Preprints of 4th IFAC Nonlinear Model Predictive Control Conference
International Federation of Automatic Control
Noordwijkerhout, NL. August 23-27, 2012

MOBY-DIC: A MATLAB Toolbox for Circuit-Oriented Design of Explicit MPC

Alberto Oliveri * **Davide Barcelli** ** **Alberto Bemporad** **
Bart Genuit *** **Maurice Heemels** *** **Tomaso Poggi** ****
Matteo Rubagotti † **Marco Storace** *

* *Department of Naval, Electric, Electronic and Telecommunications Engineering, University of Genoa, Via Opera Pia 11A, Genova, Italy; e-mail: {alberto.oliveri,marco.storace}@unige.it*
** *IMT Institute for Advanced Studies, Piazza S. Ponziano 6, Lucca, Italy; e-mail: {davide.barcelli,alberto.bemporad}@imtlucca.it*
*** *Hybrid and Networked Systems Group, Department of Mechanical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands; e-mail: {m.heemels,b.a.g.genuit}@tue.nl*
**** *RF group, ESS-Bilbao, Paseo Landabarri, 2, Leioa, Spain; e-mail: tpoggi@essbilbao.org*
† *Nazarbayev University, Kabanbay Batyr Ave 53, Astana, Kazakhstan; e-mail: matteo.rubagotti@nu.edu.kz*

Abstract: This paper describes a MATLAB Toolbox for the integrated design of Model Predictive Control (MPC) state-feedback control laws and the digital circuits implementing them. Explicit MPC laws can be designed using optimal and sub-optimal formulations, directly taking into account the specifications of the digital circuit implementing the control law (such as latency and size), together with the usual control specifications (stability, performance, constraint satisfaction). Tools for a-posteriori stability analysis of the closed-loop system, and for the simulation of the circuit in Simulink, are also included in the toolbox.

1. INTRODUCTION

Despite the importance of embedded control in many applications, there are only few integrated methods to design and deploy embedded control systems in a systematic and highly efficient manner. There is an abundance of methods for the separate design of the control algorithms eventually embedded into a hardware platform, or to create a suitable hardware platform for the implementation of a given control algorithm. These methods typically follow a sequential design procedure as depicted in Fig. 1(a). On the other hand, to the best of our knowledge, no tools are available for automated and integrated design flows from the mathematical model of the process to the synthesis of the electronic circuits. Ideally, such tools should incorporate both control specifications (stability, constraint satisfaction, performance) and circuit requirements (speed, size, power usage, cost), as depicted in Fig. 1(b).

This paper is concerned with a software toolbox (called MOBY-DIC) implementing an integrated design flow that is based on explicit MPC [Bemporad et al., 2002]. The resulting control law is a piecewise-affine (PWA) function of the system state, which allows one to cope with constraints on input and state variables without the need to solve an optimization problem on-line, and is therefore very effective to control small-sized systems for which small sampling times are needed [Alessio and Bemporad, 2009]. MATLAB toolboxes are available that allow to design explicit MPC control laws, see, e.g., Bemporad [2004], Kvasnica et al. [2004]. Interestingly, the hardware implementation of PWA functions has been deeply studied in circuit theory (see, e.g., Parodi et al. [2005], Storace and Poggi [2011] and the references therein), and the related results can be exploited to setup an integrated design methodology for MPC-based embedded controllers.

The MOBY-DIC Toolbox for MATLAB provides an automatic tool chain for the circuit design of embedded control

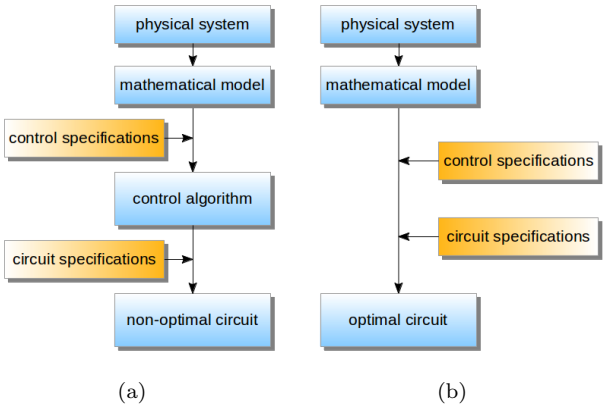

Fig. 1. Comparison between current (a) and MOBY-DIC-based (b) design vision for embedded control systems.

⋆ This work was partially supported by the European Commission under project FP7-INFSO-ICT-248858 "MOBY-DIC – Model-based synthesis of digital electronic circuits for embedded control". http://www.mobydic-project.eu/

systems, given a linear time-invariant model of the physical process, the constraints on states and input variables, and the specifications on the digital circuit actually employed to implement the control law, according to the design flow shown in Fig. 1(b). This means that the toolbox allows the designer to have information in advance on the resulting circuit implementation of the controller, in order to ease trading off between control performance, restrictions in area occupancy and speed of implementation. Circuit implementations (with automatic generation of VHDL code) are provided for both exact (optimal) control laws (obtained through the direct interaction of the MOBY-DIC Toolbox with the Hybrid Toolbox [Bemporad, 2004] or the Multiparametric Toolbox [Kvasnica et al., 2004]), and for sub-optimal ones based on recent results by Bemporad et al. [2011], Genuit et al. [2012], Lu et al. [2011]. The sub-optimal methods permit to obtain faster circuit implementations, since they exploit PWA functions defined over regular partitions of the domain, at the cost of loss of both optimality and, in some cases, a-priori closed-loop stability. Methods for the a-posteriori stability analysis of the closed-loop system, and for the evaluation of the domain of attraction (mainly based on Rubagotti et al. [2012]), are also provided with the toolbox. Moreover, the toolbox offers several methods for the simulation of the closed-loop system, which allows one to test the designed circuit and, if necessary, to intervene both on the circuit settings and on the control parameters. The toolbox can also interact with Xilinx System Generator, to obtain a Simulink block representing the controller. In this way, one can simulate the effect of the circuit latency and the quantization error introduced by the fixed point representation.

Finally, the toolbox allows also the automated circuit implementation of direct virtual sensors (Poggi et al. [2012]), but this functionality will not be described in the present paper for space limitation reasons.

The toolbox can be freely downloaded. [1] A complete HTML documentation describing all objects, methods, functions as well as the circuit architectures is also available.

2. MODEL PREDICTIVE CONTROL

Consider a linear discrete-time system

$$x(t+1) = Ax(t) + Bu(t) \tag{1}$$

where $x \in \mathbb{R}^{n_x}$ and $u \in \mathbb{R}^{n_u}$ denote the system states and inputs, respectively. The control input $u(t)$ can be computed using MPC, by solving the following quadratic program

$$\min_{U} \quad x_N' P x_N + \left(\sum_{k=0}^{N-1} x_k' Q x_k + u_k' R u_k \right) + \rho \epsilon^2 \tag{2a}$$

$$s.t. \quad \begin{array}{ll} x_0 = x(t), & \\ x_{k+1} = Ax_k + Bu_k, \ k = 0, \dots, N-1, & (2b) \\ u_k = Kx_k, \ k = N_u, \dots, N-1 & (2c) \\ E_u u_k \leq G_u, \ k = 0, \dots, N_u - 1 & (2d) \\ E_u u_k \leq G_u + V_u \epsilon, \ k = N_u, \dots, N-1 & (2e) \\ E_x u_k + F_x x_k \leq G_x + V_x \epsilon, \ k = 0, \dots, N-1 & (2f) \\ \epsilon \geq 0 & (2g) \end{array}$$

where N and N_u are the prediction and control horizons, respectively, $U = \left[u_0' \ \dots \ u_{N_u-1}' \ \epsilon \right]' \in \mathbb{R}^{n_u N_u + 1}$ is the vector of optimization variables, ϵ is a slack variable relaxing the constraints, and $\rho > 0$ is a (large) weight penalizing constraint violations. E_u, G_u and V_u are matrices of appropriate dimensions defining input constraints, while E_x, F_x, G_x and V_x represent mixed input-state constraints. The above problem can be solved off-line as a multi-parametric quadratic program (mpQP) which leads to obtaining u_0 (the other elements of U are discarded, according to the *receding horizon* principle) as an explicit function of the current state (parameter) x [Alessio and Bemporad, 2009]. More specifically, the control law $u_0(x)$ is a PWA function of the current state x_0 defined over a generic polytopic partition. Other cost functions based on 1-norm and ∞-norm can also be employed instead of (2), leading to a multi-parametric Linear Program (mpLP). Since also in this case the resulting control law $u_0(x)$ is a PWA function of the state, the approximation and circuit tools presented here can be applied analogously. [2]

3. PWA FUNCTIONS

Given a hyper-rectangular compact domain $\mathcal{D}$ partitioned into n_r convex polytopes $\mathcal{P}_i$, $i = 1, \dots, n_r$ a PWA function $f_{\text{pwa}} : \mathbb{R}^{n_d} \to \mathbb{R}^{n_f}$ can be expressed as:

$$f_{\text{pwa}}(x) = F_i x + G_i, \quad \text{if } x \in \mathcal{P}_i \tag{3}$$

where $F_i \in \mathbb{R}^{n_f \times n_d}$ and $G_i \in \mathbb{R}^{n_f \times 1}$, $i = 1, \dots, n_r$ are the coefficients of the affine expressions over each region. A generic polytope $\mathcal{P}_i$ is represented by a set of inequalities $H_i x \leq K_i$, with $H_i \in \mathbb{R}^{n_{e,i} \times n_f}$ and $K_i \in \mathbb{R}^{n_{e,i} \times 1}$; $n_{e,i}$ denotes the number of edges of polytope $\mathcal{P}_i$. Therefore, a PWA function is completely defined by all coefficients F_i,G_i,H_i and K_i. PWA functions defined over any irregular polytopic domain partition are referred to as PWAG functions. Particular cases of domain partitions (see Fig. 2) are the simplicial partition, leading to PWAS functions, and the hyper-rectangular partition, leading to PWAR functions.

• **PWAG functions:** the evaluation of a PWAG function at a point x is performed in two steps: (i) locate the polytope $\mathcal{P}_i$ containing x (point location problem) and (ii) compute the affine expression $F_i x + G_i$. The second step is trivial; a higher computational effort is required for the point location problem. Some efficient algorithms, besides the inefficient direct search, have been proposed in the literature (see for example Tøndel et al. [2003], Bayat et al. [2011]). We implemented in the toolbox the algorithm by Tøndel et al. [2003], in which a binary search tree is constructed off-line, based on the domain partition. Each non-leaf node of the tree corresponds to a partition edge and each leaf node to a polytope. The tree is then explored on-line to locate the region containing a given point. The binary search tree can be easily mapped into a Finite State Machine, for a digital circuit implementation,

[1] `http://ncas.dibe.unige.it/software/MOBY-DIC_Toolbox`

[2] In general, the solution of a mpQP or mpLP can be a PWA function defined over overlapping polytopes; this happens for example in control applications of PWA or mixed logic dynamical systems. This situation cannot be managed by our approximation techniques. Nevertheless, for the applications we are interested in (control of linear systems), the obtained polytopes never overlap.

as described in Oliveri et al. [2009]. An example of PWAG function is shown in Fig. 5(a).

• **PWAS functions:** a simplicial PWA (PWAS) function is a PWA function defined over a domain partitioned into regular simplices. A simplex is a segment in one dimension, a triangle in two dimensions, a tetrahedron in three dimensions and so on. A simplicial partition is built by subdividing each domain component into p_j segments, $j = 1, \ldots, n_d$ with arbitrary length (if all segments have the same length we obtain uniform partitions); the domain is therefore partitioned into $\prod_{j=1}^{n_d} p_j$ hyper-rectangles which are further partitioned into $n_d!$ simplices each. Any continuous PWAS function can be represented as a weighted sum of PWAS basis functions in this way:

$$f_{\text{pwas}}(x) = \sum_{i=1}^{n_b} w_i \alpha_i(x) \tag{4}$$

where w_i are the weights and α_i are the PWAS basis functions defined as $\alpha_i(v_j) = \delta_{ij}$, being v_j, $j = 1, \ldots, n_b$ the vertices of the simplicial partition and δ_{ij} the Kronecker delta. Therefore, any PWAS function is univocally characterized by its domain, the simplicial partition and the set of weights w_i, $i = 1, \ldots, n_b$. In case of α basis functions, the weights w correspond to the values of the PWAS function in the vertices of the simplicial partition. The value of the function in any point x can be computed by linear interpolation of the value of the function in the vertices of the simplex containing the point. The location of the simplex (point location problem) can be carried on in a very simple way, due to the regularity of the partition [Parodi et al., 2005]. Digital architectures for the computation of PWAS functions have been proposed in Storace and Poggi [2011] and they are employed in the toolbox. An example of a PWAS function is shown in Fig. 5(b).

• **PWAR functions:** a hyper-rectangular PWA (PWAR) function is a PWA function defined over a domain partitioned into hyper-rectangles. We consider two kinds of PWAR functions: the single- (sPWAR) and the multi-resolution (mPWAR) type. The single-resolution type is characterized by the fact that the axes x_j, $j = 1, \ldots, n_d$ of domain $\mathcal{D}$ are divided into m_j intervals, possibly of different lengths, in which case the partition is non-uniform; the regions are stacked like the components of a matrix, as shown in Fig. 2(c). In contrast, the multi-resolution type of partition is built by selecting hyper-rectangular regions from multiple single-resolution partitions of $\mathcal{D}$; these r partitions have $m_j = 2^l, l = 1, \ldots, r$ divisions along each axis (for circuit implementation reasons), as shown in Fig. 2(d). Because of its hierarchical structure, this type of partition has a corresponding search tree (a generalized oct-tree), which is very beneficial for solving the point location problem. Digital architectures for the evaluation of PWAR functions have been proposed by Comaschi et al. [2012] and are implemented in the toolbox. An example of a PWAR function is shown in Fig. 5(c).

4. DESCRIPTION OF THE TOOLBOX

The MOBY-DIC Toolbox is a MATLAB/Simulink toolbox which makes extensive use of Object Oriented Programming. In the following, the main objects for the representation of a linear system, for the definition of constraints and for the description of PWA functions are described.

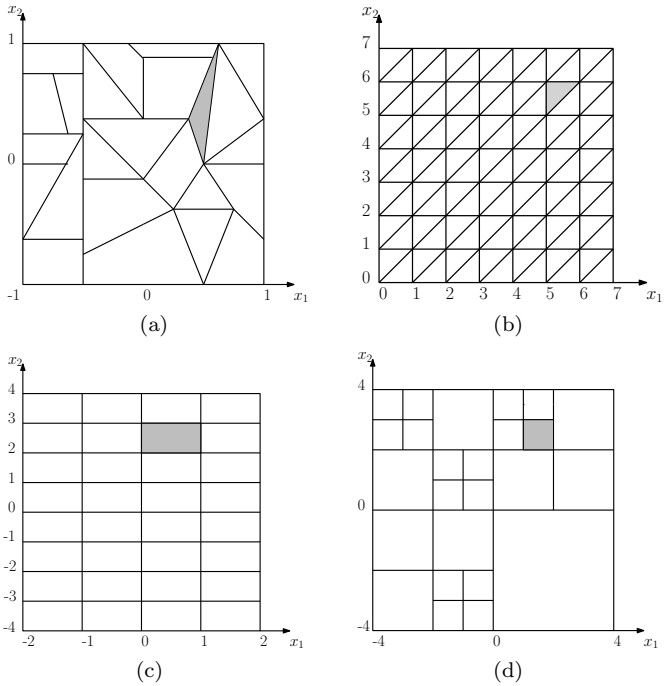

Fig. 2. Examples of two-dimensional domain partitions: generic (a), simplicial single-resolution (b), hyper-rectangular single- (c) and multi-resolution (d). The corresponding classes of PWA functions are: PWAG (a), PWAS (b), sPWAR (c), and mPWAR (d).

• **linearSystem:** this object represents a discrete time affine system in the form

$$x(t+1) = Ax(t) + Wp + Bu(t) + f$$
$$y(t) = x(t)$$

where $p \in \mathbb{R}^{n_p}$ is a vector of fixed parameters and $y_k \in \mathbb{R}^{n_x}$ represents the system outputs (always identical to the system states); A, W, B and f are matrices of appropriate dimensions. The object is created by specifying the number of state variables (n_x), parameters (n_p) and inputs (n_u); the value of the matrices and the sampling time are then set by means of methods *setMatrices* and *setSampling-Time*, respectively. Mnemonic names can also be assigned to states, parameters and inputs. Methods *simplot* and *sim* allow to simulate the linear system in closed-loop with any PWA controller, with or without time evolutions plotting.

• **constraints:** the *constraints* object allows to set constraints on system states, inputs and parameters for any time instant within a time horizon N_c. The general form of the constraints is:

$$H_c \begin{bmatrix} x_k \\ u_k \\ p \end{bmatrix} \leq K_c$$

with $k = 0, \ldots, N_c$ [3]. This formulation includes typical saturation constraints in the form $x_{MIN} \leq x_k \leq x_{MAX}$, $p_{MIN} \leq p \leq p_{MAX}$ or $u_{MIN} \leq u_k \leq u_{MAX}$. The object is created by specifying n_x, n_u, n_p as well as the horizon

[3] The constraints involving only x can be imposed from time $k+1$ to $k+N_c$; the constraints involving also u can be imposed from time k to $k+N_c-1$.

N_c. Each constraint is added to the object via method *setConstraints*, which allows a very simple and flexible usage.

• **pwaFunction:** the *pwaFunction* object is an abstract class (not instantiable directly), which represents any PWA function $f_{pwa} : \mathbb{R}^{n_d} \to \mathbb{R}^{n_f}$. Inherited objects are *pwag*, *pwas* and *pwar*. In this object the common properties to all derived classes are declared: the function domain (D) and its dimension (n_d), the number of dimensions of the codomain (n_f), and a structure (*synthesisInfo*) with details on the circuit implementation of the function. The most important common methods to all derived classes are *eval*, for the evaluation of the function in an array of points, *getRegions*, which returns all the regions partitioning the domain, *synthesize*, which generates the VHDL files describing a circuit implementing the function, and *generateSimulinkModel*, for the automatic generation of a Simulink model of the closed-loop system.

• **pwag:** the *pwag* object (derived from the *pwaFunction* class) represents a PWAG function. In this object, three structures are used to store all data, `edges`, `functions` and `regions`. Structure `edges` stores all edges of the polytopes without repetitions and without considering the sign of the edge.[4] The coefficients of the edges are stored in fields `H` and `K` of structure `edges`. Since the polytopes have, in general, some common edges, many repetitions in the coefficients H_i and K_i, also with opposite sign, can occur; the approach we used allows, therefore, to save memory occupation sensibly. Structure `functions` stores, in fields `F` and `G`, the coefficients F_i and G_i of all affine functions. As for previous structure, coefficients related to the same affine function are saved only once. `regions` is actually an array of n_r structures, one for each polytope partitioning the domain. These structures have two fields: `Iedges` and `Ifunctions`. The `Iedges` field of the i-th element of `regions` is a matrix with $n_{e,i}$ rows and 2 columns. The first column contains the indices of the coefficients H_i and K_i in matrices `edges`.H and `edges.K`. The second column contains the sign of the edges. In the same way, the field `Ifunctions` contains the indices of F_i and G_i in matrices `functions.F` and `functions.G`. The *pwag* object contains also a structure `tree` which stores the binary search tree necessary for a circuit computation of the function (Oliveri et al. [2009]). The structure contains a field `nodes` which is an array of structures whose elements correspond to the nodes of the tree. Each node is characterized by a univocal name and by the names of his children nodes; it contains moreover information about the set of edges and polytopes lying at the two sides of the edge corresponding to the node itself.

• **pwas:** the *pwas* object (derived from the *pwaFunction* class) represents a PWAS function. Any PWAS function is univocally defined by the domain partition and by the set of weights w_i; in this toolbox we only refer to PWAS functions described through α basis functions since they can be handled in a computationally efficient way (with sparse matrices). The simplicial partition is represented, in the MATLAB object, with a cell array `P` with as many elements as the function domain dimensions. The i-th entry of `P` contains the projection of the vertices of the partition along the i-th domain component. The weights are stored in an ordered one-dimensional array `w`.[5]

[4] $\hat{H}x \leq \hat{K}$ and $-\hat{H}x \leq -\hat{K}$ denote the same edge, with opposite sign.

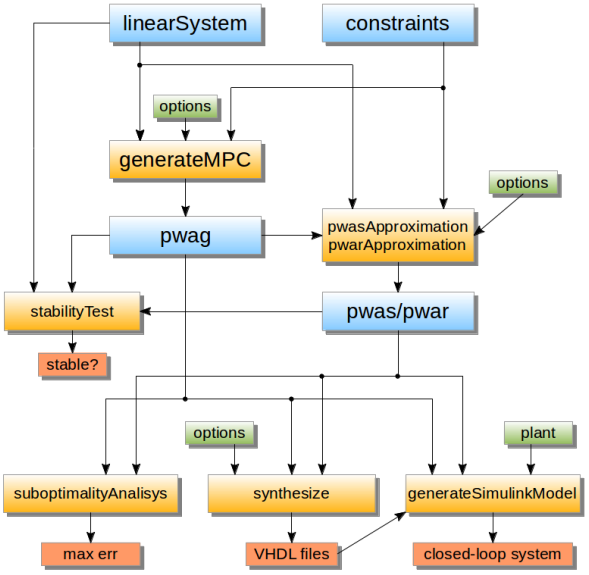

Fig. 3. Structure of the controller design flow.

• **pwar:** the *pwar* object (derived from the *pwaFunction* class) represents a PWAR function. This object has seven properties: `H`, `K`, `F`, `G`, `tree` and `details`. The main information is stored in cell arrays `H`, `K`, `F`, and `G`, with n_r elements, which contain the corresponding coefficients F_i, G_i, H_i and K_i, $i = 1, \ldots, n_r$. The matrix `tree` stores the information about the search tree necessary for circuit implementation of the function. This information is also used to speed up point location in some functions of the toolbox. The structure `details` is used to store various types of information, such as the original options used for the approximation, the type of partition (single- or multi-resolution), etc.

5. FUNCTIONALITIES

In this section the main functionalities of the toolbox are illustrated. Figure 3 shows how objects and functions are used to automatically generate a digital circuit starting from the definition of the system to control and of the constraints to impose. The functionalities are described in the following sections.

5.1 Design of MPC PWAG controller

The MOBY-DIC Toolbox provides a simple and flexible interface both to Hybrid Toolbox (Bemporad [2004]) and Multi Parametric Toolbox (Kvasnica et al. [2004]), for the design of explicit MPC controllers. Given a *linearSystem* object, a *constraints* object and some additional options, the *generateMPC* function returns a *pwag* object representing the explicit MPC control law, by resorting to one of the aforesaid toolboxes, which solve problem (2). The additional options, provided by means of a structure,

[5] If the codomain is multi-dimensional, w is a matrix with as many columns as the number of dimensions of the codomain.

are the prediction and control horizons, the cost function matrices and the type of norm for the MPC optimization problem, and some other settings specific to each toolbox. Default values are automatically set in case an option is not provided.

5.2 Design of approximate control laws

An approximate MPC controller can also be designed starting from the exact MPC control law. Two methods are available to accomplish to this task.

• **PWAS controller:** the *pwasApproximation* function allows to obtain a *pwas* object that approximates the *pwag* one by using the method described in Bemporad et al. [2011]. The same constraints satisfied by the exact MPC law are also enforced in the approximate one, though possibly softened with slack variables. The routine needs the *pwag*, *linearSystem* and *constraints* objects and the desired simplicial partition and returns a *pwas* object. Some options can be provided to this function, such as the norm to use in the optimization problem and the solver to be employed. Once the approximation has been found, the maximum approximation error can be computed with function *suboptimalityAnalysis*, as explained in Bemporad et al. [2011].

• **PWAR controller:** the *pwarApproximation* function allows to approximate the *pwag* object by a *pwar* one by using the methods described in Genuit et al. [2012] and Lu et al. [2011]. The same constraints satisfied by the exact MPC law are also enforced in the approximate one, and constraints on stability and performance can be enforced depending on the method used. The routine needs the *pwag*, *linearSystem* and *constraints* objects and returns a *pwar* object. Some options can be provided to this function, such as the norm to use in the optimization problem and the solver to be employed. To facilitate setting these options, two functions are included. The *pwarApproxSettings* function can be used to create an options structure in which the non specified entries are filled with default values, whereas *pwarApproxSettingsCheck* can be used to check for inconsistencies and commonly made errors in the options structure.

5.3 Stability analysis of the closed-loop

Most of the described design methods for approximate MPC control law are aimed at practical implementation on digital circuits, and therefore do not incorporate conditions that guarantee the a-priori stability of the closed-loop system. For this reason, the function *stabilityTest* is provided. The method employed by this function is based on the use of PWA Lyapunov functions, which permits to certify the stability of the origin and to find an evaluation of the domain of attraction as a polytope. Also the presence of additive disturbances (not taken into account during the MPC design) can be considered. Since, in case of persistent disturbances, the convergence to the origin cannot be assured, *stabilityTest* will evaluate the set including the origin, where the state will be ultimately bounded. If the stability test fails, a new approximation can be obtained with a finer domain partition and the test can be repeated. The stability test can also be performed for the exact controller (not only for the approximate one) in case it is designed without stability guarantees. For more details, the reader is referred to Rubagotti et al. [2011, 2012].

5.4 Circuit implementation of digital controllers

One of the main features of MOBY-DIC Toolbox is the possibility of automatically generating VHDL code for the circuit implementation (on FPGA) of exact and approximate MPC controllers. Serial and parallel architectures are available for PWAG, PWAS and PWAR controllers, which allow to trade-off between circuit complexity and speed. A description of the architectures can be found in Oliveri et al. [2009], Storace and Poggi [2011], Comaschi et al. [2012] and in the toolbox documentation. The VHDL files are generated by resorting to method *synthesize* of a *pwag*, *pwas*, or *pwar* object once the circuit specifications (number of bits for inputs and outputs, type of input acquisition, circuit sampling time) have been provided. The synthesis process also generates a log file in which information about the circuit performances is shown: input/output representation, latency, number of multipliers, memory occupation. The generated files include also a testbench for the circuit so that it can be simulated by any VHDL simulator (e.g., Modelsim). The results of the simulation can be read by MOBY-DIC Toolbox which automatically performs the suitable data conversions and scalings.

5.5 Closed-loop simulation of controllers

Simulations of the closed-loop system with exact or approximate MPC controller can be performed in several ways with this toolbox. The discrete-time controlled system can be simulated in MATLAB by means of methods *sim* and *simplot* of *linearSystem* object. If the discrete-time system was obtained as a discretization of a continuous-time plant, for the purpose of generating the MPC control law, it could be useful to simulate directly the continuous-time plant in closed-loop with the exact or approximate controller. This task is accomplishable with method *generateSimulinkModel*, which automatically generates a Simulink model ready to be simulated. If the VHDL files have been generated for the controller, it is also possible (always through method *generateSimulinkModel*) to directly simulate them for the computation of the controller in the closed-loop system; this allows to take into account the effects of circuit latency (delays) and of the fixed point representation (quantization error). Xilinx System Generator is required for this functionality.

6. A SIMPLE CASE STUDY

In this section we illustrate the functioning of the toolbox in a very simple example, in order to focus on the capabilities of the software rather than on the control problem itself. MOBY-DIC Toolbox has been also used, however, in more complex applications such as Adaptive Cruise Control systems and Buck-Boost DC-DC converters (Oliveri et al. [2011], Spinu et al. [2012], Comaschi et al. [2012]). Let us consider the continuous-time double integrator system defined by the following state equations:

$$\dot{x}(t) = A_c x(t) + B_c u(t) \tag{5}$$

with $A_c = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$ and $B_c = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, subject to soft constraints

$$\begin{aligned} -1 \leq u \leq 1 \\ [-8 \ -4]' \leq x \leq [8 \ 4]' \end{aligned} \tag{6}$$

In order to compute the MPC control function we discretized system (5) with sampling time $T_s = 1$, thus obtaining:

$$x(t+1) = A_d x(t) + B_d u(t) \tag{7}$$

with $A_d = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$ and $B_d = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$. The system has 2 state variables ($n_x = 2$), one input variable ($n_u = 1$) and zero parameters $n_p = 0$. We computed the MPC control law with Hybrid Toolbox by setting $N = N_u = 5$, $Q = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$ and $R = 0.1$, thus obtaining a *pwag* object. The following snippet of code shows how these steps are performed:

```
% Create linearSystem object
linSys = linearSystem(nx,nu,np);
% Set discrete-time system matrices
linSys = linSys.setMatrices('A',Ad);
linSys = linSys.setMatrices('B',Bd);
% Set sampling time
linSys = linSys.setSamplingTime(Ts);

% Create constraints object
constr = constraints(nx,nu,np,N);
% Set input constraints from k to k+Nu-1
constr = constr.setConstraints('u1',-1,1,0:Nu-1);
% Set state constraints from k+1 to k+N
constr = constr.setConstraints('x1',-8,8,1:N);
constr = constr.setConstraints('x2',-4,4,1:N);

% MPC options
MPCoptions = struct( ...
    'toolbox','hybtbx', ... % Use Hybrid Toolbox
    'N',N, ...              % Prediction horizon
    'Nu',Nu, ...            % Control horizon
    'Q',Q, ...              % Cost function matrix Q
    'R',R, ...              % Cost function matrix R
);
% Generate pwag object representing the MPC control law
Cpwag = generateMPC(linSys,constr,MPCoptions);
```

Since closed-loop stability and constraint satisfaction are not guaranteed a priori, *stabilityTest* is employed to get a stability certificate for the origin, together with an estimate of its domain of attraction, shown in Fig. 4.

```
% Generate the closed-loop system with PWAG controller
CpwagCL = Cpwag.getClosedLoop(linSys);
% Check stability of the closed-loop system
[XPg XFg] = stabilityTest(CpwagCL);
```

Then we approximate the exact MPC control law with a PWAS function defined over a non-uniform simplicial partition and we compute the maximum approximation error $M = 0.385$ at point $x = [4.83 - 2.26]$.

```
% Choose simplicial partition (non-uniform)
P = cell(1,2);
P1 = [linspace(-8,-0.5,8) linspace(0.5,8,8)];
P2 = [-4 linspace(-2.275,-0.2,7) linspace(0.3,3.619,7) 4];
% Choose domain for the pwas function
D = Cpwag.getDomain(); % Use the same as the pwag function
```

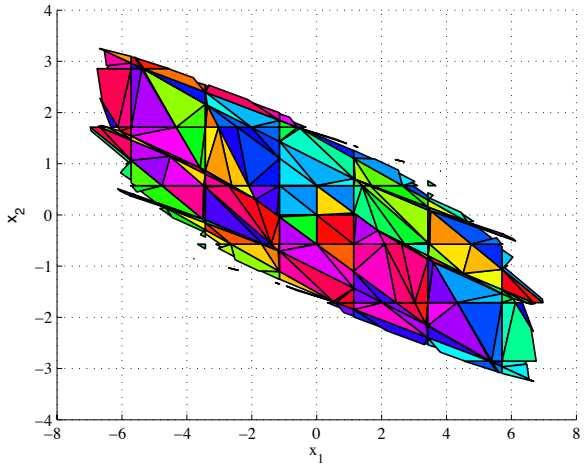

Fig. 4. Estimate of the domain of attraction of closed-loop system with PWAG controller.

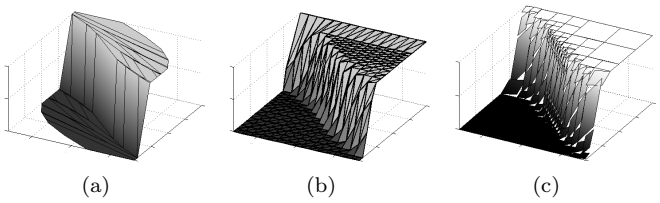

Fig. 5. Exact PWAG (a) and approximate PWAS(b) and PWAR(c) control laws.

```
% Approximation options
appr_opts = struct( ...
    'norm',2, ...           % Use 2 norm for approximation
    'solver','cvx', ...     % Use CVX solver
    'equality',2, ...       % Impose equality constraints
    'multithreading',1 ...  % Enable multithreading
);
% Perform the approximation
Cpwas = pwasApproximation(Cpwag,linSys,constr,D,P,appr_opts);

% Find maximum approximation error
[M x] = suboptimalityAnalysis(Cpwas,Cpwag);
```

Next, a multi-resolution PWAR approximation is obtained by imposing a maximum error of 0.24.

```
% Approximation options
appr_opts = pwarApproxSettings( ...
    'norm',inf, ...           % Use inf norm for approx.
    'rhomax',0.24, ...        % Maximum approx. error
    'con_input', 1, ...       % Input constraints enforced
    'resolutiontype', 'multi', ... % Type of resolution used
    'minlevel', 1, ...        % Minimal level of refinement
    'maxlevel', 5, ...        % Maximal level of refinement
    'solver', 'cdd', ...      % Use CDD solver
    'multithreading', 1 ..    % Enable multithreading
);
% Perform the approximation
Cpwar = pwarApproximation(Cpwag, linSys, constr, D, options);
```

The plots (obtained with method *plot*) of the exact and approximate control laws are shown in Fig. 5.

The following piece of code shows how to generate VHDL files for the FPGA implementations of exact and approximate controllers.

```
% Circuit specifications
cir_specs = struct( ...
 'nbit',12, ...                    % N. of bits for inputs
 'type','serial', ...              % Type of architecture
 'inputAcquisition','parallel', ... % Input acquisition
 'frequency',20e6, ...             % Working frequency
 'samplingTime',Ts, ...            % System sampling time
);

% Generate VHDL files for exact and approx. controllers
Cpwag = Cpwag.synthesize(cir_specs);
Cpwas = Cpwas.synthesize(cir_specs);
Cpwar = Cpwar.synthesize(cir_specs);
```

The circuit performance of both serial and parallel architectures, as provided by MOBY-DIC Toolbox, are shown in Table 1.

Circuit performance			
Architecture	Latency @20MHz	Mem. occup.	N. mult.
pwag_ser	1850 ns	175.5 bytes	1
pwag_par	950 ns	175.5 bytes	2
pwas_ser	400 ns	384 bytes	3
pwas_par	250 ns	1152 bytes	5
pwar_ser	450 ns	585 bytes	1
pwar_par	350 ns	585 bytes	2

Table 1. Performances of the circuit architectures implementing PWAG, PWAS and PWAR controllers

Finally, a Simulink model for the simulation of the closed-loop system considering circuit delays and fixed point representation has been generated.

```
% Model options
model_opts = struct( ...
    'simulateVHDL',1, ...   % Simulate VHDL files
    'samplingTime',Ts, ...  % Sampling time of the system
);
% Continuous time plant
plant = ss(Ac,Bc,eye(2),0);
% Initial condition for simulation
x0 = [-4 -2];
% Generate model for exact and approx. controllers
Cpwag.generateSimulinkModel(plant,x0,model_opts)
Cpwas.generateSimulinkModel(plant,x0,model_opts)
Cpwar.generateSimulinkModel(plant,x0,model_opts)
```

Figure 6 shows the results of the Simulink simulation and evidences the correct regulation of the system to the origin.[6] In case of approximate controllers, the design flow listed above corresponds to the one shown in figure 1(b). Indeed, the control law is derived taking into account both control specifications and circuit complexity either by setting the simplicial partition (i.e., the cell array P) for the PWAS approximation, or by setting the maximum level of refinement for the PWAR approximation. It is important to stress that the circuit performances are known before actually implementing the controller on FPGA and they can be known even before designing the controller itself. The user can therefore redesign or change the approximation parameters in order to meet circuit specifications. Further tools to ease this task are under development and will be available in a future release with a Graphical User Interface.

[6] Only the results obtained with the PWAS controller are shown for brevity. Almost identical plots have been obtained with PWAG and PWAR controllers.

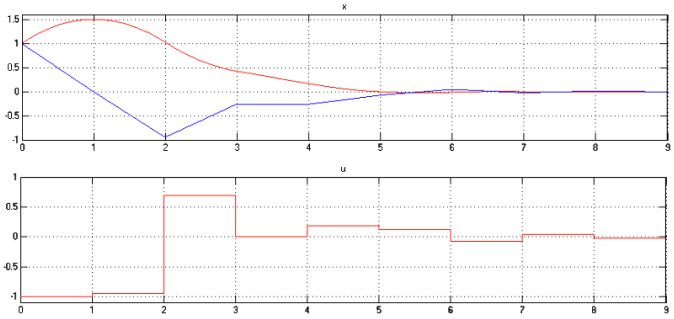

Fig. 6. Simulink simulations of the closed-loop system (plant and PWAS controller). The top panel shows the evolution of the states, the bottom panel the evolution of the input computed with VHDL simulation.

7. CONCLUSIONS AND FUTURE WORK

The paper introduces a complete software toolbox in MATLAB for the automatic generation of VHDL code describing embedded control systems, and illustrates its capabilities through a simple case study. The toolbox is still in phase of development and some capabilities will be added, such as the synthesis of a-priori stabilizing PWAS controllers. A complete Graphical User Interface (GUI) will be designed to facilitate the use of the toolbox and to graphically monitor the circuit performance as a function of control and circuit parameters.

REFERENCES

A. Alessio and A. Bemporad. A survey on explicit model predictive control. In *Nonlinear Model Predictive Control: Towards New Challenging Applications*. Springer Berlin / Heidelberg, 2009.

F. Bayat, T. A. Johansen, and A. A. Jalali. Using hash tables to manage the time-storage complexity in a point location problem: application to explicit model predictive control. *Automatica*, 47(3):571 – 577, 2011.

A. Bemporad. Hybrid Toolbox - User's Guide, 2004. `http://cse.lab.imtlucca.it/~bemporad/hybrid/toolbox`.

A. Bemporad, M. Morari, V. Dua, and E.N. Pistikopoulos. The explicit linear quadratic regulator for constrained systems. *Automatica*, 38(1):3–20, 2002.

A. Bemporad, A. Oliveri, T. Poggi, and M. Storace. Ultra-fast stabilizing model predictive control via canonical piecewise affine approximations. *IEEE Trans. Aut. Contr.*, 56(12):2883 –2897, 2011.

F. Comaschi, B. A. G. Genuit, A. Oliveri, W. P. M. H. Heemels, and M. Storace. FPGA implementations of piecewise affine functions based on multi-resolution hyperrectangular partitions. *IEEE Trans. Circ. Sys. I*, 2012. In press.

B. A. G. Genuit, L. Lu, and W. P. M. H. Heemels. Approximation of explicit model predictive control using regular piecewise affine functions: an input-to-state stability approach. *IET Contr. Th. Appl.*, 2012. In press.

M. Kvasnica, P. Grieder, and M. Baotić. Multi-Parametric Toolbox (MPT), 2004. URL `http://control.ee.ethz.ch/~mpt/`.

L. Lu, W. P. M. H Heemels, and A. Bemporad. Synthesis of low-complexity stabilizing piecewise affine controllers: A control-Lyapunov function approach. In *IEEE Conf. Dec. Contr.*, Orlando, FL, 2011.

A. Oliveri, A. Oliveri, T. Poggi, and M. Storace. Circuit implementation of piecewise-affine functions based on a binary search tree. In *Eur. Conf. Circuit Th. Des.*, 2009.

A. Oliveri, G.J.L. Naus, M. Storace, and W.P.M.H. Heemels. Low-complexity approximations of PWA functions: A case study on adaptive cruise control. In *Eur. Conf. Circuit Th. Des.*, 2011.

M. Parodi, M. Storace, and P. Juliàn. Synthesis of multiport resistors with piecewise-linear characteristics: a mixed-signal architecture. *Int. J. Circuit Th. Appl.*, 33(4):307–319, 2005.

T. Poggi, M. Rubagotti, A. Bemporad, and M. Storace. High-speed piecewise affine virtual sensors. *IEEE Trans. Ind. Elec.*, 59(2):1228 –1237, feb. 2012.

M. Rubagotti, S. Trimboli, D. Bernardini, and A. Bemporad. Stability and Invariance Analysis of Approximate Explicit MPC based on PWA Lyapunov Functions. In *Proc. IFAC World Congress*, Milan, Italy, 2011.

M. Rubagotti, S. Trimboli, and A. Bemporad. Stability and invariance analysis of uncertain discrete-time piecewise affine systems. *IEEE Trans. Aut. Contr.*, 2012. Conditionally accepted.

V. Spinu, A. Oliveri, M. Lazar, and M. Storace. FPGA implementation of optimal and approximate model predictive control for a buck-boost DC-DC converter. In *IEEE Multiconf. Sys. Contr.*, 2012. In press.

M. Storace and T. Poggi. Digital architectures realizing piecewise-linear multivariate functions: Two FPGA implementations. *Int. J. Circuit Th. Appl.*, 39(1):1–15, 2011.

P. Tøndel, T.A. Johansen, and A. Bemporad. Evaluation of piecewise affine control via binary search tree. *Automatica*, 39(5):945 – 950, 2003.