Control Software Visualization

Federico Mari, Igor Melatti, Ivano Salvo and Enrico Tronci
Department of Computer Science
Sapienza University of Rome
Via Salaria 113, 00198 Rome, Italy
Email: {mari,melatti,salvo,tronci}@di.uniroma1.it

***Abstract*—Many software as well digital hardware automatic synthesis methods define the set of implementations meeting the given system specifications with a boolean relation K (*controller*). Such relation, given a system state s and an action u, returns 1 iff taking action u in state s leads in the system goal or at least one step closer to it. In order to determine at hand if K is a "good" controller, e.g., if it covers a wide enough portion of the system state space, or to provide an high level view of the actions enabled by K, it is useful to picture K in a 2D or 3D diagram. In this paper, starting from a canonical representation for K, we propose an algorithm to automatically generate such a picture, relying on available graphing tools.**

***Keywords*-Control Software Visualization; Embedded Systems; Model Checking**

I. Introduction

Many *Embedded Systems* are indeed *Software Based Control Systems* (SBCSs). An SBCS consists of two main subsystems: the *controller* and the *plant*. Typically, the plant is a physical system consisting, for example, of mechanical or electrical devices whereas the controller consists of *control software* running on a microcontroller. In an endless loop, the controller reads *sensor* outputs from the plant and sends commands to plant *actuators* in order to guarantee that the *closed loop system* (that is, the system consisting of both plant and controller) meets given *safety* and *liveness* specifications (*System Level Formal Specifications*).

Software generation from models and formal specifications forms the core of *Model Based Design* of embedded software [1]. This approach is particularly interesting for SBCSs since in such a case system level (formal) specifications are much easier to define than the control software behavior itself.

The typical control loop skeleton for an SBCS is the following. Measure x of the system state from plant *sensors* goes through an *analog-to-digital* (AD) conversion, yielding a *quantized* value $\hat{x}$. A function *ctrlRegion* checks if $\hat{x}$ belongs to the region in which the control software works correctly. If this is not the case, a *Fault Detection, Isolation and Recovery* (FDIR) procedure is triggered; otherwise a function *ctrlLaw* computes a command $\hat{u}$ to be sent to plant *actuators* after a *digital-to-analog* (DA) conversion. Basically, the control software design problem for SBCSs consists in designing software implementing functions *ctrlLaw* and *ctrlRegion*.

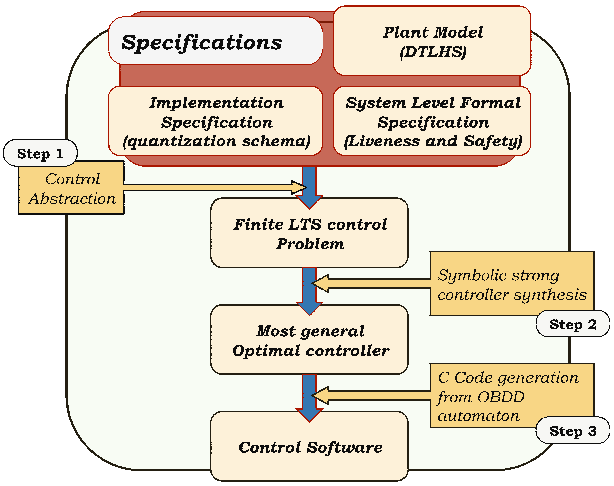

Figure 1. Control Software Synthesis Flow.

Automatic methods and tools aiming at synthesizing both functions *ctrlLaw* and *ctrlRegion* above have been developed in the last years, e.g., in [2][3][4][5][6][7]. In this paper, we will refer to the method described in [7], but the approach we describe may be applied to the other ones as well. Figure 1 shows the model based control software synthesis flow in [7]. A specification consists of a plant model, given as a Discrete Time Linear Hybrid System (DTLHS), System Level Formal Specifications that describe functional requirements of the closed loop system, and Implementation Specifications that describe non functional requirements of the control software, such as the number of bits used in the quantization process, the required worst case execution time, etc. Given such an input, in step 1 a suitable finite discrete abstraction (*control abstraction* [7]) $\hat{\mathcal{H}}$ of the DTLHS plant model $\mathcal{H}$ is computed; $\hat{\mathcal{H}}$ depends on the quantization schema and it is the plant as it can be seen from the control software after AD conversion. Then (step 2), given an abstraction $\hat{G}$ of the goal states G, it is computed a controller K that starting from any initial abstract state, drives $\hat{\mathcal{H}}$ to $\hat{G}$ regardless of possible nondeterminism. Control abstraction properties ensure that K is indeed a (quantized representation of a) controller for the original plant $\mathcal{H}$. Finally (step 3), the finite automaton K is translated into control software (C code).

In the following, we represent the control software with a boolean relation K (*controller*) taking as input (the n-bits

encoding of) a *state* x of the plant and (the r-bits encoding of) a proposed *action* to be performed u, and returns *true* (i.e., 1) iff the system specifications are met when performing action u in state x. In this approach, K is synthesized so that a given (*initial*) plant states region I (which is given as part of the system level formal specifications) is guaranteed to be covered by K. That is, for all states $x \in I$, there must exist at least an action u s.t. $K(x,u)$ holds. Typically, I is set to be small in order to increase the likelihood that a K fulfilling the above given property exists. However, the set of states covered by K, i.e., $\mathrm{dom}(K) = \{x \mid \exists u : K(x,u)\}$ may result to be much bigger than I. Therefore, once a K is built, it is useful to have a tool to graphically depict $\mathrm{dom}(K)$, in order to be able to visualize how big the region $\mathrm{dom}(K)$ is, as well as to have a glimpse of which actions are turned on by K on given plant states regions.

A. Our Main Contributions

In this paper we present an algorithm that, from an OBDD (*Ordered Binary Decision Diagram* [14]) representation of a controller K for a DTLHS modeling an SBCS, effectively generates a 2D picture (namely, an input file for Gnuplot [8]) depicting K. Such picture consists on a cartesian plane where each point corresponds to a state of the starting DTLHS, and shows as painted with the same color all regions of states for which the same *actions set* is defined on K. The color for a state (x,y) depends on which actions set is enabled by K in the DTLHS state (x,y), i.e., it is uniquely determined by $c(x,y) = \{u \mid K((x,y),u)\}$. As a special case, if $c(x,y) = \varnothing$ for some (x,y), i.e., (x,y) is not controlled by K, then the color is white. A separated picture showing the relation between a color and the corresponding actions set is also automatically generated. In this way, the state region for which any color is shown depicts the coverage of K, whilest the regions colors give a glimpse of which actions are turned on by K.

In our setting, since we seek K for which a software implementation is possible, a finite number of bits is used to encode both the states and the actions of the starting DTLHS. Suppose now that $|u| = r$, i.e., if r bits are needed in order to encode an action of the given DTLHS. Then, there may be at most 2^{2^r} different actions sets, i.e., $|\{c(x,y) \mid (x,y) \text{ is a state}\}| = 2^{2^r}$. That is, with $r = 5$ we need 4×10^9 colors, which is more than a typical RGB with 8 bits per color may achieve. Thus, our method may work only up to $r = 4$. Note however that this is not a limitation, since typical DTLHSs do not need more than 3 bits per action. Moreover, for most systems $|\{c(x,y) \mid (x,y) \text{ is a state}\}| << 2^{2^r}$, thus we may generate the picture even if $r \geq 5$.

We present experimental results showing effectiveness of the proposed algorithm. As an example, in about 1 hour we are able to generate the pairs of pictures described above for a multi-input buck DC/DC converter [9] with $r = 4$ action bit variables.

B. Paper outline

This paper is organized as follows. Section III provides the background needed to understand the results of this paper. Section IV describes our method to generate a picture visualizing a controller. Section V provides experimental results. Finally, Section VI summarizes and concludes the paper.

II. Related Work

Many papers (e.g., see [7][11][12][13]) tackling the problem of synthesizing control software (which looks to quantized states) or control laws (which looks at real states) of hybrid systems show pictures of the type we generate in this paper (with $r = 1$, i.e., only one bit for the actions). However, to the best of our knowledge there are no papers directly focusing on the method to generate such pictures, thus no automatic approach to controllers visualization is described.

Therefore, to the best of our knowledge this is the first time that an algorithm generating a picture of the coverage of a controller for a DTLHS is presented.

III. Basic Definitions

To make this paper self-contained, in this section we briefly summarize previous work on automatic generation of control software for *Discrete Time Linear Hybrid System* (DTLHS) from System Level Formal Specifications focusing on basic definitions and mathematical tools that will be useful in the sequel.

Figure 1 shows the control software synthesis flow that we consider here [7]. We model the controlled system (i.e., the plant) as a DTLHS (Section III-D), that is a discrete time hybrid system whose dynamics is modeled as a *linear predicate* (Section III-A) over a set of continuous as well as discrete variables. The semantics of a DTLHS is given in terms of a *Labeled Transition Systems* (LTS, Section III-C).

Given a plant $\mathcal{H}$ modeled as a DTLHS, a set of *goal states* G (*liveness specifications*) and an *initial region* I, both represented as linear predicates, we are interested in finding a *restriction* K *of the behaviour* of $\mathcal{H}$ such that in the *closed loop system* all paths starting in a state in I lead to G after a finite number of steps. Finding K is the DTLHS *control problem* (Section III-D) that is in turn defined as a suitable LTS control problem (Section III-C).

Finally, we are interested in controllers that take their decisions by looking at *quantized states*, i.e., the values that the control software reads after an AD conversion. This is the *quantized control problem*.

A. Predicates

We denote with $X = [x_1, \ldots, x_n]$ a finite sequence of variables. Each variable x ranges on a known (bounded or unbounded) interval $\mathcal{D}_x$ either of the reals or of the integers (discrete variables). We denote with $\mathcal{D}_X$ the set $\prod_{x \in X} \mathcal{D}_x$. Boolean variables are discrete variables ranging on the set $\mathbb{B}$

= $\{0, 1\}$. Unless otherwise stated, we suppose real variables to range on $\mathbb{R}$ and integer variables to range on $\mathbb{Z}$.

A *linear expression* over a list of variables X is a linear combination of variables in X with rational coefficients. A *linear constraint* over X (or simply a *constraint*) is an expression of the form $L(X) \leq b$, where $L(X)$ is a linear expression over X and b is a rational constant. Finally, a *conjunctive predicate* is conjunction of constraints.

B. OBDD Representation for Boolean Functions

We will denote boolean functions $f : \mathbb{B}^n \to \mathbb{B}$ with boolean expressions on boolean variables involving $+$ (logical OR), $\cdot$ (logical AND, usually omitted thus $xy = x \cdot y$), $^-$ (logical complementation) and $\oplus$ (logical XOR). We will also denote vectors of boolean variables in boldface, e.g., $\boldsymbol{x} = \langle x_1, \ldots, x_n \rangle$. Moreover, we also denote with $f|_{x_i=g}(\boldsymbol{x})$ the boolean function $f(x_1, \ldots, x_{i-1}, g(\boldsymbol{x}), x_{i+1}, \ldots, x_n)$ and with $\exists x_i \;\; f(\boldsymbol{x})$ the boolean function $f|_{x_i=0}(\boldsymbol{x}) + f|_{x_i=1}(\boldsymbol{x})$. A *truth assignment* μ is a partial map from a set of boolean variables $\mathcal{V}$ to $\mathbb{B}$. A *minterm* of μ is a total extension of μ, i.e., a total truth assignment ν s.t. $\mu(x) \neq \perp \rightarrow \nu(x) = \mu(x)$ for all $x \in \mathcal{V}$. The *value* of a minterm (or of a total truth assignment) ν is $\sum_{i=1}^{n} 2^{i-1}\nu(x_i)$, being $\mathcal{V} = \{x_1, \ldots, x_n\}$.

An *OBDD with complemented edges* (COBDD [14][15][16]) is a rooted directed acyclic graph (DAG) with the following properties. Each node v is labeled either with a boolean variable $\mathrm{var}(v)$ (an internal node) or with $1 \in \mathbb{B}$ (the unique terminal node **1**). Each internal node v has exactly two children, labeled with $\mathrm{high}(v)$ (representing the case in which $\mathrm{var}(v)$ is true) and $\mathrm{low}(v)$ ($\mathrm{var}(v)$ is false). Moreover, $\mathrm{low}(v)$ may be complemented, depending on a label $\mathrm{flip}(v)$ being true. Finally, on each path from the root to a terminal node, the variables labeling each internal node must follow the same ordering. The semantics of a COBDD internal node v w.r.t. a flipping bit b, with $\mathrm{var}(v) = x$, is the boolean function

$$[\![v, b]\!] := x[\![\mathrm{high}(v), b]\!] + \bar{x}[\![\mathrm{low}(v), b \oplus \mathrm{flip}(v)]\!]$$

C. Most General Optimal Controllers

A *Labeled Transition System* (LTS) is a tuple $\mathcal{S} = (S, A, T)$ where S is a finite set of states, A is a finite set of *actions*, and T is the (possibly non-deterministic) *transition relation* of $\mathcal{S}$. A *controller* for an LTS $\mathcal{S}$ is a function $K : S \times A \to \mathbb{B}$ enabling actions in a given state. We denote with $\mathrm{Dom}(K)$ the set of states for which a control action is enabled. An LTS *control problem* is a triple $\mathcal{P} = (\mathcal{S}, I, G)$, where $\mathcal{S}$ is an LTS and $I, G \subseteq S$. A controller K for $\mathcal{S}$ is a *strong solution* to $\mathcal{P}$ iff it drives each *initial* state $s \in I$ in a *goal* state $t \in G$, notwithstanding nondeterminism of $\mathcal{S}$. A strong solution K^* to $\mathcal{P}$ is *optimal* iff it minimizes path lengths. An optimal strong solution K^* to $\mathcal{P}$ is the *most general optimal controller* (we call such solution an *mgo*) iff in each state it enables all actions enabled by other optimal controllers. For more formal definitions of such concepts, see [7]. For efficient algorithms to compute mgos starting from suitable (nondeterministic) LTSs, i.e., see [17].

D. Discrete Time Linear Hybrid Systems

In this section we introduce the class of discrete time Hybrid Systems that we use as plant models, namely *Discrete Time Linear Hybrid Systems* (DTLHSs for short). For a more complete introduction, see [10].

Definition 1. A *Discrete Time Linear Hybrid System* is a tuple $\mathcal{H} = (X, U, Y, N)$ where: X is a finite sequence of *present state* variables (we denote with X' the sequence of *next state* variables obtained by decorating with $'$ all variables in X); U is a finite sequence of *input* variables; Y is a finite sequence of *auxiliary* variables; $N(X, U, Y, X')$ is a conjunctive predicate over $X \cup U \cup Y \cup X'$ defining the *transition relation* (*next state*) of the system. Note that X, U, Y may contain discrete as well as continuous variables.

DTLHSs may be used to represent many interesting real-world plants, such as e.g., the buck DC/DC converter with multi inputs used in Section V [9].

Given a DTLHS $\mathcal{H}$ = (X, U, Y, N), we define LTS($\mathcal{H}$) = ($\mathcal{D}_X$, $\mathcal{D}_U$, $\tilde{N}$) where: $\tilde{N} : \mathcal{D}_X \times \mathcal{D}_U \times \mathcal{D}_X \to \mathbb{B}$ is a function s.t. $\tilde{N}(x, u, x') \equiv \exists\, y \in \mathcal{D}_Y \;\; N(x, u, y, x')$. A *state* x for $\mathcal{H}$ is a state x for LTS($\mathcal{H}$). A DTLHS control problem $\mathcal{P} = (\mathcal{H}, I, G)$ is defined as the LTS control problem (LTS($\mathcal{H}$), I, G). To accommodate quantization errors, always present in software based controllers, it is useful to relax the notion of control solution by tolerating an (arbitrarily small) error ε on the continuous variables. Accordingly, we look for controllers that drive the plant to the goal G with an error at most ε (we call such a controller a *ε-solution* to $\mathcal{P}$). Such an error is defined by the given *quantization* for the DTLHS.

In classical control theory the concept of *quantization* has been introduced (e.g., see [18]) in order to manage real valued variables. Quantization is the process of approximating a continuous interval by a set of integer values. Formally, a *quantization function* γ for a real interval $I = [a, b]$ is a non-decreasing function $\gamma : I \mapsto \mathbb{Z}$ s.t. $\gamma(I)$ is a bounded integer interval. Finally, a *quantization* $\mathcal{Q} = (A, \Gamma)$ for a DTLHS encloses quantization functions Γ for all state variables as well as the bounded (safe) *admissible region* A on which the desired controller is supposed to work. Namely, A bounds both state variables (subregion A_X) on which the controller has to keep the system and action variables (subregion A_U) on which the controller works.

A control problem admits a *quantized* solution if control decisions can be made by just looking at quantized values. This enables a software implementation for a controller.

Definition 2. Given a quantization $\mathcal{Q}$, a *$\mathcal{Q}$ Quantized Feedback Control* (QFC) solution to a DTLHS control problem $\mathcal{P}$ is a $\|\Gamma\|$ solution $K(x, u)$ to $\mathcal{P}$ such that $K(x, u) =$

$\hat{K}(\Gamma(x), \Gamma(u))$, where $\hat{K} : \Gamma(A_X) \times \Gamma(A_U) \to \mathbb{B}$ and $\|\Gamma\|$ is the size of the largest interval of values that are mapped to the same quantized value.

For efficient (non-complete) algorithms to compute QFC solutions to a DTLHS control problem, e.g., see [7].

IV. Automatic Visualization of Control Software

In this section, we describe (Algorithms 1 and 2) our method to automatically generate a 2D picture describing a $\mathcal{Q}$ QFC solution K to a DTLHS control problem $\mathcal{P} = (\mathcal{H}, I, G)$ with a given quantization $\mathcal{Q} = (A, \Gamma)$.

The picture we generate lies on a 2D cartesian plane, where each axis is labeled with a state variable of $\mathcal{H}$ and has a range bounded by A. Then, a point (x, y) in the picture is colored depending on which actions set is enabled by K in the DTLHS state (x, y), i.e., on

$$c(x, y) = \{u \mid K((x, y), u) = 1\}$$

If $\mathcal{H}$ has $\ell+2$ state variables, then the actions set we consider is $c(x, y) = \{u \mid \exists d_1, \ldots, d_\ell K((x, y, d_1, \ldots, d_\ell), u) = 1\}$. Note that such a picture is practically useful if $\mathcal{H}$ has at least two real variables, which is indeed the case in most real-world SBCSs. Finally, a second picture showing the correspondence between actions sets and colors is also generated.

A. Input and Output

The above is performed by our main function *Visualize* (described in Algorithm 1), which takes as input:

- a DTLHS plant model $\mathcal{H} = (X, U, Y, N)$;
- a quantization $\mathcal{Q} = (A, \Gamma)$ for $\mathcal{H}$;
- a subset $\Xi \subseteq X$ of plant state variables s.t. $|\Xi| = 2$; variables in Ξ are those to be shown in the axes of the final 2D picture;
- a $\mathcal{Q}$ QFC solution K to a control problem involving $\mathcal{H}$. By Definition 2, K is based on a controller $\hat{K}$ that only looks at integer (quantized) values. Thus, by considering the boolean encoding of such values (as it is usual in Model Checking Applications), $\hat{K}$, and by abuse of notation K, can be represented as a COBDD ρ, a node v of ρ and a flipping bit b s.t. $[\![v, b]\!] = K$.

The output of *Visualize* is a Gnuplot [8] source files pair (P, C) describing the picture P to be generated and the color legend C. Note however that *Visualize* may be easily adjusted to work with any other graphing tool, provided that it generates pictures from textual descriptions. In Algorithm 1, we represent P as a list of rectangles in the plant state space (restricted to variables in Ξ). To each rectangle, we associate the RGB code of the corresponding color to be displayed. Analogously, C is a list of colored rectangles with height equal to the height of the picture: on the x axis the actions set corresponding to each colored rectangle is shown.

B. Algorithm Details

Function *Visualize* works as follows. First of all, in line 2, state bit variables encoding plant state variables not in Ξ (i.e., those *not* to be displayed in the final picture) are existentialized out from K, thus obtaining COBDD node v' and flipping bit b' such that $[\![v', b']\!] = \exists v_1, \ldots, v_\ell [\![v, b]\!] = \exists v_1, \ldots, v_\ell K = \tilde{K}$. As a result, the final picture will show all values for plant state variables in Ξ s.t. there exists at least a value for all plant state variables in $X \setminus \Xi$ that is controlled by K.

The workflow of the remaining lines is as follows. In order to obtain a better compression, controllers are typically represented with COBDDs where action bit variables come first in the variables ordering; this is also the case for [7]. In order to generate the desired picture, we reverse such order by placing state bit variables before action bit variables (line 4), thus obtaining a new COBDD ρ'. Since there always exists a COBDD representing a given boolean formula, in the new COBDD ρ' there will be a node v'' s.t. $[\![v'', b']\!] = \tilde{K}$. This allows us to perform a depth-first visit (DFS) of the COBDD representing $\tilde{K}$, by calling (line 5) function *CreateGnuplotBody* described in Algorithm 2. Namely, function *CreateGnuplotBody* returns a list M of (μ, v, b) triples s.t. μ is a total truth assignment to state bit variables with value $\hat{x}$, and for all plant states x in the quantized state $\hat{x}$ (i.e., such that $x \in \Gamma^{-1}(\hat{x})$) K enables the set of actions u s.t. the boolean encoding of u satisfies $[\![v, b]\!]$.

In order to achieve this goal, function *CreateGnuplotBody* of Algorithm 2 starts a depth-first visit (DFS) of ρ' from node v'' with flipping bit b'. On each path from v'' to **1**, the DFS stops as soon as an action bit variable is found at node z (i.e., $\mathrm{var}(z)$ is part of plant action variables U encoding) with flipping bit c. While exploring such a path, the corresponding truth assignment μ is maintained, i.e., if the then edge of a node w has been traversed, then $\mu(\mathrm{var}(w)) = 1$ (lines 5–6); if the else edge has been traversed, then $\mu(\mathrm{var}(w)) = 0$ (lines 7–9). Moreover, if a complemented edge is traversed, the flipping bit b is flipped (line 8). Once, in line 1, a node z is found s.t. $\mathrm{var}(z)$ is an action bit variable, or directly **1** is encountered (meaning that all actions are enabled by K for the quantized states corresponding to values of minterms of μ), the to-be-returned list M is updated (lines 2–3) by adding all minterms of the current μ together with the pair (z, b).

Once function *CreateGnuplotBody* has finished, the returned list M may be directly translated in a Gnuplot file P as follows. For each triple (μ, v, b) in M, the value $\hat{x}$ of μ is translated in a rectangle having as bounds those of $\Gamma^{-1}(\hat{x})$, i.e., of the cartesian product of the intervals that are mapped to $\hat{x}$ (line 10). The RGB color of such a rectangle may be determined starting from the address (a C language pointer) of (v, b). However, this has the following drawbacks: i) the Gnuplot file for the picture may be too big; ii) different runs

of function *Visualize* (e.g., with different quantizations, and thus different boolean encoding, for plant state variables) may result in different colors for equal actions sets, which may make difficult an effective comparison between different experiments. In order to counteract i), M is compacted, by collapsing contiguous quantized states with the same action sets (function *CompactRectangularRegions* in line 7 of Algorithm 1). To avoid ii), we first generate all possible 2^{2^r} colors (line 8, using an approach similar to [19]) and we use a lexicographical ordering on action sets to pick one of such colors. Finally, the Gnuplot file C maintaining the correspondence between colors and action sets is generated in lines 11–12, where SatAll returns all satisfying minterms of the given COBDD (boolean function).

Algorithm 1 Visualizing a controller.

Require: DTLHS $\mathcal{H}$, quantization $\mathcal{Q}$, state variables set Ξ s.t. $|\Xi| = 2$, COBDD ρ, node v, boolean b

Ensure: *Visualize*($\mathcal{H}, \Xi, \rho, v, b$):

1: let $v_1, \ldots, v_\ell$ be the state bit variables encoding plant variables in Ξ
2: let v', b' be s.t. $[\![v',\, b']\!] = \exists v_1, \ldots, v_\ell [\![v, b]\!]$
3: let $w_1, \ldots, w_r, w_{r+1}, \ldots, w_{n+r}$ be the current bit variables ordering in ρ, being r (resp. n) the number of action (state) bits variables
4: modify the ordering in $w_{r+1}, \ldots, w_{n+r}, w_1, \ldots, w_r$; call ρ' the resulting COBDD and v'' the node of ρ' s.t. $[\![v'', b']\!]_{\rho'} = [\![v', b']\!]_\rho$
5: $M \leftarrow$*CreateGnuplotBody*($\rho', v'', b', w_1, \bot, \varnothing$)
6: **for all** $i \in [|\alpha|]$ **do**
7: $\quad M \leftarrow$*CompactRectangularRegions*(M, i)
8: $\chi \leftarrow$*DifferentColorsRGB*(2^{2^r})
9: **for all** triples $(\mu, v, b) \in M$ **do**
10: $\quad$ using $\mathcal{Q}$, append to P the rectangle corresponding to μ with color $\chi_{\textit{lexOrder}(v,b)}$
11: **for all** (v, b) s.t. $\exists(\mu, v, b) \in M$ **do**
12: $\quad$ append to C a rectangle of color $\chi_{\textit{lexOrder}(v,b)}$ with label $\mathrm{SatAll}(\rho', v, b)$
13: **return** $\langle P, C\rangle$

V. Experimental Results

We implemented our picture generation algorithm in C programming language, using the CUDD package for OBDD based computations and BLIF files to represent input OBDDs. We name the resulting tool KPS (*Kontroller Picture Synthesizer*). KPS is part of a more general tool named QKS (*Quantized feedback Kontrol Synthesizer* [7]). In this section we present our experiments that aim at evaluating effectiveness of KPS.

1) Experimental Settings: We present experimental results obtained by using KPS on given COBDDs $\rho_1, \ldots, \rho_4$ and DTLHSs $\mathcal{H}_1, \ldots, \mathcal{H}_4$ s.t. for all $i \in [4]$ ρ_i represents the mgo $K_i(\boldsymbol{x}, \boldsymbol{u})$ for a *buck DC/DC converter with i inputs* (see [9] for a description of this system) modeled by $\mathcal{H}_i$, where quantization $\mathcal{Q}$ is s.t. $n = |\boldsymbol{x}| = 20$ and $r_i = |\boldsymbol{u}| = i$. K_i is an intermediate output of the QKS tool described in [7]. For each ρ_i, we run KPS so as to compute *Visualize*($\mathcal{H}_i$, $\mathcal{Q}$, X, ρ_i, v_i, b_i) (see Algorithm 1). All our experiments have been carried out on a 3.0 GHz Intel hyperthreaded Quad Core Linux PC with 8 GB of RAM.

Algorithm 2 Visualizing a controller: Gnuplot body.

Require: COBDD ρ, node v, boolean b, first action bit variable a, truth assignment μ, (assignment, COBDD node, flipping bit) triples set M

Ensure: *CreateGnuplotBody*(ρ, v, b, a, μ, M):

if $(v = \mathbf{1} \wedge \neg b) \vee (v \neq \mathbf{1} \wedge \mathrm{var}(v) > a)$ **then**
$\quad$ **for all** minterms ν of μ **do**
$\quad\quad M \leftarrow M \cup (\nu, v, b)$
else if $v \neq \mathbf{1}$ **then**
$\quad \mu(\mathrm{var}(v)) \leftarrow 1$
$\quad M \leftarrow$*CreateGnuplotBody*($\rho, \mathrm{high}(v), b, a, \mu, M$)
$\quad \mu(\mathrm{var}(v)) \leftarrow 0$
$\quad$ **if** $\mathrm{flip}(v)$ **then** $b \leftarrow \neg b$
$\quad M \leftarrow$*CreateGnuplotBody*($\rho, \mathrm{low}(v), b, a, \mu, M$)
return M

Table I
KPS performance (CPU times are in seconds).

r	CPU(P)	CPU(G)	$\lvert P\rvert$	$\lvert J\rvert$	$\lvert E\rvert$
1	9.15e+00	3.25e+02	6.17e+03	2.46e+01	5.19e+03
2	1.00e+01	1.47e+03	1.29e+04	2.91e+01	1.09e+04
3	1.06e+01	2.43e+03	1.67e+04	2.91e+01	1.39e+04
4	1.10e+01	3.58e+03	2.02e+04	3.16e+01	1.68e+04

2) KPS Performance: In this section we will show the performance (in terms of computation time and output size) of the algorithms discussed in Section IV. Table I show our experimental results. The i-th row in Table I corresponds to experiments running KPS so as to compute *Synthesize*($\mathcal{H}_i$, $\mathcal{Q}$, X, ρ_i, v_i, b_i). Columns in Table I have the following meaning. Column r shows the number of action variables $|\boldsymbol{u}|$ (note that $|\boldsymbol{x}| = 20$ on all our experiments). Column *CPU(P)* shows the computation time of KPS, i.e., of function *Visualize* of Algorithm 1 (in seconds). Columns $|P|$, $|J|$ and $|E|$ show the size in KB of, respectively, the source Gnuplot file for the 2D picture (i.e., the output P of function *Visualize* of Algorithm 1), the JPEG file generated by Gnuplot from P (i.e., with compression), and the EPS file generated by Gnuplot from P (i.e., without compression). Finally, Column *CPU(G)* shows the computation time of Gnuplot (in seconds) to generate the JPEG and the EPS files (computation time and size for file C are negligible).

From Table I we can see that, in slightly more than 10 seconds we are able to generate the Gnuplot file for the multi-input buck with $r = 4$ action variables. Then, Gnuplot needs about one hour to synthesize the actual picture (either in JPEG or in EPS).

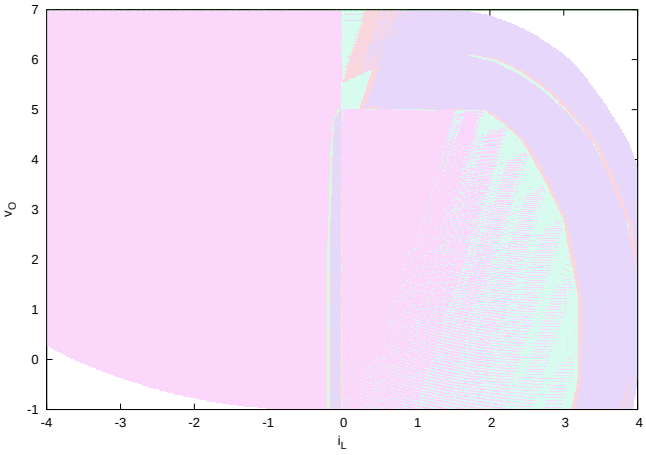

Figure 2. KPS+Gnuplot generated picture (P) for K_2.

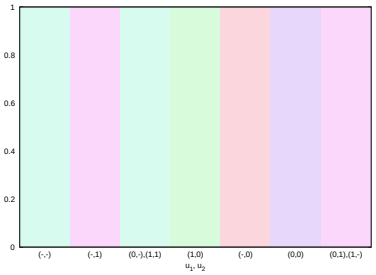

Figure 3. KPS+Gnuplot generated picture (C) for K_2.

3) KPS Evaluation: In Figures 2 and 3 we show the pictures generated by the KPS–Gnuplot chain for K_2. First of all, from Figure 3 we note that only 7 actions sets out of $2^{2^2} = 16$ are indeed enabled in K. Moreover, from Figure 2 we may immediately see that K indeed covers nearly all the admissible region of the buck converter. Finally, combining the two figures, we may see that the actions set $\{(-, 1)\}$ (i.e., $u_2 = 1$ and u_1 may be either 1 or 0) is the most used one.

VI. Conclusions

In this paper, we addressed the problem of visualizing a controller K for a DTLHS modeling an embedded system (plant). To this aim, we presented an algorithm and a tool KPS implementing it, which, from an OBDD representation of K, effectively generates a 2D picture depicting K. Such picture consists on a cartesian plane where each point corresponds to a state of the starting DTLHS, and colors with the same color all regions of states for which the same actions set is defined on K. A separated picture showing the relation between a color and the corresponding actions set is also automatically generated. In this way, the state region for which any color is shown depicts the coverage of K, whilest the regions colors give a glimpse of which actions are turned on by K on given plant states regions. We have shown feasibility of our proposed approach by presenting experimental results on using it to visualize the controller for a multi-input buck DC-DC converter.

The proposed approach currently generates a 2D picture, which forces to focus on just two plant state variables. Thus, a natural possible future research direction is to investigate how to generate a 3D picture. Finally, a 3D bar picture may also be used if there are more than 2 state variables in the input DTLHS plant, in order to show for each quantized value of the variables to be shown (i.e., those in Ξ) the percentage of coverage w.r.t. variables not to be shown (i.e., not in Ξ).

Acknowledgments: We are grateful to our anonymous referees for their helpful comments. Our work has been partially supported by: MIUR project DM24283 (TRAMP) and by the EC FP7 project GA218815 (ULISSE).

References

[1] T. A. Henzinger and J. Sifakis, "The embedded systems design challenge," in *FM'06*, LNCS 4085.
[2] T. Henzinger, P.-H. Ho, and H. Wong-Toi, "Hytech: A model checker for hybrid systems," *STTT*, 1(1), pp. 110–122, 1997.
[3] G. Frehse, "Phaver: algorithmic verification of hybrid systems past hytech," *STTT*, 10(3), pp. 263–279, 2008.
[4] H. Wong-Toi, "The synthesis of controllers for linear hybrid automata," in *CDC'97*, pp. 4607–4612.
[5] C. Tomlin, J. Lygeros, and S. Sastry, "Computing controllers for nonlinear hybrid systems," in *HSCC'99*, LNCS 1569.
[6] M. Mazo, A. Davitian, and P. Tabuada, "Pessoa: A tool for embedded controller synthesis," in *CAV'10*, LNCS 6174.
[7] F. Mari, I. Melatti, I. Salvo, and E. Tronci, "Synthesis of quantized feedback control software for discrete time linear hybrid systems," in *CAV'10*, LNCS 6174.
[8] "Gnuplot: http://www.gnuplot.info/," accessed: Jul 31, 2012.
[9] F. Mari, I. Melatti, I. Salvo, and E. Tronci, "On model based synthesis of embedded control software," in *EMSOFT'12*.
[10] F. Mari, I. Melatti, I. Salvo, E. Tronci. Quantized feedback control software synthesis from system level formal specifications. *CoRR*, abs/1107.5638v1, 2011.
[11] A. Girard, "Synthesis using approximately bisimilar abstractions: time-optimal control problems," in *CDC'10*.
[12] M. J. Mazo and P. Tabuada, "Symbolic approximate time-optimal control," *Systems & Control Letters*, 60(4), pp. 256–263, 2011.
[13] A. Girard, G. Pola, and P. Tabuada, "Approximately bisimilar symbolic models for incrementally stable switched systems," *IEEE Trans. on Aut. Contr.*, 55(1), pp. 116–126, 2010.
[14] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a bdd package," in *DAC'90*.
[15] S. Minato, N. Ishiura, and S. Yajima, "Shared binary decision diagram with attributed edges for efficient boolean function manipulation," in *DAC'90*, pp. 52–57.
[16] F. Mari, I. Melatti, I. Salvo, and E. Tronci, "From boolean relations to control software," in *ICSEA'11*.
[17] A. Cimatti, M. Roveri, and P. Traverso, "Strong planning in non-deterministic domains via model checking," in *AIPS'98*.
[18] M. Fu and L. Xie, "The sector bound approach to quantized feedback control," *IEEE Trans. on Automatic Control*, 50(11), pp. 1698–1711, 2005.
[19] "How to generate random colors programmatically: http://martin.ankerl.com/2009/12/09/how-to-create-random-colors-programmatically/," accessed: Jul 31, 2012.
[20] F. Mari, I. Melatti, I. Salvo, and E. Tronci, "Synthesis of quantized feedback control software for discrete time linear hybrid systems," in *CAV'10*, LNCS 6174.