Autonomic Management of Missions and Reconfigurations in FPGA-based Embedded System

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

▶ **To cite this version:**

Soguy Mak-Karé Gueye, Eric Rutten, Jean-Philippe Diguet. Autonomic Management of Missions and Reconfigurations in FPGA-based Embedded System. The 2017 NASA/ESA Conference on Adaptive Hardware and Systems, Jul 2017, Pasadena, California, United States. pp.8. hal-01582518

HAL Id: hal-01582518

https://hal.inria.fr/hal-01582518

Submitted on 6 Sep 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Autonomic Management of Missions and Reconfigurations in FPGA-based Embedded System

Soguy Mak-Karé Gueye and Éric Rutten
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG,
F-38000 Grenoble France
Email: {soguy-mak-kare.gueye,eric.rutten}@inria.fr

Jean-Philippe Diguet
CNRS, Université Bretagne Sud, LAB-STICC,
F-56321 Lorient, France
Email: jean-philippe.diguet@univ-ubs.fr

***Abstract*—Implementing self-adaptive embedded systems, such as UV, involves an offline provisioning of the several implementations of the embedded functionalities with different characteristics in resource usage and performance in order for the system to dynamically adapt itself under uncertainties. FPGA-based architectures offer for support for high flexibility with dynamic reconfiguration features. We propose an autonomic control architecture for self-adaptive and self-reconfigurable FPGA-based embedded systems. The control architecture is structured in three layers: a mission manager, a reconfiguration manager and a scheduling manager. In this paper we focus on the design of the reconfiguration manager. We propose a design approach using automata-based discrete control. It involves reactive programming that provides formal semantics, and discrete controller synthesis from declarative objectives.**

Reactive languages, Space systems and missions, Performance, cost, power, and reliability strategies, Reconfigurable circuits

I. Introduction

Like large-scale distributed systems, embedded systems such as Unmanned Vehicles (UV) are more and more required to be self-adaptive and self-reconfigurable, for resource management, energy efficiency, or by functionality. Many embedded systems, particularly embedded cameras, operate in dynamic, and often unpredictable environments so that a variety of complex tasks is required for a robust behavior of the system. Mission management and embedded intelligence also require online complex tasks. Context-aware and resource-aware adaptation by re-organizing the running tasks can lead to a better utilization of the system resources while retaining and possibly optimizing the performance and processing quality. Field-programmable gate array (FPGA) [(12)] devices are a promising solution for self-adaptive embedded systems. FPGAs allow to reach High Performances with the design of dedicated hardware implementations. Furthermore they also offer flexibility by means of Dynamic Partial Reconfiguration. DPR allows to track the best hardware implementation according to active task requirements. DPR FPGA supports virtually more hardware space for execution than statically available. Offline provisioning of several implementations of the tasks with different characteristics in resource usage (e.g., size and surface used) and performance (e.g., speed, quality) can be envisaged. All the tasks can not be active simultaneously due to area limitations. So when the context changes, adaptation and reconfiguration can be performed to select the appropriate subset of tasks suitable for the context; and run their compatible versions. As a result, the available resources can be optimally utilized under the control of reconfiguration managers which decide online on the moment when to switch, and on the choice of the next configuration to load according to mission requirements.

We propose an autonomic control architecture[1] for self-adaptive and self-reconfigurable FPGA-based embedded systems. The control architecture is layered so that the adaptation and reconfiguration decisions are taken at different levels. The architecture is structured in three layers. The higher-layer consists of a mission manager. The latter is responsible for adapting the system based on uncertainties in the environment. It determines the list of tasks that must be running. The middle-layer consists of a reconfiguration manager which receives from the mission manager the list of tasks to run. The reconfiguration manager is responsible for selecting the tasks implementations that satisfy the execution constraints specified by the mission manager; and that are compatible regarding the resources constraints. The lower-layer consists of a scheduling manager which receives the tasks implementations to run from the reconfiguration manager. The scheduling manager is responsible for processing the sequences of reconfigurations.

In this paper we focus on the design of the reconfiguration manager. Manual programming of such a manager could be error-prone, costly and complex due to the design space, namely the number of possible configurations to consider. Instead, we propose a design approach based on discrete control. The latter provides high level programming languages for formal specification of possible configurations, tools such as Discrete Controller Synthesis; and powerful compilers automatically generating an

[1]This work is partially funded by ANR under project HPeC (2015-2018)

executable implementation in *C*. This approach produces correct-by-construction controllers enforcing desired control objectives, and avoids error-prone manual programming and tedious debugging. We also detail the scheduling layer which executes the sequences of reconfigurations by generating a table encoding the scheduling process based on the tasks implementations to run.

In the rest of the paper, Section II presents DPR in FPGA, Autonomic Computing and reactive languages and tools upon which we base our approach. Section III presents our control architecture in three layers. Section IV details our design approach for a reconfiguration manager. Section V briefly describes the scheduling layer. Section VI presents the design of a reconfiguration manager for controlling the execution of a tracking task. We conclude in Section VII and give directions for future work.

II. Background

A. Dynamic Partial Reconfigurable FPGA

Dynamic Partial Reconfiguration (DPR) is a promising solution for applications that require high performance and high flexibility. It provides a way to modify (part of) the implemented logic in the FPGA at runtime. A dynamic partial reconfiguration consists in loading a bitstream, which contains only the configuration for the target region of the FPGA. The unmodified regions can continue to work without interruption during partial reconfiguration. This enables DPR FPGA to support more available hardware than statically possible. These hardware implementations can be stored in memory and fetched when needed. Hence, multiple applications can run on a single FPGA by sharing hardware resources. Such reconfigurations can be performed at fine-grain, but in this work we consider more coarse-grained configurations compliant with the good ratio between execution and configuration times. We consider tasks like image processing, with execution times comparable to the video frame rate. In such a class of systems, reconfiguration time of about 10 ms is acceptable. This is especially the case when reconfigurations concerning change of image processing are only sporadic e.g., depending on image contents, and occur only every large number of images, making the cost of reconfigurations negligible, or at least manageable, w.r.t. gains.

Research works like [(7),(5)] have focused on the dynamically reconfigurable hardware to meet both performance and cost required in most of embedded systems. They demonstrate how dynamic reconfigurable hardware can be suitable for implementing compute-intensive embedded applications while minimizing the costs. [(7)] experienced sequences of reconfigurations to run a fingerprint recognition application. They show how the reconfiguration overhead can be minimized to avoid performance degradation when performing sequences of reconfigurations. The transfer is done at the maximum throughput (the lowest latency) by using Native Port Interface (NPI) bus specifically adapted to establish a fast link between the external memory and the ICAP primitive. However, they pay less attention on the design of the reconfiguration manager which can, at run-time, choose from several possible configurations, the appropriate one satisfying execution constraints under uncertainties at runtime.

Dynamic reconfiguration requires making decisions about the choice of new configurations, depending on occurring events and sensor values in a system, on past events and sequences history, and on predictive knowledge about possible outcomes of reconfigurations. Such decision components are difficult to design because of the combinatorics of possible choices, the transversal constraints between them to be respected, and even more, the history aspects. We observe that designing and programming the code that manages reconfiguration remains a challenge, and is the object of research. The design of such managers is generally the object of Autonomic Computing [(10)], where self-adaptation of computing systems is managed in a feedback loop. We explore its use in the domain of reconfigurable hardware.

B. Autonomic computing

Autonomic computing [(10)] is a self-management approach proposed by IBM to address the increasing complexity of computing system administration. It consists in providing to a system with the capability of managing itself. An autonomic computing system is able to control and adapt the functioning of its components with no (or less) input from the human administrator. It must be able to self-configure to adapt dynamically to environmental conditions. It must be self-healing and able to find alternate ways to function when it encounters problems such as failures. It must be able to detect threats and self-protect itself from them. It must be able to self-optimize to tune resources to satisfy user demand with the only necessary resources. In order to achieve these functions, it must be able to constantly monitor itself and to act upon itself.

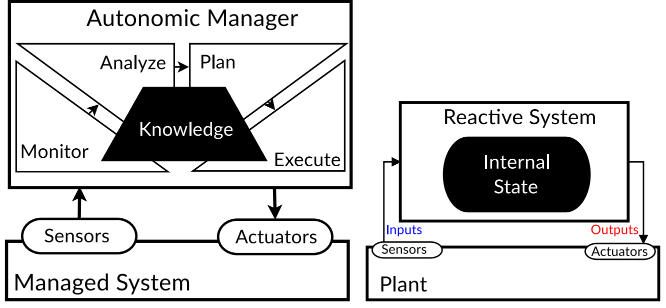

Fig. 1: Autonomic system (MAPEK)

Fig. 2: Reactive system

Figure 1 shows the architecture of an autonomic computing system. A self-managing system must maintain comprehensive knowledge about all its components through sensors. This knowledge guides the decision-making functions to take the appropriate actions to apply

through effectors. The decision-making functions which provides the self-managing capability are called autonomic manager. An autonomic manager is a feedback control loop that collects details from the system and acts accordingly based on the knowledge it has about the managed system. Research has focussed on the design of feedback loops e.g., for dynamic hardware reconfiguration [(11)] and [(9)] for time-varying image constraints for applications in video communications.

An autonomic manager is built as a closed loop. One design methodology to build closed loop is to apply techniques from Control Theory, classically continuous, or Discrete [(4)]. Autonomic managers can be considered as reactive systems, characterized by their continuous interaction with their environment, reacting to flows of inputs (received through sensors) by producing flows of outputs (actions to perform through effectors). So the techniques used to design reactive systems are suited for the design of autonomic managers.

C. A reactive language for automata-based control

We briefly introduce the reactive language and tools used in our approach : Heptagon/BZR (http://bzr.inria.fr/). It supports programming of data-flow equations and automata nodes, with parallel and hierarchical composition, and behavioral contracts [(6)] for *discrete controller synthesis (DCS)*.

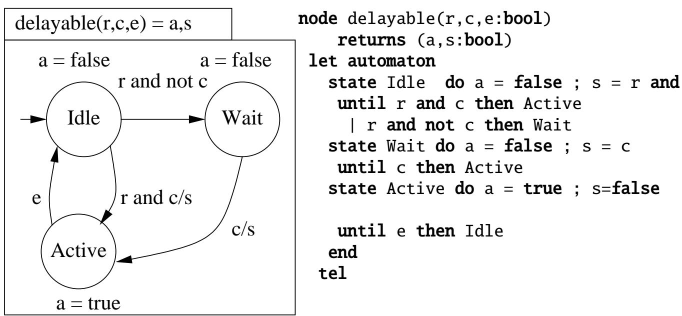

Fig. 3: Graphical Fig. 4: Textual syntax

Fig. 5: Delayable task

Figure 5 shows a small program, that controls a `delayable` task, which can either be idle, waiting or active. It has two outputs: `a` for the task status and `s`, the command triggering the starting operation. In the `Idle` state, the **`true`** value of the input `r` requests the starting of the task which can either directly go to `Active`, or go to a `waiting` state depending on the value of the input `c`. Input `e` notifies termination.

Figure 6(c) shows an example of composition of nodes and behavioral contract. Its body has two instances of the `delayable` node. The contract is declared in three parts. The **`enforce`** part declares the properties to be guaranteed, and the **`with`** part the controllable variables allowing to enforce the properties. The **`assume`** part declares properties about the environment. In the **twotasks** node, the goal is that the two **delayable** tasks are never active simultaneously: **`not`** $(a_1$ **`and`** $a_2)$. The controllable variables are c_1 and c_2. It is assumed that tasks are never requested at the same time: **`not`** $(r_1$ **`and`** $r_2)$. At compilation, DCS synthesizes a control logic which assigns values to the controllable variables such that the goals are guaranteed.

`twotasks`(r_1, e_1, r_2, e_2)
$= a_1, s_1, a_2, s_2$
`assume not` $(r_1$ **`and`** $r_2)$
`enforce not` $(a_1$ **`and`** $a_2)$
`with` c_1, c_2

$(a_1, s_1) =$ `delayable`(r_1, c_1, e_1) ;
$(a_2, s_2) =$ `delayable`(r_2, c_2, e_2)

Fig. 6: Exclusion contract.

The compilation produces an implementation in C or Java in two functions: an initialization function *reset*, and a transition function *step* which takes values of input flows, computes the next state, and returns values of the output flows. In order to integrate the generated implementation we need to implement a mechanism that invokes the *step* with the inputs and applies its outputs. The *step* can be invoked periodically with the available inputs, sporadically (event-driven) based on the occurrence of the inputs, or a mix of both [(3)].

III. Layered control architecture

A. System architecture

The system architecture we address shown in Figure 7. The board is equipped with a dynamically reconfigurable hybrid FPGA (e.g. Xilinx Zynq) including ARM processors. Two DDRAM memories are connected to the FPGA, the first one is usual and implements the ARM memory system. The second one is used to store bitstreams and also implements a shared memory for applications running in software or hardware.

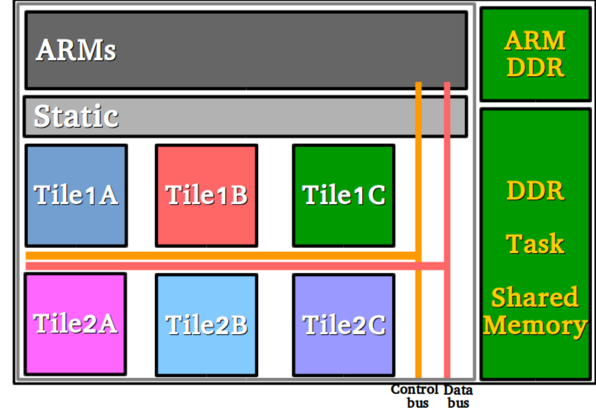

Fig. 7: FPGA device

The FPGA programmable circuit is divided into tiles which will be shared by the tasks at runtime. Sharing the tiles leads naturally to performing sequences of reconfigurations so that all tasks requiring hardware can be executed with an hyper-period. Depending on task timing constraints and priorities, different configurations from a SW version using NEON coprocessor for instance to hardware versions using one or more tiles according to area / performance tradeoff. We consider a mixed data-flow and shared-memory programming model based on coarse-grain functions. A task can be a data dependency and Directed Acyclic Graph (DAG) of functions, each implemented in a distinct bitstream.

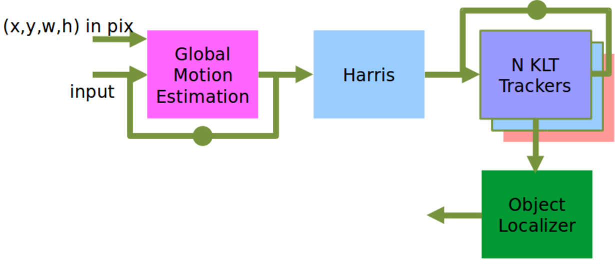

Fig. 8: Example of tracking task

Figure 8 shows an example of a task for tracking a target. It first computes motion estimation, then points of interest. It tracks the selected points to localize the target. The task is composed of four nodes and uses at least four tiles. Depending on the implementations, the nodes can communicate by "pure" data-flow which requires all nodes to be active; or by shared-memory when the tiles must be shared by multiple tasks which involve sequences of reconfigurations of the tiles. In this work a task has multiple implementations with different characteristics.

B. Control architecture

Our control architecture is structured in three layers: mission layer, reconfiguration layer and scheduling layer. In this paper we focus on the design of the reconfiguration layer and the scheduling layer. However we give a description of the mission layer in order to show their interactions. Figure 9 shows the layered control architecture.

The higher-layer is the mission layer where a mission manager takes optimal decision about the mission. The unpredictability of the dynamic of the environment and the system state make difficult to fully and statically specified at design/programming time the evolution of a mission. It is mostly influenced by the uncertainties and the system health. Consequently, the system must be able to properly adapt to them at runtime. In our work, the decisions taken by the mission manager consist in defining the objectives to achieve; and determining the computation tasks with respect to the objectives. The computation tasks with their priorities and their execution constraints (e.g., deadline) are sent to the reconfiguration layer. Stochastic and probabilistics models [(13)] are explored for the design of the mission manager.

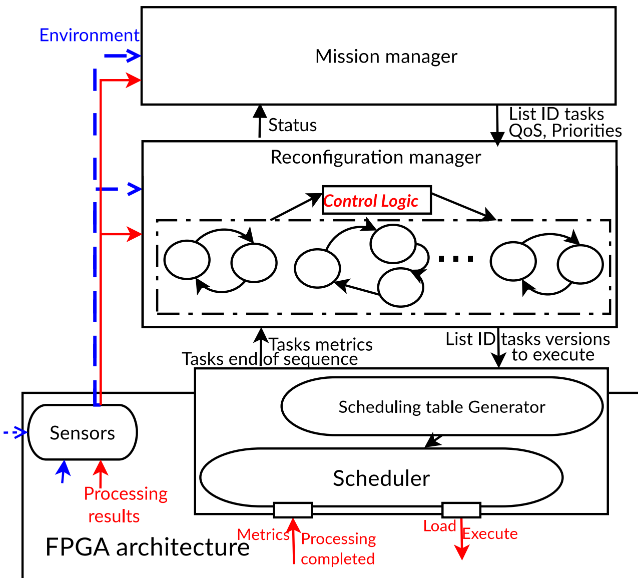

Fig. 9: The three-layered control architecture

The middle-layer represents the reconfiguration layer which is responsible for orchestrating the mapping of the selected tasks to the computing resources. It dynamically adapts the mapping to meet execution requirements and efficient resources sharing. The reconfiguration manager has a global view of the system architecture. It knows the available resources. It also knows all the tasks and their different implementations. Each task implementation has specific characteristics (e.g., required resources) which help choosing at runtime the appropriate one to run when the task is requested. When it receives from the mission layer the set of computation tasks to activate, the reconfiguration manager selects for each task, an implementation that statisfies the requirements and enables to run the other tasks. It constantly monitors the active tasks to detect when execution requirements are violated. In this case, it adapts the mapping by selecting other implementations which satisfy the execution requirements specified by the mission layer.

The lower-layer is the scheduling layer where a scheduler is in charge of running the versions of the tasks to execute. It receives the tasks versions from the reconfiguration layer. It knows the computing resources for each version. Scheduling and executing the versions involve processing reconfigurations.

IV. Reconfiguration layer: automata-based modeling

We detail the design methodology of the reconfiguration manager. It includes behavioral modeling, using automata, of the relevant aspects of the system architecture and the involved tasks which are considered for the reconfigurations control. Earlier works [(8),(2)] experienced the use of

Heptagon/BZR for the design of reconfiguration manager to handle sequences of reconfigurations of a task graph depending mostly on deadline and energy constraints. The reconfiguration manager is generated in C and could be executed in an ARM processor or a small footprint soft core. In this work we focus on the concurrent execution and runtime adaptation of computation tasks to meet performance and processing quality while guaranteeing resource sharing and optimization.

A. Input – output

The reconfiguration manager receives its inputs from both the higher and lower layers. It receives from the higher layer the new subset of tasks to execute and their execution constraints such as quality of service, deadline miss ratio, power peak cap, etc. The lower layer sends to the reconfiguration manager notifications of task termination, tasks metrics, fault, etc. Measures of the processing results are received from the system sensors. The outputs of the reconfiguration manager include commands for the lower layer. The commands can be to start and/or stop a task; or for an active task to switch to a next version. The reconfiguration manager can also sends notifications to the higher-layer. It can notify that there is no available resources or there is no configuration that can meet the execution constraints defined by the higher-layer.

B. Control objectives

They consist in achieving performance constraints as well as resources constraints. These could be conflicting and involve trade-off. The control objectives in the reconfiguration layer consist in finding, regarding the subset of tasks to execute, a coherent configuration that meets performance, processing quality and resource constraints.

C. Models

We define behavioral models of the architecture elements involved in the execution of the tasks. We also define behavioral models of the available tasks and their different execution configurations. These models are defined at the level of abstraction necessary for declaring the control objectives in their composition. The models reflect all possible configurations, and exhibit control points (i.e., input `c`). Discrete controller synthesis is then applied to refine the composition by generating a control logic enforcing the control objectives. The generated control logic restrains in a deterministic way the composition to only the subset of configurations that satisfies the objectives.

We propose generic modeling patterns which can help modeling in a systematic way. Hence, designers who are not familiar with automata and formal languages can still adopt our approach easily if we provide a domain Specific Language (DSL) and a framework that automatically constructs the automata.

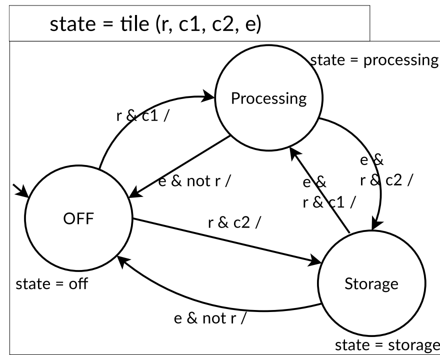

Fig. 10: Reconfigurable tile

1) Reconfigurable tiles: The reconfigurable tiles are modeled in order to control their allocation. Figure 10 shows an example of the model of a tile.

The automaton has three states: `OFF`, `Processing` and `Storage`. Initially a tile is `OFF`. At the occurrence of the input `r` the tile goes to the `Processing` if `c1` is **`true`** or it goes to the `Storage` state if `c2` is **`true`**. The `Processing` state means that the tile is allocated for computations. The `Storage` state means that the tile is allocated as memory. The tile returns back to the `OFF` state at the occurrence of `e` being **`true`**, if **`not`** `r`. The output `state` indicates the current state of the tile. In this model we do not consider failure of a tile. However it could be integrated in the model as well. This is just for illustration, the model can be designed regarding the control objectives.

2) Computation tasks: A task can have multiple implementations with different characteristics, computing resources and bitstreams.

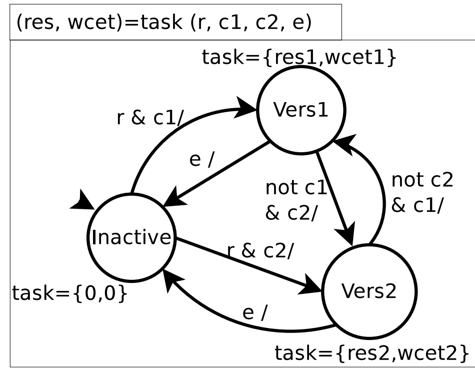

Fig. 11: Computation task

Figure 11 shows an example of the model of a task. The task has two versions (execution configurations): `Vers1` and `Vers2`. They are distinguished by the computing resources `res` they require and their worst case execution time `wcet`. More attributes could be considered but for simplicity we use these two. Initially, the task is inactive, i.e., the state `Inactive`. When its execution is requested through the input `r` depending on the value of the inputs `c1` and `c2`, one of its versions is selected to be executed.

0:TileID	1:FileID	2:Next	3:Load	4:Activate	5:Countdown	6:Count	7:taskID	8:ReplaceBy	9:Starting

Fig. 12: Scheduling Table: a row

	Tile_{ID}	File_{ID}	Next	Load	Activate	Countdown	Count	task_{ID}	ReplaceBy	Starting
row_{ID} 0	3	0	row_{ID} 2	row_{ID} 10	-1	0	1	1	-1	1
row_{ID} 1	0	6	row_{ID} 6	row_{ID} 4	-1	0	1	2	-1	1
row_{ID} 2	1	1	row_{ID} 3	row_{ID} 3	row_{ID} 4	1	1	1	-1	0
row_{ID} 3	1	3	row_{ID} 7	row_{ID} 6	-1	1	1	1	-1	0
row_{ID} 4	0	2	row_{ID} 5	row_{ID} 5	-1	1	1	1	-1	0
row_{ID} 5	0	4	row_{ID} 7	row_{ID} 1	-1	1	1	1	-1	0
row_{ID} 6	3	7	row_{ID} 8	row_{ID} 7	-1	1	1	2	-1	0
row_{ID} 7	1	5	row_{ID} 0	row_{ID} 9	-1	2	2	1	-1	0
row_{ID} 8	2	8	row_{ID} 9	-1	-1	1	1	2	-1	0
row_{ID} 9	1	9	row_{ID} 10	row_{ID} 2	-1	1	1	2	-1	0
row_{ID} 10	3	10	row_{ID} 1	row_{ID} 0	-1	1	1	2	-1	0

Fig. 13: Example of scheduling Table

When `c1` is `True` and `c2` is `False`, `Vers1` is executed. When `c2` is `True` and `c1` is `False`, `Vers2` is executed. When active, the task can switch from `Vers1` to `Vers2` and vice versa depending on the value of the inputs `c1`, `c2`. The task becomes `Inactive` at the occurrence of `e` being `True`.

3) Battery and peripheral devices: In addition to the models for tiles and tasks, more models can be defined, such as models for the battery status (i.e., relevant levels, Figure 14), sensors and effectors status (i.e., available, failure, active); and other embedded devices (e.g., camera and GPS, Figure 15), which might be also relevant.

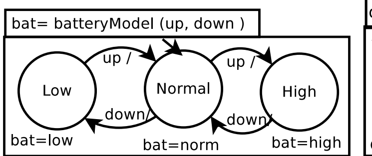

Fig. 14: Battery level

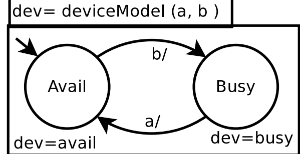

Fig. 15: Device status

V. Scheduling layer

The scheduling layer receives from the reconfiguration layer the list of IDs of the tasks to execute. It executes the selected versions until the reconfiguration layer requests change. The list of IDs corresponds to a set of $DAGs$. The scheduling layer knows for each ID the associated DAG of functions. The execution of the selected $DAGs$ involves sequences of reconfigurations to load and executed the nodes within the $DAGs$. Compared to [(1)], the scheduling layer does not migrate processing nodes. Instead it orchestrates the sharing of the tiles and defines a fixed path to process the nodes within the $DAGs$. It reduces reconfiguration overhead by keeping a node configured if the associated tile is not required by another node. It generates on-line from the received list of IDs a scheduling table which encodes the sequences of reconfigurations to perform. Then it reads the table to load and execute the nodes of the $DAGs$.

Table 12 shows a row of the table. Each row encodes a node of one of the $DAGs$ to execute. The scheduling table has 9 columns. The two first columns contain the mapping of the bitstream of node `n` (`column_1`) and the associated tile (`column_0`). `column_2` contains the row_{ID} encoding the next node to execute after the node `n` which belongs to the same DAG as `n`. `column_3` contains the row_{ID} encoding the next node `m` to allocate the tile used by `n` after its completion. `m` and `n` can belong to the same DAG or to different $DAGs$. `column_4` contains the row_{ID} encoding the node `k` that must be activated at the same time as `n`. `column_4` allows the parallel activation of "son" nodes of a parent node since with this proposed table the parent node can be linked to only one of its "son" nodes through `column_2`. `column_5` allows to know when the node `n` must be executed. Its initial value is the number of dependencies. It is decreased after the activation or completion of a parent node of `n`. `n` is ready to run when the value of `column_5` is `0`. `column_6` contains the number of dependencies of `n`. It allows to reset `column_5`. `Column_7` contains the ID of the task which `n` belongs to. `column_8` contains the row_{ID} in the next table to switch to. It will allow to switch from a table to another without brutal interruption. `column_9` says if node `n` is a starting node of a DAG.

A. Illustration

Figure 16 shows two $DAGs$, each corresponding to a task. The two $DAGs$ share some tiles: `3, 0` and `1`.

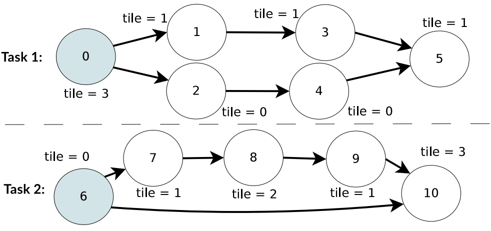

Fig. 16: Two DAGs

Table 13 shows the generated scheduling table encoding the sequences of reconfigurations necessary for the execution of the two $DAGs$. `-1` indicates there is nothing to do. For instance, when the value of `column_3` is `-1`, like in row 8, once loaded the bitstream remains configured in the

tile until the next table. The table supports both "pure" data-flow and shared memory $DAGs$. As said above, it is the responsibility of the reconfiguration layer to select compatible tasks versions.

B. Switching to a new table

When the scheduling layer receives a new list of IDs of tasks to execute, it generates the corresponding table of the new list. To enable the transition, it updates the active table by changing the value of `column_8` of the rows encoding the starting nodes of the $DAGs$. `column_8` has the value `-2` if the DAG must be stopped. Otherwise, it has as value the row_{ID} of the starting node of the new version of the DAG in the new table.

VI. Case study: self-adaptive object tracking application

Drones perform missions such as observing the environment through embedded cameras, detecting then tracking a predefined target. As case study, we consider the control of a tracking task. This task is composed of three computing blocks and their execution involves sequences of reconfigurations of the allocated tiles. The first block consists of Motion Estimation. The second block computes the Points of Interest (Harris). The third block consists of tracking the points of interest. The tracking task is only available on hardware. For the different hardware implementations both the first and the second blocks are fixed. The implementations are differentiated by the number and the window size of the trackers within the third block. Intuitively, It could be necessary to use a large window size for tracking when the speed of the target is high in order to cover a large surface of tracking. This can prevent from loosing the target. Furthermore, it could be necessary to increase the number of trackers when the number of points of interest increases. This allows more parallelism which could allow reducing the computation time for tracking all the points. However increasing window size or/and number of trackers necessitate more computing resources.

A. Modeling and control

1) Inputs: The inputs that trigger the activation (the end) of the tracking are the requests to start (stop) it. The inputs that allow triggering reconfigurations of the tracking task are the speed of the target and the execution time of the current running version; and the value interval of good performance. The interval is delimited by a minimum threshold and a maximum threshold. The speed of the target allows determining when it is necessary to scale the window size of the trackers. The execution time allows to determine if the deadline is respected and how fast the chosen version of the task is regarding the deadline.

2) Automata models: We present the automaton for the tracking task.

It can be seen that Figure 17 is very similar to Figure 11. `Version_1` consumes less computing capacity but has

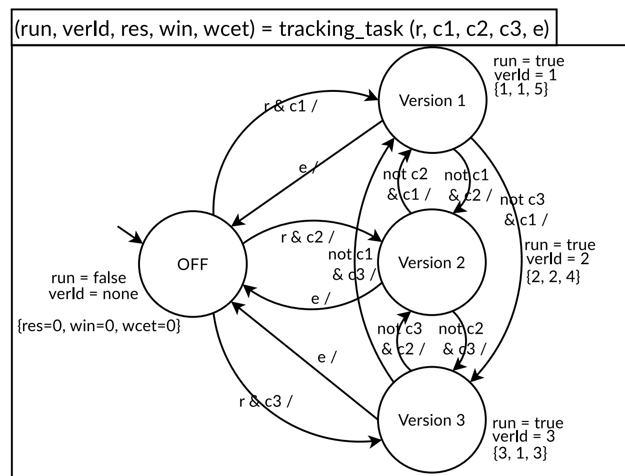

Fig. 17: Automata-based model for the tracking task

higher execution time. `Version_3` has the lowest execution time but consumes higher computing capacity. It has one more state representing a version of the tracking and has more outputs: **`run`** indicates the status of the task (stopped or running), **`verId`** indicates which version is selected for executing and **`win`** represents the window size.

3) Control objectives: The objective is to ensure good performance with the minimum resources. The performance could be evaluated through the execution time and possibly the capacity to track the target. The value interval of the execution time that is considered as good is defined outside the reconfiguration manager (e.g., the mission manager). Once the lower and upper bounds of the value interval are defined, achieving good performance consists in selecting the version of the tracking with the lowest resource capacity for which: $(\texttt{min_thres} < \texttt{time} < \texttt{max_thres})$. When $(\texttt{time} \geqslant \texttt{max_thres})$, a version with shorter execution time must be selected. When $(\texttt{time} \leqslant \texttt{min_thres})$, a version with lower resource capacity could be selected if its execution time is in the range between **`min_thres`** and **`max_thres`**. The window size must also be adapted when the speed of the target scales. The control objectives are expressed in `Heptagon/BZR` as :

- Execution time
  1) $(\texttt{time} \geqslant \texttt{max_thres}) \Rightarrow (\texttt{pre wcet} > \texttt{wcet})$
  2) $(\texttt{time} \leqslant \texttt{min_thres}) \Rightarrow (\texttt{pre wcet} < \texttt{wcet})$
- Window size
  1) $(\texttt{speed} = \texttt{High}) \Rightarrow (\texttt{pre win} < \texttt{win})$
  2) $(\texttt{speed} = \texttt{Low}) \Rightarrow (\texttt{pre win} > \texttt{win})$

We define assumptions of about the value of **`time`**, **`min_thres`** and **`max_thres`**, so that it can know that **`min_thres`** and **`max_thres`** are the lower and upper bounds of an interval: $(\texttt{min_thres} < \texttt{max_thres}) \wedge ((\texttt{time} \geqslant \texttt{max_thres}) \Rightarrow (\texttt{time} > \texttt{min_thres})) \wedge ((\texttt{time} \leqslant \texttt{min_thres}) \Rightarrow (\texttt{time} < \texttt{max_thres}))$

B. Reconfiguration manager behavior

We design a reconfiguration manager for adapting the tracking task. However in order to integrate it into a real system, it is necessary to define the integration code that collects inputs, invokes at the right time the manager `step` and applies its outputs. In this example the right time to invoke a `step` is when the starting (`r`) or the end (`e`) are requested, and the reception of the execution time (`time`). New values for `min_thres` and `max_thres` are kept until the reception of the inputs that trigger a `step`.

We show the behaviors of the reconfiguration manager that automatically adapts the version of the tracking task at runtime. We scale the value of the minimum threshold (i.e., `min_thres`) and the maximum threshold (i.e., `max_thres`) after the activation of the task.

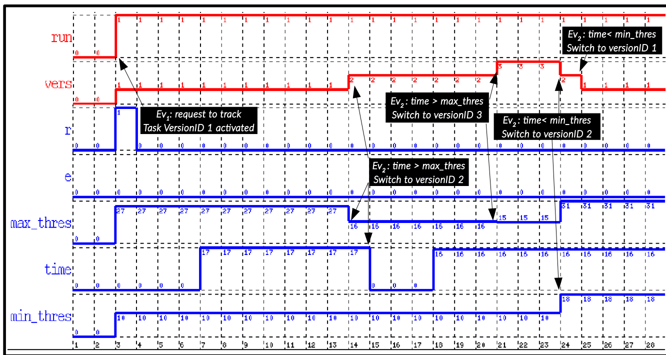

Fig. 18: Behaviors of the reconfiguration manager

In Figure 18, initially the task is stopped (`run=0`, `vers=0`). When the starting of the tracking is requested (`r=1`, `event1`), the version which uses less computing resources is started (`run=1`, `vers=1`). When the execution time (`time`) is greater than `max_thres=16`, the manager switches to `vers=2` (`event2`), then further to `vers=3` (`event3`). When the execution time (`time`) is lower than `min_thres=18`, the manager switches back to `vers=2` (`event4`), then `vers=1` (`event5`). We can conclude that the manager reacts as expected by changing the task version when the execution time is out of bounds.

VII. Conclusions and perspectives

This paper describes an autonomic control architecture for the runtime management of DPR-based hardware reconfiguration, based on behavioral models using automata. We propose a framework defining multiple layers of control and the interactions between them, and a method for systematic modeling of the reconfigurability and configuration space of the target class of systems. We show our design approach through a case study on a video tracking system.

For future work, we are going to implement the case study on a video tracking system on a DPR FPGA, for which the bitstream implementations are ongoing. Moreover we will define a DSL allowing to describe architecture and application and objectives for automatic generation of the automata models and generation of the manager runtime code. We will exploit modularity supported by `Heptagon/BZR` to control complex DPR FPGA architectures for scalability of design space exploration.

References

[1] A. Al-Wattar, S. Areibi, and F. Saffih. Efficient on-line hardware/software task scheduling for dynamic run-time reconfigurable systems. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, pages 401–406, May 2012.

[2] Xin An, Eric Rutten, Jean-Philippe Diguet, and Abdoulaye Gamatié. Model-based design of correct controllers for dynamically reconfigurable architectures. *ACM Transactions on Embedded Computing Systems (TECS)*, 15(3), June 2016.

[3] Nicolas Berthier, Éric Rutten, Noël De Palma, and Soguy Mak-Karé Gueye. Designing Autonomic Management Systems by using Reactive Control Techniques. *IEEE Transactions on Software Engineering*, 42(7):18, July 2016.

[4] Christos G. Cassandras and Stephane Lafortune. *Introduction to Discrete Event Systems.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[5] Edward Chen, Victor Gusev Lesau, Dorian Sabaz, Lesley Shannon, and William A. Gruver. Fpga framework for agent systems using dynamic partial reconfiguration. In *Proceedings of the 5th International Conference on Industrial Applications of Holonic and Multi-agent Systems for Manufacturing*, HoloMAS'11, pages 94–102, Berlin, Heidelberg, 2011. Springer-Verlag.

[6] Gwenaël Delaval, Éric Rutten, and Hervé Marchand. Integrating discrete controller synthesis into a reactive programming language compiler. *Discrete Event Dynamic Systems*, 23(4):385–418, 2013.

[7] Francisco Fons, Mariano Fons, Enrique Cantó, and Mariano López. Real-time embedded systems powered by fpga dynamic partial self-reconfiguration: A case study oriented to biometric recognition applications. *J. Real-Time Image Process.*, 8(3):229–251, September 2013.

[8] Sébastien Guillet, Florent de Lamotte, Nicolas le Griguer, Éric Rutten, Guy Gogniat, and Jean-Philippe Diguet. Extending uml/marte to support discrete controller synthesis, application to reconfigurable systems-on-chip modeling. *ACM Trans. Reconfigurable Technol. Syst.*, 7(3):27:1–27:17, September 2014.

[9] Yuebing Jiang and Marios S. Pattichis. A dynamically reconfigurable architecture system for time-varying image constraints (drastic) for motion jpeg. *Journal of Real-Time Image Processing*, pages 1–17, 2014.

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

[11] Sascha Mühlbach and Andreas Koch. *Reconfigurable Computing: Architectures, Tools and Applications: 7th International Symposium, ARC 2011, Belfast, UK, March 23-25, 2011. Proceedings*, chapter NetStage/DPR: A Self-adaptable FPGA Platform for Application-Level Network Security, pages 328–339. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[12] Wikipedia. Field-programmable gate array — wikipedia, the free encyclopedia, 2016.

[13] Sara Zermani, Catherine Dezan, Chabha Hireche, Reinhardt Euler, and Jean-Philippe Diguet. Embedded context aware diagnosis for a {UAV} soc platform. *Microprocessors and Microsystems*, pages –, 2017.