

A Domain-specific Language for Autonomic Managers in FPGA Reconfigurable Architectures

Soguy Mak-Karé Gueye, Gwenaël Delaval, Eric Rutten, Dominique Heller, Jean-Philippe Diguet

▶ **To cite this version:**

Soguy Mak-Karé Gueye, Gwenaël Delaval, Eric Rutten, Dominique Heller, Jean-Philippe Diguet. A Domain-specific Language for Autonomic Managers in FPGA Reconfigurable Architectures. ICAC 2018 - 15th IEEE International Conference on Autonomic Computing, Sep 2018, Trento, Italy. pp.1-10. hal-01868675

HAL Id: hal-01868675
https://hal.science/hal-01868675

Submitted on 5 Sep 2018

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.

A Domain-specific Language for Autonomic Managers in FPGA Reconfigurable Architectures

Soguy Mak karé Gueye [1], Gwenaël Delaval [1], Éric Rutten [1], Dominique Heller [2], Jean-Philippe Diguet [2]
[1] Univ. Grenoble Alpes, CNRS, Inria, LIG, F-38000 Grenoble France {soguy-mak-kare.gueye, eric.rutten}@inria.fr
[2]Lab-STICC - UMR CNRS 3192, Lorient, France, jean-philippe.diguet@univ-ubs.fr

***Abstract*—Field Programmable Gate Array (FPGA) architectures are suitable hardware platforms for systems that need high performance and flexibility, because they support dynamic partial reconfiguration (DPR) to implement adaptive hardware algorithms e.g., for performance or energy efficiency. They are used for example in embedded systems such as UAV, e.g. for video processing. It is a challenge to design Autonomic Managers for such highly dynamic systems, taking into account the combinatorial design space of configurations and criteria and policies to decide on whether to reconfigure, and what next configuration to choose. In this paper, we propose a Domain Specific Language (DSL) called Ctrl-DPR, allowing designers to easily generate Autonomic Managers. They can describe their system and their management strategies, in terms of the entities composing the system : tasks, versions, applications, ressources, policies. The DSL relies on a behavioural modelling of these entities, targeted at the design of autonomic managers to control the reconfigurations in such a way as to enforce given policies and strategies. The models we use involve automata to describe the state space of configurations, and the transitions representing reconfigurations; they also involve discrete control techniques exploiting such models in order to obtain a correct runtime manager. These model-based control techniques are embedded in a compiler, connected to a reactive language and discrete controller synthesis tool, which enables to generate a C implementation of the controller enforcing the management strategies. We apply our DSL for the management of a video application on a UAV.**

Keywords*—*FPGA, Dynamic Partial Reconfiguration, Domain Specific Language, Reactive Language.

I. Introduction

Embedded systems can benefit from Field Programmable Gate Array (FPGA) architectures to improve performance compared to software-based systems and to fulfill processing space requirements by exploiting the flexibility offered by FPGA. The latter supports dynamic reconfiguration which enables to realize adaptive hardware algorithms e.g., to meet performance or to reduce power consumption. Furthermore, Dynamic Partial Reconfiguration (DPR) improves the flexibility of FPGA by enabling partial reconfiguration of a subfield of the architecture. This can be valuable when the FPGA host mission critical systems that operate in a changing environment, and cannot be disrupted[1] . While parts of the FPGA are being redefined to adapt to uncertainties, the rest remains functioning. FPGA architectures are suitable platforms to design robust and flexible embedded systems, supporting changes of their mission parameters or functions at runtime.

[1]This work was supported by the French ANR and the Labex IMobS3 through the High Performance embedded Computing (HPeC) project.

Changing mission parameters or functions might lead to reorganizing the processings and the allocation of the resources based on their redefined priority and execution requirements. This management can be automated by control loops as addressed in Autonomic Computing [1]. Autonomic Computing has stemmed from distributed and Cloud systems, but is also sometimes considered in reconfigurable hardware architectures [2] as is the case in our present work.

Such control loops can be implemented with low-level programming, but this 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 for Autonomic Managers based on methods and tolls from Control Theory [3], [4]. In particular we consider discrete control [5], [6], where logical properties and control problems are considered. In the context of embedded FPGA-based architectures, this is justified by the fact that configurations are defined by the set of bitstreams uploaded onto the FPGA, and that reconfigurations consist of exchanging bitstreams on part of the FPGA. These characteristics call naturally for models of the kind of Petri nets or Finite State Automata. We us a high level programming language for specification of possible configurations, tools such as Discrete Controller Synthesis, and powerful compilers automatically generating an 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 had previous work and experience on the systematic but manual design of reconfiguration controllers for FPGA using such a reactive language [7], [8], [9].

On the other hand, using such behavioral models can be difficult, because of their technicality and the fact that they do not represent directly the entities of the system under design. Therefore, our contribution in this paper is an automated methodology for the control of dynamic adaptation of FPGA-based self-adaptive embedded systems, in the form of the automatic generation, from a Domain-Specific Language (DSL). We transformed earlier work in the different field of software components [10] and renewed the approach for DPR FPGA. It is targeted at the programming of the control logic, for a class of DPR FPGA architectures and applications. The benefit is manifold: the designer's work is eased by the automatic generation, the latter is done based on formal techniques, from discrete control, insuring correctness of the result.

In the remainder, Section II presents DPR in FPGA, the Autonomic Computing approach and behavioral methods and tools upon which we base our approach. Section III presents

the class of embedded systems we target. Section IV details our domain-specific language. Section V presents the design of an adaptation manager for search landing area 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 since it provides a way to modify (part of) the implemented logic in the FPGA when the device is on. A dynamic partial reconfiguration consists in loading a bitstream which contains only the new logic for the target region of the FPGA. The other regions keep executing their current configuration. This allows an FPGA with DPR capability to support more hardware functions 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.

Research works like [11],[12] have focused on the dynamically reconfigurable hardware to meet both performance and cost required in most of embedded system. They demonstrate how dynamic reconfigurable hardware can be suitable for implementing compute-intensive embedded applications while minimizing the costs. In [11] , the authors 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, these works 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.

Dynamic reconfiguration requires making decisions about whether to reconfigure or not, as well as, when yes, the choice of a new configuration, 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. Therefore there is a need for design methods for such managers of self-adaptations.

B. Autonomic computing

Autonomic computing [1] is a self-management (self-configure, self-heal, self-protect, self-optimize) approach proposed by IBM to address the increasing complexity of computing system administration. It consists in providing a system with the capability of managing itself automatically. An autonomic computing system is able to control and adapt the functioning of its components with no (or less) input from a human administrator. As shown in Figure 1, 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 actuators. The decision-making functions which make the self-managing capability are called autonomic manager. An autonomic manager is a feedback

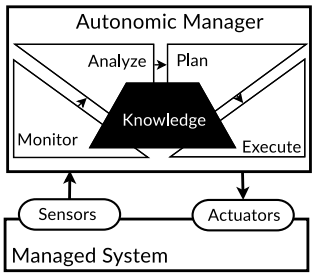


Fig. 1: MAPE-K loop.

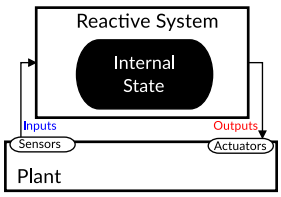


Fig. 2: Reactive system

control loop that collects details from the system and acts accordingly based on the knowledge it has about the managed system. Autonomic Computing has stemmed essentially from distributed and Cloud systems, but is also sometimes considered in reconfigurable hardware architectures [2] as is the case in our present work.

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.

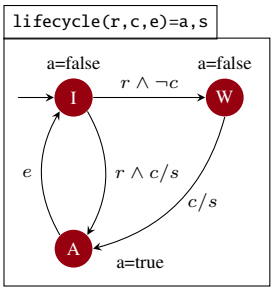
C. Control techniques for autonomic loops

An autonomic manager is built as a closed loop and one design methodology to build closed loop is to apply techniques from Control Theory, classically continuous, or discrete [13]. Indeed, control theory is the classical discipline for the design of automatic controllers of devices, with the advantage of offering interesting properties on the resulting behavior of the controlled system i.e., on its possible evolutions and on those which will be insured to be avoided. Control theory and techniques have began to be used for computing systems, which are quite different from the usual electro-mechanical systems. In most of the cases, continuous models are used, typically for quantitative aspects [14].

More recently, there has been work relying on Discrete Event Systems (DES) [13], and using the notions of supervisory control, for logical or synchronization purposes [15]. Essentially, discrete control uses models like Petri nets or automata, with uncontrollable and Boolean controllable variables ; objectives are logical properties e.g., invariance of a set of good states, or reachability of goal states. Algorithms for Discrete Controller Synthesis automatically explore the state space, and extract the constraint on controllable variables such that the resulting behaviors satisfy the objective properties.

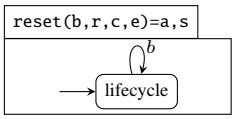
D. Reactive Systems and languages

For such reactive systems, languages have been proposed to describe systems that at each reaction perform a step taking input flows, computing transitions, updating states, triggering actions, emitting output flows [16]. Their definition is often based on Finite State Automata (FSA), which constitute the basic formalism for representing behaviours, as is the case of StateCharts [17] and of synchronous languages.




```
node lifecycle(r,c,e:bool)
      returns(a,s:bool)
let automaton
  state I do
   a=false;s=r & c
   until r & c then A |
         r & not c then W
  state W do
   a=false;s=c
   until c then A
  state A do
   a=true;s=false
   until e then I
 end; tel
```

Fig. 3: Graphical and Textual Representation of Component Life-cycle.




```
node reset(b,r,c,e:bool)
    returns(a,s:bool)
let automaton
  state H do (a,s)=lifecycle(r,c,e)
        until b then H
  end; tel
```

Fig. 4: Example of Hierarchical Composition.

1) Heptagon: Heptagon/BZR [18] is an example of such languages. It allows the definition of reactive systems by means of generalized Moore machines, i.e., with mixed synchronous data-flow equations and automata. An Heptagon program is modularly structured with a set of *nodes*. Each node corresponds to a reactive behaviour that takes as input and produces as output a set of stream values. The body of a node consists of a set of declarations that take the form of either automata or equations. The equations determine the values for each output, in terms of expressions on inputs' instantaneous values or other flows values. Figure 3 shows an Heptagon program in both graphical and textual representations. The program describes the control of a component's life-cycle that can be in either idle (*I*), waiting (*W*) or active (*A*) states. The program takes as input three boolean variables: `r`, which represents a request signal for the component; `c`, which represents an external condition (to be used later on as controllable variable); and `e`, to represent an end signal. It produces as output two boolean values, one that indicates whether the component is active (`a`) the another indicating a start action (`s`). When in the initial state, upon a request signal (i.e., when r is true), the automaton leads to either waiting or active states, depending whether the condition `c` holds. If it does not, it goes first to the waiting state and then to active when `c` becomes true. All the incoming transitions arriving at active state triggers the start action (`s`). From active state, it goes back to idle state upon an end signal.

One important characteristic of Heptagon/BZR is the support for hierarchical and parallel automata composition. Figure 4 illustrates an example of hierarchical composition, in which a single-stated super-automaton embodies the *lifecycle* automaton. It has a self-transition that results in the resetting of the containing automata (i.e., *lifecycle*) at every occurrence of signal *b*. Listing 1 illustrates the parallel composition of two instances of the *delayable* node (and the operator ';'). They run in parallel, in a synchronous way, meaning that one global step corresponds to one local step for every node.

2) Contracts and Discrete Controller Synthesis: BZR is an extension of Heptagon with specific constructs for Discrete Controller Synthesis (DCS). That makes Heptagon/BZR distinguishable since its compilation may involve formal tools such as Reax [19] for DCS purposes. A DCS consists in automatically generating a controller capable of acting on the original program to control input variables such that a given temporal property is enforced. In Heptagon/BZR, DCS is achieved by associating a *contract* to a node. A contract is itself a program with two outputs: e_A, an assumption on the node environment; and e_G, a property to be enforced by the node. A set $\{c_1, c_2, \ldots, c_q\}$ of local controllable variables is used for ensuring this objective. Putting it differently, the contract means that the node will be controlled by giving values to $\{c_1, \ldots, c_q\}$ such that given any input flow satisfying assumption e_A, the output will always satisfy goal e_G. When a contract has no controllable variables specified, a verification that e_G is satisfied in the reachable state space is performed by model checking, even if no controller is generated.

```
1 node twocomponents(r1,r2,e1,e2:bool) returns (a1,a2,s1,s2:bool)
2 contract
3 assume true
4 enforce not(a1 and a2)
5 with (c1,c2)
6 let
7  (a1,s1)=lifecycle(r1,c1,e1);(a2,s2)=lifecycle(r2,c2,e2)
8 tel
```

Listing 1: Example of Contract in Heptagon/BZR.

Listing 1 shows an example of contract on a node enclosing a parallel composition of two instances of *lifecycle* (cf. Figure 3). It is composed of three blocks. The *assume* block (line 3), which in this case, states that there is no assumption on the environment (i.e., $e_A = true$). The *enforce* block (line 4) describes the control objective : $e_G = \neg(a1 \wedge a2)$, meaning that both components are mutually exclusive, i.e., they cannot be active at the same time. Lastly, the *with* block (line 5) defines two controllable variables that are used within the node (line 7) : In practice they will be given values such that variables `a1` and `a2` are never both true at the same instant.

3) Compilation and code generation: The Heptagon/BZR compilation chain is as follows: from source code, the Heptagon/BZR compiler produces as output a sequential code in a general-purpose programming language (e.g., Java or C) implementing the control logic, in the form of a step function to be called at each decision in the autonomic loop. At the same time, if the code provided as input contains any *contract*, the compiler will also generate an intermediary code that will be given as input to the model checker (e.g., Sigali or Reax), which will, in turn, perform the DCS and produce as output an Heptagon/BZR code corresponding to the generated controller. The latter is then compiled again so as to have an executable code also for the generated controller.

III. Target Class of Reconfigurable Systems

Before defining our Domain Specific Language, we are describing the domain which we are targeting : the particularities of FPGA architectures that we are considering, at a given level of abstraction, as well as the structure of the programs running on these architectures, composed of tasks with different versions (software or hardware). We also describe how the

Autonomic Manager controlling reconfigurations is integrated in the general architecture of the system.

A. DPR FPGA architecture and configurations

The system architecture we address is a board equipped with a dynamically reconfigurable hybrid FPGA (e.g. Altera, Xilinx Zynq) including ARM processors. DDRAM memories are connected to the FPGA for ARM memory system and also storing the bitstreams.

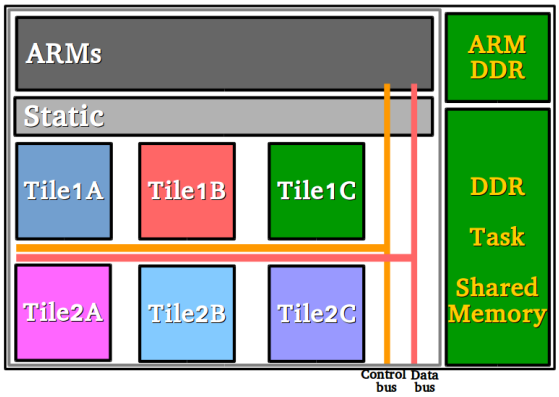


Fig. 5: FPGA device

Figure 5 shows an example of an FPGA. 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 as shared memory for hardware and software tasks. The FPGA programmable circuit is divided into tiles which will be shared by the tasks at runtime. The sharing leads to perform sequences of reconfigurations so that all tasks requiring hardware can be executed.

B. Processings

The system is provisioned offline with all the required tasks, in the form of their bistream implementations, even if all can not be running simultaneously due to area limitations, as shown in Figure 6.

1) Application: At any given moment in time, the application level determines the subset of tasks to be activated : this can change due to changes in the sequential application definition and its goals, the results of the current processings, the architecture state changes (e.g. one tiles becoming unavailable because of a fault) or the environment conditions to which the must react (e.g. by changing functionalities).

2) Tasks: Furthermore, a given activated task may have multiple versions, software (executed on the ARMs) and/or hardware (executed on the FPGA), which differ in terms of used computing resources (e.g. number of tiles), as well as provided performance and processing quality. The version of an active task to activate must be chosen based on the execution requirements and the priority level associated with the task.

One possible example of a scenario is that, when the system mission requires a new task to be activated, some of the already active other tasks may have to be reconfigured to other versions, either requiring less hardware resource (tiles) or

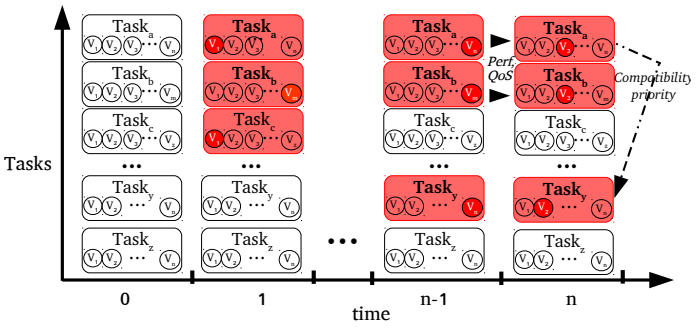


Fig. 6: Reconfigurations: Applications (subset of active tasks), Task (active version)

software (to free the tiles completely), in order to make place for the new one, while still having to insure their required quality of service.

C. Policies

The architectures we consider hence have a space of possible configurations, all presenting dofferent characteristics, providing for flexibility in the realizations of the system fucntionalities. The reconfigurations constitute a navigation across this configurations space, and allow for dynamically reacting to uncertainties from the environment or the platform, in order to maintain overall objectives. These latter have to be described as a part of the specification, and it will be the task of the Autonomic Manager to enforce them, using the controllability choices of the system.

These policies typically state that the tasks required by the application have to be active, and that their current implementation should fulfill the given requirements in performance (e.g. response time) and quality of service (e.g. tracknig quality in a video processing task). These parameters might be subject to dynamical change, implying that the manager performs reconfigurations in order to satisy the new values.

Other policy elements can involve energy management e.g., choosing the configuration using the minimal number of tiles in order to swith off (e.g., by clock gating) the unused ones. On the other hand, in case of urgency determined by the mission level, this economy policy might be suspended, and processings optimized in quality whatever energy costs.

D. Control loop

The control loop, involving the adaptation manager which we want to desing, is responsible for dynamically adapting the configuration of the system. This involves the processing resources, the active tasks and the applications. The processing resources are reconfigured to reduce the energy consumption or to enhance the performance of the active tasks in order to meet their execution requirements.

As shown in Figure 7, the management decisions are based on the execution requirements associated with the active tasks and their priority. The manager monitors the system to detect violation and reacts. It receives the subset of tasks that must be running as well as their execution requirements and priority.

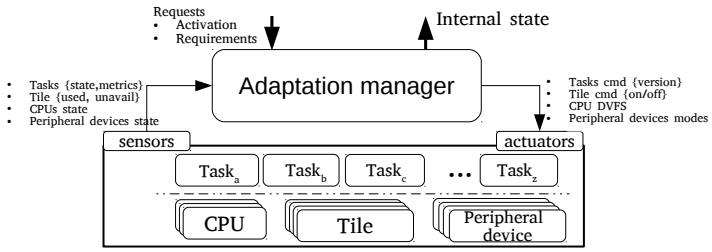


Fig. 7: Self-adaptation Manager

The manager must know the possible configurations of each architecture element and their difference according to relevant aspects for management strategy. It also must know for each task, the available versions and their characteristics in terms of resource, performance and quality of service.

IV. Domain specific language for the design of adapatation manager

This work is based on [9] which details the approach for designing an adaptation manager based on discrete control. We adopt automata-based modeling to capture the dynamics and the controllability of the resources of the system (CPUs, tiles, peripherals) and the processings (the tasks and the applications). The composition of the behavioral models describes the configurations space of the system including the subset which violates the management policies. *Discrete Controller Synthesis* is then applied to refine the composition with a control logic. The latter restrains the composition to the subset of configurations which satisfies the management policies.

The contribution of this paper is a high level, domain specific language. This enables designers to describe their FPGA architecture as well as the tasks and applications, and to declare the management policies. The compiler of the language is able to translate the description into automata and the policies as control objectives. Hence, designers who are not familiar with automata and formal languages can easily adopt the approach detailled in [9]. The automata are automatically contructed by the compiler of the DSL from the description.

A. System description

1) FPGA architecture: We consider that the relevant aspects about an FPGA necessary for the control problem are the set of tiles and the set of CPU that compose it and the set of connected peripheral devices.

a) Resources (Tiles/CPUs/Peripherals): The DSL allows describing the states of a resource in relation with its utilization, availability and configurations. The utilization and the availability are relevant information necessary for the allocation decisions. Indeed resources like tiles must be exclusively used. So, it is important to know when a tile is being used. The configuration of resources like CPU can be updated. For example, a CPU can be equipped with Dynamic Voltage Frequency Scaling (DVFS) configurations which enable changing the voltage and frequency to improve performance or reduce power consumption. A peripheral device can also have several possible configurations. The FPGA architecture is described by a set of resources.

2) Processings:

a) Task: A task is described in terms of processing states. A task can be active or not. It could also be waiting (this will be optional). This can happen when the task is requested but it has no software version or when we limit the number of software concurrently running and there is no available hardware. Multiple hardware and/or software versions of a task can be available. Each version is described in terms of required computing resources, performance and processing quality.

b) Application: An application is described as a set of tasks. The DSL enables specifying the execution mode of the tasks (sequential (DAG), parallel, data-flow or a mix of them).

B. Management strategies

The management strategies are mostly related to the performance, the quality of service and energy optimization. The objectives consist in:

- Maintaining the performance in defined intervals, e.g., execution time of a task greater than a minimum threshold and/or lower than a maximum threshold.
- Ensuring coherent usage of the resources, e.g., mutual exclusion in relation to the tiles.
- Ensuring coherent configuration of the resources to reduce energy consumption while maximizing the performance.

C. Description of the Syntax

Each managed element can have these following statements: `in`, `out` and `policy`. The `in` (resp. `out`) statement contains the declaration of the input (resp. output) variables separated by a **comma**, and ends with **semi-colon**. The input variables correspond to the events (e.g., failure, metrics) upon which the decisions are based. The output variables contain the value that are the criteria for choosing the next configuration. The `policy` statement contains the declaration of the management strategies to satisfy. A variable is declared as follows : `name [int | bool]` (possibly followed by = *`default_value`*). It has an identifier (`name`), a type. Currently, the supported types are : `int` and `bool`. An output variable can have a default value.

1) Resource (Tile, CPU, Peripheral): A resource can be exclusive or shareable. The keyword `provides` enables declaring shareable elements that the resource provides. The syntax `provides N of U` means that the resource provides `N` elements of unit `U`.

Figure 8 shows a description of a tile. It can be used as processing element (`Pe`) or for storage (`Mem`). When used as processing element, it provides 100 units of CPU (which can be read, e.g., as a percent of use of the CPU by unit of time) When used as `Mem`, it provides 1024 MBytes of RAM.

A resource can have different possible configurations. An example is the DVFS available in modern CPU or the quality of produced images of a camera. The configurations are

```
resource tile:
    configurations
        config PE:
            provides 100 of CPU
        end;
        config Memory:
            provides 1024 of mbytes
        end
    end
end
```

Fig. 8: Example of tile description

```
resource arm:
    out: speed int, energy int;
    provides 5 of arm_unit;
    configurations DVFS:
        config Low : speed = 10; energy = 5 end;
        config Medium : speed = 50; energy = 10 end;
        config High : speed = 100; energy = 25 end
    end
end
```

Fig. 9: Example of CPU description

declared in the section called **configurations**. Each item of a configuration has characteristics, as shown in Figure 9 and 10.

Figure 9 describes an ARM CPU. The number of active tasks simultaneously executed by the CPU is limited to 5. It is equipped with DVFS enabling three possible configurations: `Low`, `Medium` and `High` which differ in terms of processing speed and energy consumption.

```
resource camera:
    out: pixel int, delay int, factor int;
    provides 5 of camera_unit;
    configurations Size:
        config Small: pixel = 256; delay = 3 end;
        config Normal: pixel = 1024; delay = 12 end;
        config Large: pixel = 4096; delay = 48 end;
    end;
    configurations Quality:
        config Low : factor = 1 end;
        config Fine : factor = 3 end;
        config High : factor = 7 end;
    end
end
```

Fig. 10: Example of Peripheral description

Figure 10 describes a camera. The size of the images produced by the camera can be configured as well as the quality of the images. There are three possible configurations for the size : `Small`, `Normal` and `Large`. They differ in terms of pixels (`pixel`) and the processing time (`delay`). The quality of the images can be configured in `Low`, `Fine` or `High`.

2) Processings:

a) Task: The description of a task can constain four sections: **in, out**, **version** and **policy**. A **version** section contains the description of one implementation of the task. It describes the required resources, the estimated performance, quality of service and other criteria necessary for the decision.

Figure 11 shows an example of the description of a task. The task has two versions: v_1 and v_2. The version v_1 uses

```
task task_a :
    in : e_a bool, e_b bool;
    out : wcet int;
    version v_1: uses 1 of arm, wcet = 500 end
    version v_2: uses 20 of tile.Pe, wcet = 300 end
    policy:
        e_a then v_1;
        e_b then v_2;
    end
end
```

Fig. 11: Example of task description

an ARM and takes 500 sec in the worst case execution time (wcet). The version v_2 uses a tile as processing element. It takes 300 sec in the worst case execution time. The input e_a (resp. e_b) is the event that triggers switching to v_1 (resp. v_2).

b) Application: The description of an application can constain four sections : **in, out**, **execution** and **policy**. The **execution** section contains the description of the execution of the tasks.

```
application app:
    execution: task_a ; task_b end
end
```

Fig. 12: Example of application description

The *execution flow* of the tasks can be sequential. This is represented with **";"** as shown in Figure 12. Parallel execution is represented with "||" (e.g., ($task_a$ || $task_b$)). In our definition, parallel execution does not necessarily mean that the involved tasks are simultaneously activated. It just means that there is no *precedence relationship* in term of execution flow. Hence, to represent *data-flow* execution, we use the same syntax as for the parallel execution ("||"), and we declare a statement in the **policy** section to specify that the involved tasks must be activated (stopped) at the same time. This is declared with "▷" as shown in this example : ($task_a$.v_1 ▷ $task_b$.v_2). This says that the version v_1 of $task_a$ matches with the version v_2 of $task_b$ and whenever $task_a$.v_1 is running $task_b$.v_2 must be running.

3) Policies: The management strategies can be declared in the form of :

- **Condition then action**: As shown in Figure 11 (e_a **then** v_1), the designer can explicitly "program" the decision. In this case the synthesized control logic will only apply the decision if it is possible. However, this approach can be too constraining.
- **Invariance**: The designer can also specify invariants (e.g., `min` $\leq$ `time_t`). In this case, the control logic is responsible of choosing which is the next configuration such that the invariants are satisfied. Defining invariants can be more efficient. It gives more flexibility to the control logic. The synthesized control logic is maximally permissive.

The following Keywords available for expressing the management strategies are: **not**, **or**, **and**, **then**, **prior**, **pre**, ▷.

D. Compilation to a reactive language

The DSL is compiled towards Heptagon/BZR, used as a back-end. The declarative/imperative feature of this target language allows the structural translation of DSL elements (resources, tasks and applications) towards Heptagon/BZR equations, automata and synthesis objectives.

Each resource is translated into equations and automata describing the current configuration of the resource. Each task is translated into equations and automata describing current used versions, and synthesis objectives associating these versions with the appropriate resource configuration. Each application is translated into automata, emitting requests for task launches.

The inputs `c*` in Figures 13,14,15 are automatically added in the automata. They are the controllable variables through which the control decisions are applied.

1) Resources translation: A resource declaration is composed of equations, defining outputs, whose value can depend on the current configuration of the resource. Providing of shared units are also translated to equations, defining the number of units provided:

DSL	Heptagon/BZR
provides (x) of unit;	provides_unit = x;

A set of configurations is translated to one automaton, in which each configuration corresponds to one state (see Figure 13).

DSL	Heptagon/BZR

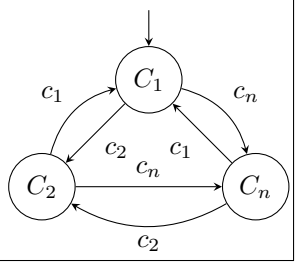
`resource ...`
`  ...`
`  configurations C:`
`  config C1: ...`
`  config C2: ...`
`  ...`
`  config Cn: ...`
`  end`
`  ...`
`end`
	

Fig. 13: Translation of configurations to Heptagon/BZR automaton

2) Tasks translation: A task is translated into an automaton composed of at least two states: `Inactive` and `Active`. The `Inactive` state is the initial state. The automaton can also have an additional state named `Wait`. This represents that the task is suspended or its activation is delayed.

As shown in Figure 14, the input `c` represents the control of the activation of the task. The input `r` represents the request to start the task. When it is **`true`**, the task becomes `Active` if `c` is **`true`**. Otherwise it goes to the `Wait` state waiting for `c` to become **`true`**. The input `e` represents the request to stop the task. In `Wait` or `Active`, when `e` is **`true`** the task becomes `Inactive`. Depending on the description of the user, the inputs `r` and/or `e` can be notifications instead of requests. This is the case when the task is not controlled but just observed. The output `act` indicates the current state of the task.

The `Active` state represents the running state. It encapsulates an sub-automaton in which each state corresponds to a declared `version`. The initial state is the first declared `version`. The output `ver` indicates the active version.

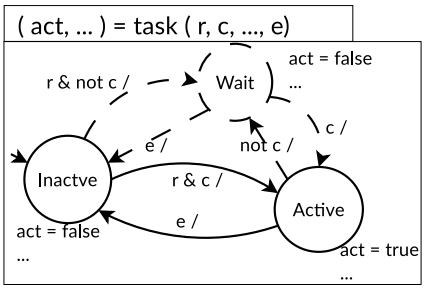


Fig. 14: Translation of a task

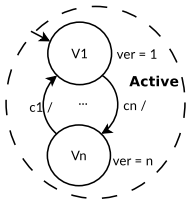


Fig. 15: Translation of a task

3) Applications translation: An application is translated in a composition of automata corresponding to the instances of the tasks which compose the application.

$(\texttt{act}_1,\ \texttt{ver}_1,\ \ldots) = \texttt{application}\ (\texttt{r}_1,\ \texttt{e}_1,\ \ldots)$

$(\texttt{act_1},\ \texttt{ver}_1,\ \ldots) = \texttt{task}_1\ (\texttt{r}_1,\ \texttt{e}_1,\ \ldots)\ ;$
$\ldots$
$(\ldots) = \texttt{task}_n\ (\ \ldots\ )$

Fig. 16: Application translation

Figure 16 shows an illustration of a translation of an application. The application is composed of `n` tasks. The automata are composed in parallel.

4) Policies translation: The policy section is translated into BZR contract.

```
assume  true
enforce  equations
with  c, ...
```

Fig. 17: policy

Figure 17 shows an illustration. The equations within the policy are formalized in the **`enforce`** section of the contract. The control variables of automata are declared in the **`with`** section.

5) Main program translation: The partial translations above are composed into a global program in H/BZR, where domain-specific features have been encoded and formalized as equations, automata and contracts. This H/BZR program is compiled, involving DCS and resolution of the constraints and contracts, into executable C code. This executable implements

the Autonomic Manager, which is ready for integration into the execution runtime of the system, by connection with the monitored variables and the reconfiguration actions.

V. Case study : Search-landing-area

This section shows an example of the application of Ctrl-DPR, first on the very simple case of a single task, which enables us to illustrate our approach from high-level specification, through automata compilation, executable C code generation, all the way down to FPGA implementation.

A. Informal description

We consider an FPGA-based embedded system with one task: search-landing-area. The task receives flow of images from the camera and performs a sequence of transformation on each image in order to determine suitable areas for landing. The task can process and image with a SW or a HW versions. The architecture model is based on ARM CPUs running SW tasks and an FPGA area divided in a static part and a single reconfigurable tile. Depending on the urgency of landing, the landing task can be executed with the HW or the SW version. The interest of going for a much slower software version of the task is that the FPGA resource can then be released, and used by another task high priority task such as obstacle avoidance.

The execution requirements associated with the search-landing-area task is defined in terms of interval delimited by a minimum threshold and a maximum threshold. The thresholds can be adapted at runtime depending on the urgency of landing. The management strategy within the control loop consists in maintaining the processing time of an image between the minimum threshold and the maximum threshold. We can see that this strategy cannot be achieved all time.

The control strategy consists of keeping the execution time of the task between a minimum threshold and a maximum threshold, which define the range of acceptable performance. Below the minimum threshold, a version which uses lower resources can be executed if its performance is inside the interval in order to minimize the active hardware resources. Reducing the hardware resources used can make sense in systems where many tasks share them, so that hey are allocated to the tasks needing them most urgently.

B. Specification in Ctrl-DPR

With Ctrl-DPR, we describe the system and the control problem as shown in Figure 18. The tile can be used only as a processing element and can become unavailable. The CPU as well as the camera are shareable. The description of the task has three inputs corresponding to the value of the thresholds and the measured execution time (`time_t`). The ouput `wcet` indicates the estimated execution time of each version of the task. In the policy, `time_t` $\cong$ `wcet` indicates that there is a relation between `time_t` and `wcet`. Hence, acting on `wcet` affects `time_t`. The `main` decribes the global composition.

C. Automata-based Behavioral models and contract

Figure 19 shows Heptagon/BZR code [18] generated for the description of the tile and Figure 20 the model of the task.

```
resource tile :
  ... ;
end

resource arm :
  ... ;
end

resource camera :
  ... ;
end

task searchArea :
  in : min int, max int, time_t int;
  out : wcet int;
  version Software : uses arm, wcet = 1500 end
  version Hardware : uses tile, wcet = 55 end
  policy :
      (min < time_t) and (time_t < max) and
      (time_t ~ wcet);
end
```

Fig. 18: Description of the control problem

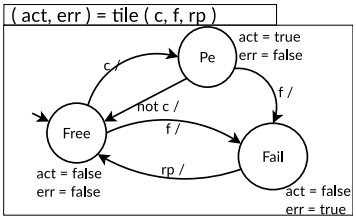


Fig. 19: Tile automaton

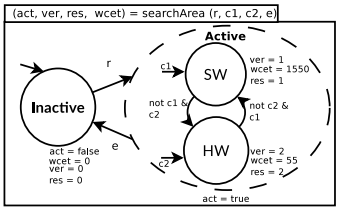


Fig. 20: Task automaton

Figure 21 shows the global composition of the Heptagon/BZR models (task and FPGA resources). The contract shows objectives (`objective`) where the three first lines correspond to the policy in Fig. 18. When the observed execution time (`time_t`) is greater than the `max`, it switches to a version with lower `wcet`. When `time_t` is lower than `min`, it switches to a version with greater `wcet` but lower than `max`. The other lines consist in preventing the `hw` version of the task from being selected when the tile is unavailable, and also preventing the tile from being active while not used.

```
main (r, e, time_t, min, max, f, rp)
= act_t, res, wcet, act, err, ..., objective
assume true
enforce objective
with cp1, cp2, cp3, c1, c2, c, ...

(act_t, ver, res, wcet) = searchArea (r, c1, c2, e) ;

(act, err) = tile (c, f, rp) ;
(...) = arm ( ... ) ;
(...) = camera ( ... )

objective = (
  (cp1 ⇒ (((0 fby wcet) > wcet) ⇒ (wcet < max))) and
  (cp2 ⇒ ((time_t < min) ⇒ ((0 fby wcet) < wcet))) and
  (cp3 ⇒ ((time_t > max) ⇒ ((0 fby wcet) > wcet))) and
  (err ⇒ not (res = 2)) and (not (res = 2) ⇒ not act)
 ) ;
```

Fig. 21: Global Heptagon/BZR program

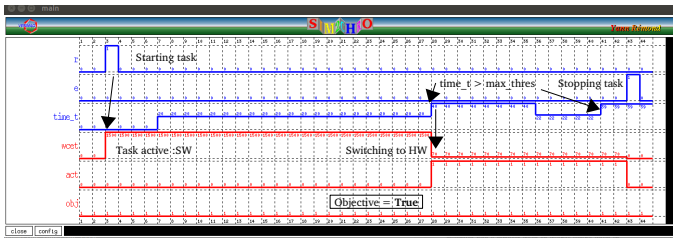


Fig. 22: Simulation

D. Controller simulation before integration

Figure 22 shows a snapshot of the simulation of the generated manager. The minimum threshold is fixed at 8 and the maximum threshold at 29. We observe, at step 3, when the stask is requested, the manager triggers the activation of the SW version of the task. At step 28, the execution time of the task becomes greater than the maximum threshold (`time_t > max_thres`). The manager switches to the HW version which is the fastest version and requires the tile. We see that the manager reacts correctly with respect to the defined objectives.

E. A variant of the case study with two tasks

We describe the control of a system including two tasks. The FPGA board has one tile and one CPU ARM. Each task has two versions (`sw` and `hw`). The `hw` versions of the tasks can not be active simultaneously because there is only one tile in the system. In this example, we show that the generated manager enforces exclusive use of the tile, and how conflict are resolved based on priority specified with the keyword `prior`.

```
...
main :
  in : prio_t1 int, prio_t2 int;
  Involve :
      1 tile, 1 arm, 1 camera, 1 task1, 1 task2;
  Policy :
      (prio_t2 ≤ prio_t1) then (task1 prior task2);
      (prio_t2 > prio_t1) then (task2 prior task1);
end
```

Fig. 23: Description of the two tasks control problem

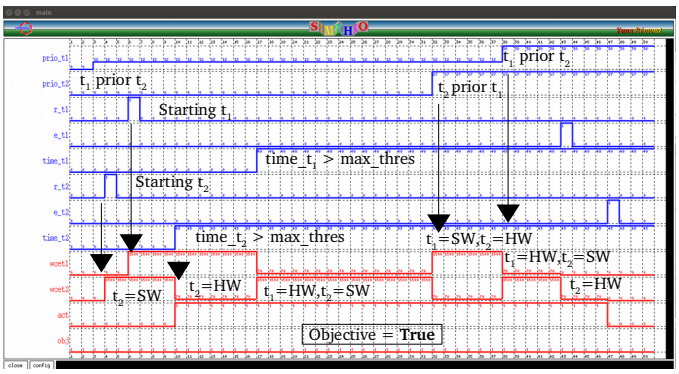


Fig. 24: Simulation

Figure 24 shows a snapshot of the simulation of the generated manager. The minimum threshold is fixed at 5 and the maximum threshold at 40. At step 6, the `sw` versions of the tasks are running. $task_1$ is prior, but when the execution time of $task_2$ is greater than the maximum threshold, the tile is allocated to $task_2$ (step 10). When the execution time of $task_1$ becomes greater than the maximum threshold, the tile is retreived from the $task_2$ and allocated to $task_1$ because it is prior. At step 32, the priority are changed and the tile is allocated to $task_2$ which was the priority task.

Ctrl-DPR allows describing the control of a system with multiple tasks. Conflicting decisions are resolved based on priority, which can be changed dynamically by the environment.

VI. FPGA IMPLEMENTATION OF THE CASE STUDY

To validate our approach, we execute the generated adaptation manager for the search-landing-area task. Although the example is very simple, it illustrates the approach down to hardware implementation. We use the DE1-SoC board, and the Robot Operating System (ROS) [20] for the communication between the adaptation manager and the controlled task.

a) DE1-soc system: The board is based on a Altera/Intel Cyclone® V SoC chip which supports DPR. It includes a Hard Processor System (HPS) and an FPGA. The HPS comprises an ARM Cortex A9 dual-core processor, a DDR3 memory port, and a set of peripheral devices. The FPGA implements the reconfigurable tile (one in this experiment) and different peripheral controllers. We run a Linux OS on the HPS side. We implement a cma_driver module for the interaction with the hardware implementation of the task. The module allocates a continuous area of memory in the kernel space.

b) Search-landing-area application: The application is composed of 6 functions : 1) pixel format converter, 2) median filter, 3) Canny filter, 4) Dilatation/Erosion, 5) area coordinate extraction and 6) landing pattern matching to identify valid candidates. We consider some benchmark pictures which are stored and an external SD card and so not a camera for the experiment. In the first version all the functions are implemented in SW and the WCET is 1500 ms including overheads due to the SD card access and the use of a middleware, which is necessary for a flexible configuration of application, platform and sensors (ROS over Linux, described in SectionVI-0c).

The HW version implements functions 1-4 on the FPGA and a DMA port to execute functions 5 and 6 on the CPU. It allows to reach 60fps with a very limited clock frequency (25Mhz). On the target Cyclone-V, it uses 2.4%, 4% and 3% of ALM, BRAM and DSP blocks respectively. It means it can be easily implemented in a single tile in case of a 4 tile-architecture model. The complete worst case execution time is 55ms including the previously cited overheads.

c) ROS: support for the communication: We use the Robot Operating System (ROS) [20] to enable the manager to communicate with the tasks. ROS enables to run in parallel a large number of executables that must be able to exchange information synchronously or asynchronously. The fundamental concepts of the ROS implementation are `nodes`, `messages`, `topics`, and `services`. In this evaluation we have two ROS nodes. We define a Task node for the search-landing-area task.

We also implement a ROS node for the adaptation manager. ROS supports the two modes to execute the **step** from the compilation in Section IV-D : even-driven (e.g., **ros::spin** ()) or periodic (i.e., **ros::spinOnce** ()). In this evaluation we choose the periodic method.

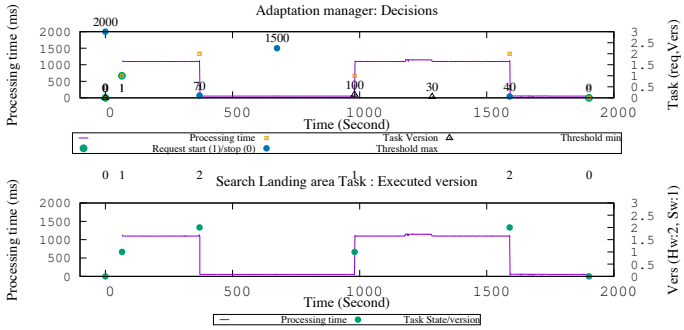


Fig. 25: Manager decisions

d) Experimentations: We first send to the manager a request to start the search-landing-area task. When the task is active, we change the value of the minimum and maximum thresholds to see which decisions the manager takes. Figure 25 shows an execution in which we set the value of the thresholds as follows. Initially the maximum threshold is set to 2000 ms while the minimum threshold is set to 0 ms. After *370 sec* of execution the maximum threshold is set to 70 ms. Later, it is set to 1500 ms and finally to 40 ms after *1590 sec*. The minimum threshold is set to 100 ms after *980 sec* and later to 30 ms. As shown in Figure 25, the manager dynamically adapts the version executed depending on minimum and maximum thresholds.

VII. Conclusion

The contribution of this paper is a high level, domain-specific language for the control of FPGA-based systems. It allows designers to describe their FPGA architecture as well as the tasks and application, and to declare the management policies. The compiler of the language is able to translate the system description into automata and the policies into control objectives; the latter are then compiled by H/BZR into an executable, correct Autonomic Manager enforcing the policy.

Perspectives are in several directions, amongst which coordination of multiple autonomic loops, switching controllers (integrating into the DSL schemes explored in another context [21]) ; modularity and hierarchical loops, both for re-use and to managed compilation complexity, and considering more advanced discrete control features like logico-numeric properties [22], [23] ; and integrating the DSL for Autonomic Managers into a global design process for FPGA-based applications.

References

[1] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, pp. 41–50, January 2003.

[2] M. D. Santambrogio, "From reconfigurable architectures to self-adaptive autonomic systems," in *Int. Conf. Computational Science and Engineering, 2009. CSE '09.*, vol. 2, Aug 2009, pp. 926–931.

[3] M. Litoiu, M. Shaw, G. Tamura, N. M. Villegas, H. Müller, H. Giese, R. Rouvoy, and E. Rutten, "What Can Control Theory Teach Us About Assurances in Self-Adaptive Software Systems?" in *Software Engineering for Self-Adaptive Systems III. Assurances.*, R. de Lemos, D. Garlan, C. Ghezzi, and H. Giese, Eds., 2018, vol. LNCS 9640.

[4] A. Filieri, H. Hoffmann, and M. Maggio, "Automated design of self-adaptive software with control-theoretical formal guarantees," in *Proc. 36th Int. Conf. Software Engineering*, ser. ICSE 2014, 2014.

[5] N. D'Ippolito, V. Braberman, J. Kramer, J. Magee, D. Sykes, and S. Uchitel, "Hope for the best, prepare for the worst: Multi-tier control for adaptive systems," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, 2014.

[6] E. Rutten, N. Marchand, and D. Simon, "Feedback Control as MAPE-K loop in Autonomic Computing," in *Software Engineering for Self-Adaptive Systems III. Assurances.*, ser. LNCS, R. de Lemos, D. Garlan, C. Ghezzi, and H. Giese, Eds. Springer, Jan. 2018, vol. 9640.

[7] X. An, E. Rutten, J.-P. Diguet, N. le Griguer, and A. Gamatié, "Autonomic management of dynamically partially reconfigurable fpga architectures using discrete control," in *Proc. 10th Int. Conf. Autonomic Computing (ICAC 13)*, San Jose, CA, 2013.

[8] X. An, E. Rutten, J.-P. Diguet, and A. Gamatié, "Model-based design of correct controllers for dynamically reconfigurable architectures," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 15, no. 3, Jun. 2016. [Online]. Available: http://dx.doi.org/10.1145/2873056

[9] S. M. K. Gueye, É. Rutten, and J. Diguet, "Autonomic management of missions and reconfigurations in fpga-based embedded system," in *NASA/ESA Conf. Adaptive Hardware and Systems, AHS, Pasadena, CA, USA, July 24-27, 2017*, 2017, pp. 48–55. [Online]. Available: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=8030149

[10] F. Alvares, E. Rutten, and L. Seinturier, "Behavioural model-based control for autonomic software components," in *Proc. IEEE Int. Conf. Autonomic Computing, ICAC*, 2015, pp. 187–196.

[11] F. Fons, M. Fons, E. Cantó, and M. 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.*, vol. 8, no. 3, pp. 229–251, Sep. 2013.

[12] E. Chen, V. G. Lesau, D. Sabaz, L. Shannon, and W. A. Gruver, "Fpga framework for agent systems using dynamic partial reconfiguration," in *Proc. 5th Int. Conf. Industrial Applications of Holonic and Multi-agent Systems for Manufacturing*, ser. HoloMAS'11, 2011, pp. 94–102.

[13] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[14] J. Hellerstein, Y. Diao, S. Parekh, and D. Tilbury, *Feedback Control of Computing Systems*. Wiley-IEEE, 2004.

[15] Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, and S. Mahlke, "The theory of deadlock avoidance via discrete control," in *Proc. Conf. POPL*, 2009.

[16] D. Harel and A. Pnueli, "Logics and models of concurrent systems," K. R. Apt, Ed. New York, NY, USA: Springer-Verlag New York, Inc., 1985, ch. On the Development of Reactive Systems, pp. 477–498.

[17] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, Jun. 1987.

[18] G. Delaval, É. Rutten, and H. Marchand, "Integrating discrete controller synthesis into a reactive programming language compiler," *Discrete Event Dynamic Systems*, vol. 23, no. 4, pp. 385–418, 2013.

[19] N. Berthier and H. Marchand, "Discrete controller synthesis for infinite state systems with reax," in *IEEE International Workshop on Discrete Event Systems*, Cachan, France, May 2014, pp. 46–53.

[20] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[21] A. N. Sylla, M. Louvel, E. Rutten, and G. Delaval, "Design Framework for Reliable Multiple Autonomic Loops in Smart Environments," in *2017 IEEE International Conference on Cloud and Autonomic Computing (ICCAC)*, Tucson, AZ, United States, Sep. 2017.

[22] F. Alvares, G. Delaval, E. Rutten, and L. Seinturier, "Language support for modular autonomic managers in reconfigurable software components," in *2nd Workshop on Self-Aware Computing, SeAC@ICAC*, 2017.

[23] N. Berthier, F. Alvares, G. Delaval, H. Marchand, and E. Rutten, "Logico-numerical control for software components," in *1st IEEE Conference on Control Technology and Applications, CCTA*, 2017.