shows the transformation result of the previous MARTE model into BZR. Each circled number in this figure is a reference to a corresponding equation. To avoid redundancies, the transformation of the two ModeBehaviors from Figure 4 is shown in Figure 10, which gives their graphical representation. This transformation is also explained in the following equations.

As can be seen from Definition 4.1, a direct mapping from a MARTE ModeBehavior is trivial: a mode is a state, a ModeTransition is a transition, and a ModeBehavior is an LTS; inputs of the LTS come from both the *inputs* and *controllables* sections of the controller proposition in MARTE. Finally, the control objective and hypothesis come respectively from the *contract* and *hypothesis* sections of the controller specification. But this simple mapping into an LTS with a contract and a hypothesis is not satifying as it does not take advantage of the transition choice when several transitions from a state are possible (i.e., when multiple configurations are allowed by the controller). This is why this part shows how to transform such an LTS into one that can be connected to a decision module able choose between several reconfiguration propositions.

Let $S = \langle \mathcal{Q}, q_0, \mathcal{I}, \mathcal{O}, \mathcal{T} \rangle$ be a system defined as an LTS or a synchronous composition of several LTS, and $\mathcal{F}$ its control objective guaranteed wrt a 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 $\mathcal{H}$ (refer to Definition 4.4), integrating a decision system that makes a choice on accessible configurations at each step.

5.1. Modifying Inputs and Outputs

S' encapsulates both the inputs and outputs of S and extends them in order to take into account the configuration's 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 $\text{card}(\mathcal{D}) = \text{card}(\mathcal{P}) = n$, 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: $\forall d_i \in \mathcal{D}, \{d_i = false | i \neq x, 1 \leq x \leq n\}$, (hence all decision inputs are false except for the one reflecting the chosen configuration : d_x), where 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

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

$\mathcal{I}$ being equal to $\mathcal{I}_C \cup \mathcal{I}_U$ and where the controllables $\mathcal{I}_C$ can be seen in the *with* part of Figure 8. 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. Then p_i is defined by

$$p_i = \bigwedge_{j=1}^{m} s_{i_j} \tag{2}$$

with each s_{i_j} defined by

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

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

$$\mathcal{F}' = \bigvee_{i=1}^{n} p_i \wedge \mathcal{F}, p_i \in \mathcal{P}. \tag{4}$$

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 $\mathcal{F}$.

5.2. 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: (1) at least one accessible configuration, given in the previous step, should be given as input (at least one boolean of $\mathcal{D}$ is true); and (2) 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 defined in synchronous languages, allows here to refer to the values of configuration accessibility (values of $\mathcal{P}$) at the 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, $1 = z = n$, *atLeastOne* is specified by

$$atLeastOne = ((true \rightarrow pre(p_z)) \wedge d_z) \vee \left(\bigvee((false \rightarrow pre(p_i)) \wedge d_i)\right), i \neq z, \tag{6}$$

which means that, at the 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

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

5.3. 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 that is needed to evaluate b_q in the specification of a state accessibility.

Let S_k and S'_k (with $1 = k = m$) 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

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

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

The result of this transformation is shown on the transitions of the automata from Figure 10, which now reacts only on decision inputs (and not original inputs e.g., *goHigh*, *goColor*, etc.). Finally, in accordance with the synchronous composition principle, transitions $\mathcal{T}'$ of $\mathcal{S}'$ are determined by

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

This transformation proposition has shown the way to instrument an equivalent version S' of S wrt Definition 4.4. This version allows the designer to implement her own decision system without interfering with the control objectives as long as it complies with the control interface, consisting of:

—a set of booleans $\mathcal{P}$ for configuration accessibility as input; and
—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 the previous step (so there is a direct correspondence between $\mathcal{D}$ and $\mathcal{P}$).

5.4. Weights Computation

To complete the transformations of a MARTE specification into BZR, weights combinations (i.e., weights of all configurations, defined as nonfunctional properties) have to be computed. Types of weights appear in the specification, but the actual way to combine and compare them is a technical detail implemented directly in the transformations.

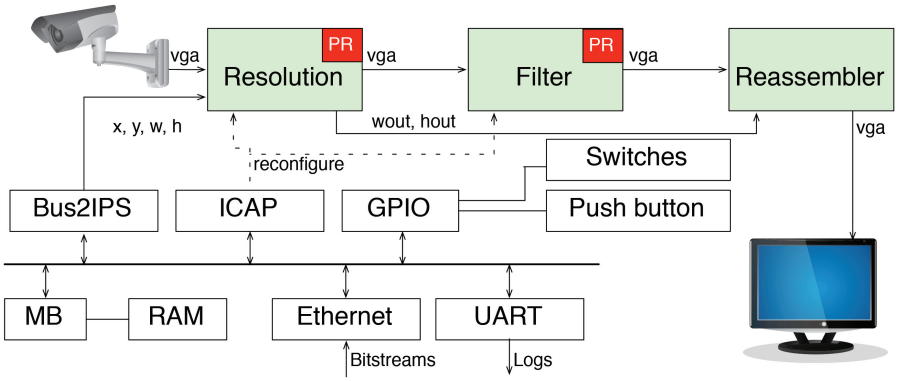

Fig. 9. Target platform.

This way, the designer can use the already implemented types and operations or implement his own by extending the type system as long as he follows the following rules. Let W be the set of types of weights. Each type $w_f \in W$ is associated to a partially ordered 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 weight values v_{f_k} of each mode q_k of a configuration (for all configurations) so that global weight constraints can be set, variables v_f for each weight type w_f are defined in S' by the following equations.

$$\forall w_f \in \mathcal{W}, v_f = \left\{ \bigstar_{k=1}^{m}{}_f \; v_{f_k} \mid b_{q_k} \right\}, b_{q_k} = \begin{cases} true & \text{if } q_k \in q', (q \xrightarrow{g/a} q') \in \mathcal{T} \\ false & \text{otherwise} \end{cases} \tag{8}$$

Computation of these global weight values happens offline during the model transformations, so every v_f variable is statically defined.

Currently, BZR only proposes the order relation "*lower or equal*" on integers and floats, so every type and operation 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 8) 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 decision component. The next section shows the execution of the given example.

6. EXPERIMENT

The generated BZR from Figure 8 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 that conforms to the execution model presented in Figure 1. The platform is a XUPV5 board based on a Xilinx Virtex V FPGA, and provides two video ports that we use to capture images from a camera and display on a screen. The global architecture of the system is presented in Figure 9.

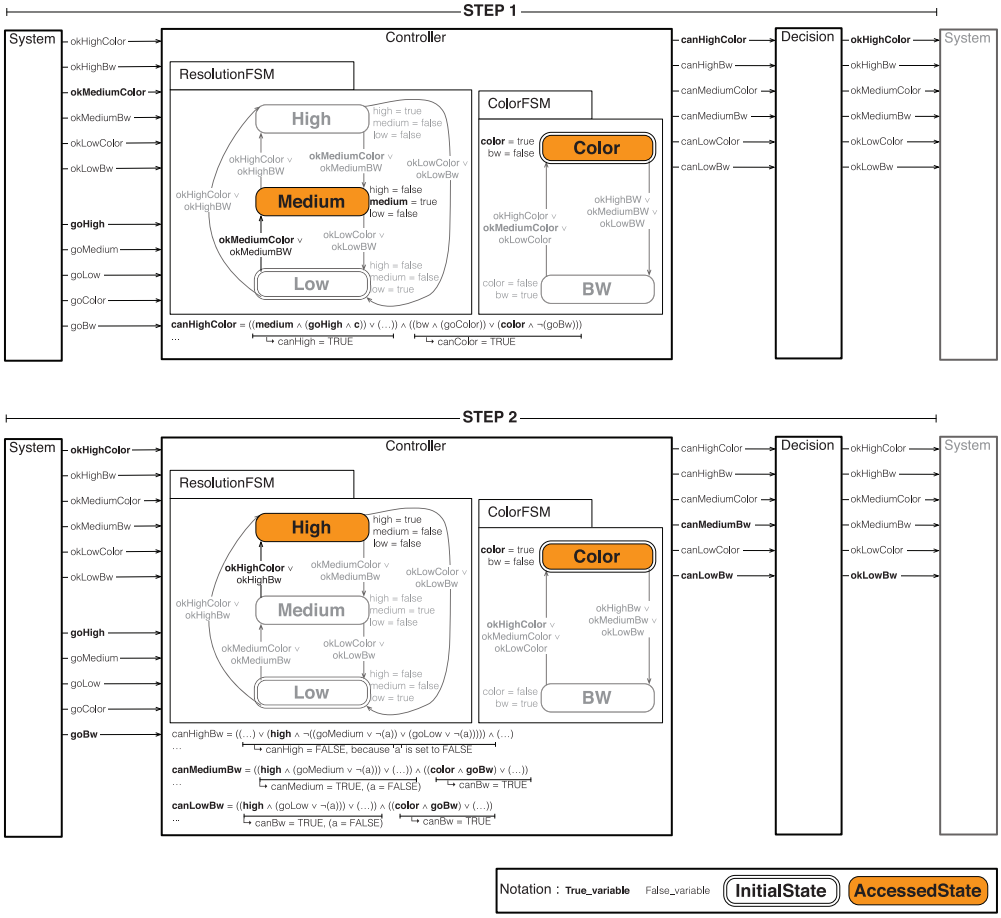

Fig. 10. Two first execution steps.

6.1. Architecture

This architecture is composed of two parts, namely, an operative part that is the video pipeline, implementing the application, and a command part consisting of a microprocessor architecture on which the controller runs.

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 that either provides a grayscale or color image (respectively with bw and color IPs). The first zone is directly fed by the data from the camera. For the display, a third IP named reassembler has been designed. 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 [Bomel et al. 2009]. 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 applies them through the ICAP.

6.2. Execution

Controller steps are triggered by a push button, while the event values are taken from the states of switches, so five switches are dedicated to the events goHigh, goMedium, goLow, goColor, and goBw. The two first execution steps of the controller are shown in Figure 10 (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 for the system instead of a boolean for each configuration. Actually, a module (also generated) has this role but is not represented here for space reasons.

On the first step, the system gives the initial values of the decision reflecting the initial configuration; here okMediumColor is set to true, which is correct with respect to the fact that the atLeastOne equation (refer to point 2 of Figure 8) requires okMediumColor to be true at the first step, and only this variable is true, which respects the hypothesis defined by the atMostOne equation (refer to point 3 of Figure 8). 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 as of yet takes no transition (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, namely, 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 is the 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), as well as 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. However, this time, it should not go to [High, Bw] even if the inputs goHigh and goBw are true, because this 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, thus forcing it 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 that will compute the available configurations and let the decision process inform the system about the configuration choice that performs reconfigurations.

7. CONCLUSION AND PERSPECTIVES

This article presented a way to specify a reconfiguration controller for a DPR SoC—using an extension proposition (RecoMARTE) of the MARTE metamodel—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. The proposed transformations have been implemented using the SDMetrics Open Core[6], and an associated toolbox that can be found online[7] allows to design RecoMARTE models in Papyrus[8] and to transform them into simulable and executable reconfiguration controllers.

As usual when speaking about state combination, this approach has its limits due to the state explosion problem when solving the controller using DCS. State space exploration comes at this price, as does proof of correctness on such a representation. To the best of our knowledge, a less expensive methodology than DCS is not yet availabel to solve the controller of an abstract control model.

Regarding performance, the controller timings are about 0.1 ms (maximum) for less than 10,000 instructions executed in sequence, which is usually far more than what is obtained in reality (especially in this exemple). Actually, only the decision part of the control loop could easily exceed this number, but this reponsibility is up to the designer, not DCS. This timing is almost inisignificant compared to the hardware reconfiguration cost, which is usually between 10 and 20ms, added to the bitstream downloading from the network upon request in our example (20 to 40ms). Moreover, the reconfiguration is coarse grained and this video processing example runs at 24fps, thus letting a fair amount of time for reconfiguration to happen.

This methodology is integrated in a design flow from where the reconfiguration controller can be designed and transformed into an executable one. RecoMARTE, the MARTE-based profile used in this study, provides a clear definition of control and configurations. Including its modeling and transformation concepts into MARTE is an interesting perspective to allow 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 a great improvement.

REFERENCES

K. Altisen, A. Clodic, F. Maraninchi, and E. Rutten. 2003. Using controller-synthesis techniques to build property-enforcing layers. In *Proceedings of the 12th European Conference on Programming (ESOP'03)*. Springer, 174–188.

C. Andre. 1996. Representation and analysis of reactive behaviors: A synchronous approach. In *Proceedings of the Conference on Computational Engineering in Systems Applications (CESA)*. IEEE-SMC, 19–29.

C. Andre, F. Mallet, and R. De Simone. 2007. Time modeling in MARTE. In *Proceedings of the ECSI Forum on Specification and Design Languages (FDL'07)*. 268–273.

A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone. 2003. The synchronous languages 12 years later. *Proc. IEEE* 91, 1, 64–83.

P. Bomel, J. Crenne, L. Ye, J.-P. Diguet, and G. Gogniat. 2009. Ultra-fast downloading of partial bitstreams through ethernet. In *Proceedings of the 22nd International Conference on Architecture of Computing Systems (ARCS'09)*. Springer, 72–83.

[6]http://www.sdmetrics.com/OpenCore.html.

[7]http://www-labsticc.univ-ubs.fr/∼lamotte/ctrlreconf/.

[8]The UML modeling tool Papyrus: http://www.eclipse.org/papyrus/.

M. Borgatti, A. Fedeli, U. Rossi, J.-L. Lambert, I. Moussa, F. Fummi, C. Marconcini, and P. Pravadelli. 2004. A verification methodology for reconfigurable systems. In *Proceedings of the 5th International Workshop on Microprocessor Test and Verification (MTV'04).* 85–90.

C. G. Cassandras. and S. Lafortune. 2006. *Introduction to Discrete Event Systems*. Springer.

G. Delaval, H. Marchand, and E. Rutten. 2010. Contracts for modular discrete controller synthesis. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'10).* ACM Press, New York, 57–66.

J.-P. Diguet, Y. Eustache, and G. Gogniat. 2011. Closed-loop–based self-adaptive hardware/software-embedded systems: Design methodology and smart cam case study. *ACM Trans. Embedd. Comput. Syst.* 10, 3, 38:1–38:28.

M. Dossis. 2011. A formal design framework to generate coprocessors with implementation options. *Int. J. Res. Rev. Comput. Sci.* 2, 4, 929–936.

E. Dumitrescu, M. Ren, L. Pietrac, and E. Niel. 2008. A supervisor implementation approach in discrete controller synthesis. In *Proceedings of the IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'08).* 1433–1440.

A. Gamatie, S. Le Beux, E. Piel, R. Ben Atitallah, A. Etien, P. Marquet, and J.-L. Dekeyser. 2011. A model-driven design framework for massively parallel embedded systems. *ACM Trans. Embedd. Comput. Syst.* 10, 4, 39:1–39:36.

S. Guillet, F. De Lamotte, N. Le Griguer, E. Rutten, G. Gogniat, and J.-P. Diguet. 2012. Designing formal reconfiguration control using UML/MARTE. In *Proceedings of the 7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC'12).* 1–8.

S. Gupta, R. K. Gupta, N. D. Dutt, and A. Nicolau. 2004. Coordinated parallelizing compiler optimizations and high-level synthesis. *ACM Trans. Des. Autom. Electron. Syst.* 9, 4, 441–470.

J. O. Kephart and D. M. Chess. 2003. The vision of autonomic computing. *Comput.* 36, 1, 41–50.

S. Kundu, S. Lerner, and R. Gupta. 2011. *High-Level Verification: Methods and Tools for Verification of System-Level Designs*. Springer.

H. Marchand, P. Bournai, M. L. Borgne, and P. L. Guernic. 2000. Synthesis of discrete-event controllers based on the signalenvironment. *Discr. Event Dynam. Syst.* 10, 4, 325–346.

O. Pell. 2006. Verification of FPGA layout generators in higher-order logic. *J. Autom. Reason.* 37, 117–152.

I. R. Quadri. 2010. MARTE based model driven design methodology for targeting dynamically reconfigurable FPGA based SoCs. Ph.D. thesis, Universite des Sciences et Technologie de Lille 1. 1–252.

P. J. G. Ramadge and W. M. Wonham. 1989. The control of discrete event systems. *Proc. IEEE* 77, 1, 81–98.

W. Zhao, B. H. Kim, A. Larson, and R. Voyles. 2005. Fpga implementation of closed-loop control system for small-scale robot. In *Proceedings of the 12th International Conference on Advanced Robotics (ICAR'05).* 70–77.

Received January 2013; revised February 2014; accepted March 2014