Synthesizing Control Software from Boolean Relations

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. In such a context a fundamental step in the software (hardware) synthesis process is finding effective solutions to the functional equation defined by K. This entails finding a (set of) boolean function(s) F (typically represented using OBDDs, *Ordered Binary Decision Diagrams*) such that: 1) for all x for which K is satisfiable, $K(x, F(x)) = 1$ holds; 2) the implementation of F is efficient with respect to given implementation parameters such as code size or execution time. While this problem has been widely studied in digital hardware synthesis, little has been done in a software synthesis context. Unfortunately, the approaches developed for hardware synthesis cannot be directly used in a software context. This motivates investigation of effective methods to solve the above problem when F has to be implemented with software. In this paper, we present an algorithm that, from an OBDD representation for K, generates a C code implementation for F that has the same size as the OBDD for F and a worst case execution time linear in nr, being $n = |x|$ the number of input arguments for functions in F and r the number of functions in F. Moreover, a formal proof of the proposed algorithm correctness is also shown. Finally, we present experimental results showing effectiveness of the proposed algorithm.**

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

I. Introduction

Many software as well digital hardware automatic synthesis methods define the set of implementations meeting the given system specifications with a boolean relation K. Given an n-bits (resp., r-bits) *binary encoding* of *states* (resp., *actions*) of the system as it is usually done in Model Checking [7] (see Sect. III-B), such relation typically takes as input the n-bits encoding of a state x and the r-bits encoding of a proposed action to be performed u, and returns *true* (i.e., 1) if and only if the system specifications are met when performing action u in state x. In such a context, a fundamental step in the software (hardware) synthesis process is finding effective solutions to the functional equation defined by K, i.e., $K(x, u) = 1$. This entails finding a tuple of boolean functions $F = \langle f_1, \ldots, f_r \rangle$ (typically represented using OBDDs, *Ordered Binary Decision Diagrams* [2]) such that 1) for all x for which K is satisfiable (i.e., it enables at least one action), $K(x, F(x)) = 1$ holds, and 2) the implementation of F is efficient with respect to given implementation parameters such as code size or execution time.

While this problem has been widely studied in digital hardware synthesis [3][4], little has been done in a software synthesis context. This is not surprising since software synthesis from formal specifications is still in its infancy. Unfortunately the approaches developed for hardware synthesis cannot be directly used in a software context. In fact, synthesis methods targeting a hardware implementation typically aim at minimizing the number of digital gates and of hierarchy levels. Since in the same hierarchy level gates output computation is *parallel*, the hardware implementation WCET (*Worst Case Execution Time*) is given by the number of levels. On the other hand, a software implementation will have to *sequentially* compute the gates outputs. This implies that the software implementation WCET is the number of gates used, while a synthesis method targeting a software implementation may obtain a better WCET. This motivates investigation of effective methods to solve the above problem when F has to be implemented with software.

A. Our Contribution

In this paper, we present an algorithm that, from an OBDD representation for K, effectively generates a C code implementation for K that has the same size as the OBDD for F and a WCET linear in linear in nr, being $n = |x|$ the number of bits encoding state x and $r = |u|$ the number of bits encoding action u. This is done in two steps:

1) from an OBDD representation for K we effectively compute an OBDD representation for F, following the lines of [5];
2) we generate a C code implementation for F with the above described properties of code size and WCET.

We formally prove both steps 1 and 2 to be correct.

This allows us to synthesize correct-by-construction *control software*, provided that K is provably correct with respect to initial formal specifications. This is the case of [6], where an algorithm is presented to synthesize K starting from a) the formal specification of a Discrete-Time Linear Hybrid System (*DTLHS* in the following) modeling the system (plant) to be controlled, b) its system level formal specifications (specifying the goal to be reached and the safe states to be traversed in order to reach it)

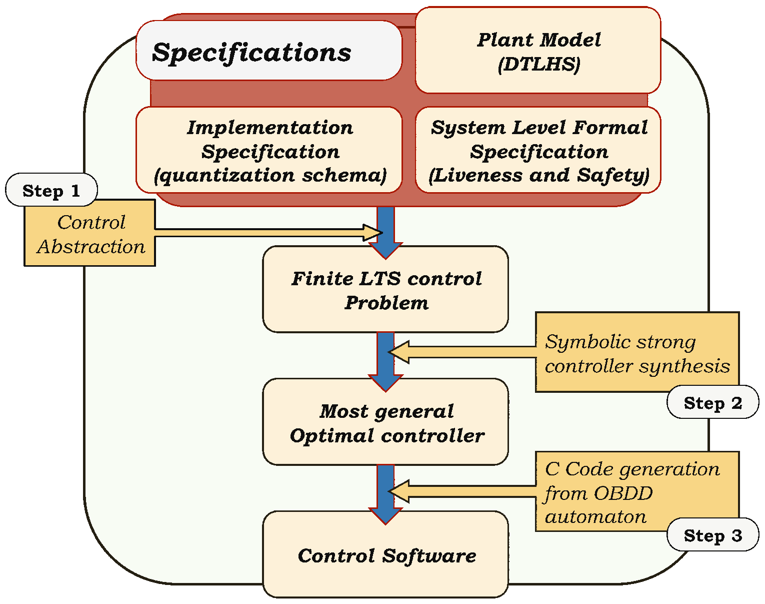
Specifications
Plant Model (DTLHS)
Implementation Specification (quantization schema)
System Level Formal Specification (Liveness and Safety)
Step 1
Control Abstraction
Finite LTS control Problem
Symbolic strong controller synthesis
Step 2
Most general Optimal controller
C Code generation from OBDD automaton
Step 3
Control Software

Figure 1. Control Software Synthesis Flow

and c) the quantization schema (i.e., the number of bits available for analog-to-digital conversion). The framework in [6] is depicted in Figure 1. With respect to Figure 1, the approach proposed in this paper may be used to perform step 3. Thus, this methodology allows a correct-by-construction control software to be synthesized, starting from formal specifications for DTLHSs.

Note that the problem of solving the functional equation $K(x, F(x)) = 1$ with respect to F is trivially decidable, since there are finitely many F. However, trying to explicitly enumerate all F requires time $\Omega(2^{r2^n})$. By using OBDD-based computations, we are able to compute F in time $O(r2^n)$ in the worst case. However, in many interesting cases OBDD sizes and computations are much lower than the theoretical worst case (e.g., in Model Checking applications, see [7]).

Furthermore, once the OBDD representation for F has been computed, a trivial implementation of F could use a look-up table in RAM. While this solution would yield a better WCET, it would imply a $\Omega(r2^n)$ RAM usage. Unfortunately, implementations for F in real-world cases are typically implemented on microcontrollers (this is the case, e.g., for *embedded systems*). Since microcontrollers usually have a small RAM, the look-up table based solution is not feasible in many interesting cases. The approach we present here will rely on OBDDs compression to overcome such obstruction.

Moreover, $F : \mathbb{B}^n \to \mathbb{B}^r$ is composed by r boolean functions, thus it is represented by r OBDDs. Such OBDDs typically share nodes among them. If a trivial implementation of F in C code is used, i.e., each OBDD is translated as a stand-alone C function, such inter-OBDDs nodes sharing will not be exploited. In our approach, we exploit inter-OBDDs nodes sharing, thus the control software we generate fully takes advantage of OBDDs compression.

Finally, we present experimental results showing effectiveness of the proposed algorithm. As an example, in less than 1 second and within 350 MB of RAM we are able to synthesize the control software for a function K of 25 boolean variables, divided in $n = 20$ state variables and $r = 5$ action variables, represented by an OBDD with about 6.6×10^4 nodes. Such K represents the set of correct implementations for a real-world system, namely a multi-input buck DC/DC converter [8], obtained as described in [6]. The control software we synthesize in such a case has about 1.7×10^4 lines of code, whilst a control software not taking into account OBDDs nodes sharing would have had about 2.1×10^4 lines of code. Thus, we obtain a 20% gain towards a trivial implementation.

This paper is organized as follows. In Section III we give the basic notions to understand our approach. In Section IV we formally define the problem we want to solve. In Section V we give definition and main properties of COBDDs (i.e., *Complemented edges OBDDs*), on which our approach is based. Section VI describes the algorithms our approach consists of, whilst Section VII proves it to be correct. Section VIII presents experimental results showing effectiveness of the proposed approach. Finally, Section IX presents the concluding remarks and gives some ideas for future work.

II. Related Work

This paper is an extended version of [1]. With respect to [1], this paper provides more details in the introduction and in the related work description, extends basic definitions and algorithms descriptions, shows omitted proofs for theorems and provides a revised version of the experiments.

Synthesis of boolean functions F satisfying a given boolean relation K in a way such that $K(x, F(x)) = 1$ is also addressed in [3]. However, [3] targets a hardware setting, whereas we are interested in a software implementation for F. Due to structural differences between hardware and software based implementations (see the discussion in Section I), the method in [3] is not directly applicable here. An OBDD-based method for synthesis of boolean (reversible) functions is presented in [4] (see also citations thereof). Again, the method in [4] targets a hardware implementation, thus it is not applicable here.

An algorithm for the synthesis of C control software is also presented in [9]. However, in [9] the starting point is a (multioutput) boolean function, rather than a boolean relation. That is to say, the starting point is F rather than K (with respect to the discussion in Section I-A, it is supposed that step 1 has already been performed). Moreover, the algorithm in [9], though OBDD-based, does not generate a software with the same size of the OBDDs for F, nor an estimation of its WCET (in the sense explained in Section I) is provided. Finally, an implementation of the algorithm in [9] is not provided, thus we cannot make a direct experimental comparison with our method.

Synthesis of control software is also addressed in [10], where the focus is on the generation of control protocols. Such method cannot be applied in our context, where we need a C software implementation.

In [6], an algorithm is presented which, starting from a formal specification of a DTLHS, synthesizes a correct-by-construction boolean relation K, and then a correct-by-construction control software implementation for K (see Figure 1). However, in [6] the implementation of K is not described in detail. Furthermore, the implementation synthesis described in [6] has not the same size of the OBDD for F, i.e., it does not exploit OBDD nodes sharing.

Many other works in the literature has the goal of synthesizing controllers as boolean relations K, under very different assumptions for the target dynamic system to be controlled. Such works do not deal with the effective implementation of K, thus they may use the approach described here in order to have an effective software implementation of K. As an example, the following works may be cited as closer to ours. In [11] controllers are generated starting from finite-state nondeterministic dynamic systems (arising from planning problems). In [12] a method to synthesize non-optimal (but smaller in size) controllers is presented.

In [5], an algorithm is presented which computes boolean functions F satisfying a given boolean relation K in a way such that $K(x, F(x)) = 1$. This approach is very similar to ours. However [5] does not generate the C code control software and it does not exploit OBDD nodes sharing.

Finally, we note that our work lies in the wider area of software synthesis, which has been widely studied since a long time in many contexts. For a survey on such (non-control) software synthesis works, see [13][14].

III. Basic Definitions

In the following, we denote with $\mathbb{B} = \{0, 1\}$ the boolean domain, where 0 stands for *false* and 1 for *true*. 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 $xy = x \cdot y$), $\bar{}$ (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 $f|_{x_i=0}(\boldsymbol{x}) + f|_{x_i=1}(\boldsymbol{x})$.

Finally, we denote with $[n]$ the set $\{1, \ldots, n\}$.

A. 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 $\mathcal{P}$ if and only if 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* if and only if it minimizes path lengths. An optimal strong solution K^* to $\mathcal{P}$ is the *most general optimal controller* (we call such solution an *mgo*) if and only if in each state it enables all actions enabled by other optimal controllers. For more formal definitions of such concepts, see [15].

Efficient algorithms, typically reminiscent of early work on minimum paths by Dijkstra [16], to compute controllers starting from suitable (nondeterministic) LTS control problems have been proposed in the literature: e.g., [11] presents an algorithm to generate mgos, while [12] show an algorithm for non-optimal (but smaller in size) controllers. Once a controller K has been computed, solving and implementing the functional equation $K(\boldsymbol{x}, \boldsymbol{u}) = 1$ allows a correct-by-construction control software to be synthesized.

B. Binary Encoding for States and Actions

Vectors of boolean values $\boldsymbol{x} \in \mathbb{B}^n$ (resp., $\boldsymbol{u} \in \mathbb{B}^r$) may be used to represent states $s \in S$ (resp., actions $a \in A$) of an LTS $\mathcal{S} = (S, A, T)$ (and thus of a controller for $\mathcal{S}$) as follows. Let $n = \lfloor \log_2(|S|) \rfloor + 1$. Then, n boolean values (bits) may be used to represent any $s \in S$. As an example, in Model Checking applications [7] an order on $S = \{s_1, \ldots, s_m\}$ is fixed (let $s_1 < \ldots < s_m$ be such order), and then the binary encoding $\eta : S \to \mathbb{B}^n$ is defined as $\eta(s_i) = \boldsymbol{b}$ such that $\sum_{j=1}^{n} 2^{j-1} b_j = i - 1$. An analogous construction may be applied to actions.

C. OBDD Representation for Boolean Functions

A *Binary Decision Diagram* (BDD) R is a rooted directed acyclic graph (DAG) with the following properties. Each R node v is labeled either with a boolean variable $\mathrm{var}(v)$ (internal node) or with a boolean constant $\mathrm{val}(v) \in \mathbb{B}$ (terminal node). Each R internal node v has exactly two children, labeled with $\mathrm{high}(v)$ and $\mathrm{low}(v)$. Let $x_1, \ldots, x_n$ be the boolean variables labeling R internal nodes. Each terminal node v represents $f_v(\boldsymbol{x}) = \mathrm{val}(v)$. Each internal node v represents $f_v(\boldsymbol{x}) = x_i f_{\mathrm{high}(v)}(\boldsymbol{x}) + \bar{x}_i f_{\mathrm{low}(v)}(\boldsymbol{x})$, being $x_i = \mathrm{var}(v)$. An *Ordered BDD* (OBDD) is a BDD where, on each path from the root to a terminal node, the variables labeling each internal node must follow the same ordering.

IV. Solving a Boolean Functional Equation

Let $K(x_1, \ldots, x_n, u_1, \ldots, u_r)$ be the mgo for a given control problem $\mathcal{P} = (\mathcal{S}, I, G)$. We want to solve the *boolean functional equation* $K(\boldsymbol{x}, \boldsymbol{u}) = 1$ with respect to variables $\boldsymbol{u}$, that is we want to obtain boolean functions $f_1, \ldots, f_r$ such that $K(\boldsymbol{x}, f_1(\boldsymbol{x}), \ldots, f_r(\boldsymbol{x})) = K|_{u_1 = f_1(\boldsymbol{x}), \ldots, u_r = f_r(\boldsymbol{x})}(\boldsymbol{x}, \boldsymbol{u}) = 1$. This problem may be solved in different ways, depending on the *target implementation* (hardware or software) for functions f_i. In both cases, it is crucial to be able to bound the WCET (*Worst Case Execution Time*) of the obtained controller. In fact, controllers must work in an endless closed loop with the system $\mathcal{S}$ (*plant*) they control. This implies that, every T seconds (*sampling time*), the controller has to determine the actions to be sent to $\mathcal{S}$. Thus, in order for the entire system (plant + control software) to properly work, the controller WCET upper bound must be at most T.

In [3], $f_1, \ldots, f_r$ are generated in order to optimize a *hardware* implementation. In this paper, we focus on software implementations for f_i (*control software*). As it is discussed in Section I, simply translating an hardware implementation into a software implementation would result in a too high WCET. Thus, a method directly targeting software is needed. An easy solution would be to set up, for a given state $\boldsymbol{x}$, a SAT problem instance $\mathcal{C} = C_{K1}, \ldots, C_{Kt}, c_1, \ldots, c_n$, where $C_{K1} \wedge \ldots \wedge C_{Kt}$ is equisatisfiable to K and each clause c_i is either x_i (if x_i is 1) or $\bar{x}_i$ (otherwise). Then $\mathcal{C}$ may be solved using a SAT solver, and the values assigned to $\boldsymbol{u}$ in the computed satisfying assignment may be returned as the action to be taken. However, it would be hard to estimate a WCET for such an implementation. The method we propose in this paper overcomes such obstructions by achieving a WCET proportional to rn.

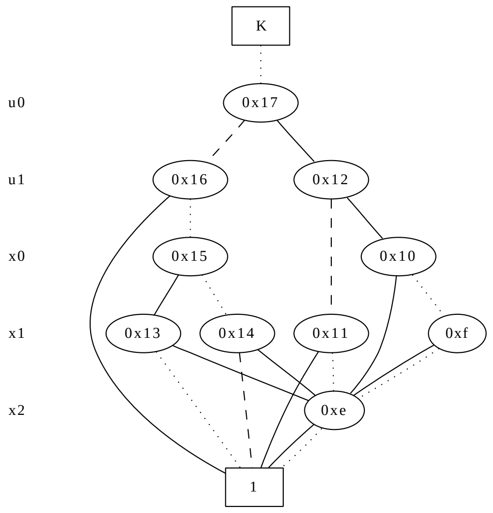

Figure 2. An mgo example

V. OBDDs with Complemented Edges

In this section, we introduce OBDDs with complemented edges (COBDDs, Definition 1), which were first presented in [17][18]. Intuitively, they are OBDDs where else edges (i.e., edges of type $(v, \mathrm{low}(v))$) may be complemented. Then edges (i.e., edges of type $(v, \mathrm{high}(v))$) complementation is not allowed to retain canonicity. Edge complementation usually reduce resources usage, both in terms of CPU and memory.

Definition 1. An *OBDD with complemented edges* (*COBDD* in the following) is a tuple $\rho = (\mathcal{V}, V, \mathbf{1}, \mathrm{var}, \mathrm{low}, \mathrm{high}, \mathrm{flip})$ with the following properties:

1) $\mathcal{V} = \{x_1, \ldots, x_n\}$ is a finite *ordered* set of boolean variables;
2) V is a finite set of *nodes*;
3) $\mathbf{1} \in V$ is the *terminal* node of ρ, corresponding to the boolean constant 1 (non-terminal nodes are called *internal*);
4) $\mathrm{var}, \mathrm{low}, \mathrm{high}, \mathrm{flip}$ are functions defined on internal nodes, namely:
   - $\mathrm{var} : V \setminus \{\mathbf{1}\} \to \mathcal{V}$ assigns to each internal node a boolean variable in $\mathcal{V}$
   - $\mathrm{high}[\mathrm{low}] : V \setminus \{\mathbf{1}\} \to V$ assigns to each internal node v a *high child* [*low child*] (or *then child* [*else child*]), representing the case in which $\mathrm{var}(v) = 1$ [$\mathrm{var}(v) = 0$]
   - $\mathrm{flip} : V \setminus \{\mathbf{1}\} \to \mathbb{B}$ assigns to each internal node v a boolean value; namely, if $\mathrm{flip}(v) = 1$ then the else child has to be complemented, otherwise it is regular (i.e., non-complemented);
5) for each internal node v, $\mathrm{var}(v) < \mathrm{var}(\mathrm{high}(v))$ and $\mathrm{var}(v) < \mathrm{var}(\mathrm{low}(v))$.

A. COBDDs Associated Multigraphs

We associate to a COBDD $\rho = (\mathcal{V}, V, \mathbf{1}, \text{var}, \text{low}, \text{high}, \text{flip})$ a labeled directed multigraph $G^{(\rho)} = (V, E)$ such that V is the same set of nodes of ρ and there is an edge $(v, w) \in E$ if and only if w is a child of v. Moreover, each edge $e = (v, w) \in E$ has a type $\text{type}(e)$, indicating if e is a *then edge* (i.e., if w is a then child of v), a *regular else edge* (i.e., if w is an else child of v with $\text{flip}(v) = 0$), or a *complemented else edge* (i.e., if w is an else child of v with $\text{flip}(v) = 1$). Figure 2 shows an example of a COBDD depicted via its associated multigraph, where edges are directed downwards. Moreover, in Figure 2 then edges are solid lines, regular else edges are dashed lines and complemented else edges are dotted lines.

The graph associated to a given COBDD $\rho = (\mathcal{V}, V, \mathbf{1}, \text{var}, \text{low}, \text{high}, \text{flip})$ may be seen as a forest with multiple rooted multigraphs. In order to select one root vertex and thus one rooted multigraph, we define the *COBDD restricted to* $v \in V$ as the COBDD $\rho_v = (\mathcal{V}, V_v, \mathbf{1}, \text{var}, \text{low}, \text{high}, \text{flip})$ such that $V_v = \{w \in V \mid \text{there exists a path from } v \text{ to } w \text{ in } G^{(\rho)}\}$ (note that $v \in V_v$).

B. COBDDs Properties

For a given COBDD $\rho = (\mathcal{V}, V, \mathbf{1}, \text{var}, \text{low}, \text{high}, \text{flip})$ the following properties follow from definitions given above:

1) $G^{(\rho)}$ is a rooted directed acyclic (multi)graph (DAG);
2) each path in $G^{(\rho)}$ starting from an internal node ends in $\mathbf{1}$;
3) let $v_1, \ldots, v_k$ be a path in $G^{(\rho)}$, then $var(v_1) < \ldots < var(v_k)$.

We define the *height of a node* v *in a COBDD* ρ (notation $\text{height}_\rho(v)$, or simply $\text{height}(v)$ if ρ is understood) as the height of the DAG $G^{(\rho_v)}$, i.e., the length of the longest path from v to $\mathbf{1}$ in $G^{(\rho)}$.

C. Semantics of a COBDD

In Definition 2, we define the semantics $[\![\cdot]\!]$ of each node v of a given COBDD ρ as the boolean function represented by v, given the parity b of complemented edges seen on the path from a root to v.

Definition 2. Let $\rho = (\mathcal{V}, V, \mathbf{1}, \text{var}, \text{low}, \text{high}, \text{flip})$ be a COBDD. The *semantics of the terminal node* $\mathbf{1}$ *with respect to a flipping bit* b is a boolean function defined as $[\![\mathbf{1}, b]\!]_\rho := \bar{b}$. The *semantics of an internal node* $v \in V$ *with respect to a flipping bit* b is a boolean function defined as $[\![v, b]\!]_\rho := x_i[\![\text{high}(v), b]\!]_\rho + \bar{x}_i[\![\text{low}(v), b \oplus \text{flip}(v)]\!]_\rho$, being $x_i = \text{var}(v)$. When ρ is understood, we will write $[\![\cdot]\!]$ instead of $[\![\cdot]\!]_\rho$.

Note that the semantics of a node of a COBDD ρ is a function of variables in $\mathcal{V}$ and of an additional boolean variable b. Thus, on each node *two* boolean functions on $\mathcal{V}$ are defined (one for each value of b). It can be shown (see [15]) that such boolean functions are complementary.

Example 1. *Let ρ be the COBDD depicted in Figure 2. If we pick node* 0xe *we have* $[\![0\text{xe}, b]\!] = x_2[\![\mathbf{1}, b]\!] + \bar{x}_2[\![\mathbf{1}, b \oplus 1]\!] = x_2\bar{b} + \bar{x}_2 b = x_2 \oplus b$.

D. Reduced COBDDs and COBDDs Canonicity

Two COBDDs are *isomorphic* if and only if there exists a mapping from nodes to nodes preserving attributes var, flip, high and low. A COBDD is called *reduced* if and only if it contains no vertex v with $\text{low}(v) = \text{high}(v) \wedge \text{flip}(v) = 0$, nor does it contains distinct vertices v and v' such that ρ_v and $\rho_{v'}$ are isomorphic. Note that, differently from OBDDs, it is possible that $\text{high}(v) = \text{low}(v)$ for some $v \in V$, provided that $\text{flip}(v) = 1$ (e.g., see nodes 0xf and 0xe in Figure 2).

Theorem 1 states that reduced COBDDs are a *canonical* representation for boolean functions (see [17][18]). As a consequence, software packages implementing COBDDs operations only deal with reduced COBDDs, since this allows very fast equality tests between COBDDs (it is sufficient to check if the (root node, flipping bit) pair is the same). Accordingly, in the following we will deal with reduced COBDDs only.

Theorem 1. *Let $f : \mathbb{B}^n \to \mathbb{B}$ be a boolean function. Then there exists a reduced COBDD* $\rho = (\mathcal{V}, V, \mathbf{1}, \text{var}, \text{low}, \text{high}, \text{flip})$, *a node $v \in V$ and a flipping bit $b \in \mathbb{B}$ such that* $[\![v, b]\!] = f(x)$. *Moreover, let* $\rho = (\mathcal{V}, V, \mathbf{1}, \text{var}, \text{low}, \text{high}, \text{flip})$ *be a reduced COBDD, let $v_1, v_2 \in V$ be nodes and $b_1, b_2 \in \mathbb{B}$ be flipping bits. Then $[\![v_1, b_1]\!] = [\![v_2, b_2]\!]$ if and only if $v_1 = v_2 \wedge b_1 = b_2$.*

VI. Synthesis of C Code from a COBDD

Let $K(x_1, \ldots, x_n, u_1, \ldots, u_r)$ be a controller for a given control problem. Let $\rho = (\mathcal{V}, V, \mathbf{1}, \text{var}, \text{low}, \text{high}, \text{flip})$ be a COBDD such that there exist $v \in V$, $b \in \mathbb{B}$ such that $[\![v, b]\!] = K(x_1, \ldots, x_n, u_1, \ldots, u_r)$. Thus, $\mathcal{V} = \mathcal{X} \uplus \mathcal{U} = \{x_1, \ldots, x_n\} \uplus \{u_1, \ldots, u_r\}$ (we denote with $\uplus$ the disjoint union operator, thus $\mathcal{X} \cap \mathcal{U} = \varnothing$). We will call variables $x_i \in \mathcal{X}$ as *state variables* and variables $u_j \in \mathcal{U}$ as *action variables*.

We want to solve the boolean functional equation problem introduced in Sect. IV targeting a *software* implementation. We do this by using a COBDD representing all our boolean functions. This allows us to exploit COBDD nodes sharing. This results in an improvement for the method in [5], which targets a software implementation but which does not exploit sharing. Finally, we also synthesize the software (i.e., C code) implementation for $f_1, \ldots, f_r$, which is not considered in [5]. This allows us to finally have a *control software* for the starting LTS. If K is an mgo, this results in an *optimal control software* for the starting LTS.

A. Synthesis Algorithm: Overview

Our method *Synthesize* takes as input ρ, v and b such that $[\![v, b]\!] = K(\boldsymbol{x}, \boldsymbol{u})$. Then, it returns as output a C function `void K(int *x, int *u)` with the following property: if, before a call to K, $\forall i$ `x[`$i-1$`]`$= x_i$ holds (array indexes in C language begin from 0) with $\boldsymbol{x} \in \mathrm{Dom}(K)$, and after the call to K, $\forall i$ `u[`$i-1$`]`$= u_i$ holds, then $K(\boldsymbol{x}, \boldsymbol{u}) = 1$. Moreover, the WCET of function K is $O(nr)$.

Note that our method *Synthesize* provides an effective *implementation* of the controller K, i.e., a C function which takes as input the current state of the LTS and outputs the action to be taken. Thus, K is indeed a control software.

Function *Synthesize* is organized in two phases. First, starting from ρ, v and b (thus from $K(\boldsymbol{x}, \boldsymbol{u})$), we generate COBDD nodes $v_1, \ldots, v_r$ and flipping bits $b_1, \ldots, b_r$ for boolean functions $f_1, \ldots, f_r$ such that each $f_i = [\![v_i, b_i]\!]$ takes as input the state bit vector $\boldsymbol{x}$ and computes the i-th bit u_i of an output action bit vector $\boldsymbol{u}$, where $K(\boldsymbol{x}, \boldsymbol{u}) = 1$, provided that $\boldsymbol{x} \in \mathrm{Dom}(K)$. This computation is carried out in function *SolveFunctionalEq*. Second, $f_1, \ldots, f_r$ are translated inside function `void K(int *x, int *u)`. This step is performed by maintaining the structure of the COBDD nodes representing $f_1, \ldots, f_r$. This allows us to exploit COBDD nodes sharing in the generated software. This phase is performed by function *GenerateCCode*.

Thus, function *Synthesize* is organized as in Algorithm 1. Correctness for function *Synthesize* is stated in Theorem 6.

Algorithm 1 Translating COBDDs to a C function

```
Require: COBDD $\rho$, node $v$, boolean $b$
Ensure: Synthesize($\rho, v, b$):
1: $\langle v_1, b_1, \ldots, v_r, b_r\rangle \leftarrow$ SolveFunctionalEq($\rho, v, b$)
2: GenerateCCode($\rho, v_1, b_1, \ldots, v_r, b_r$)
```

B. Synthesis Algorithm: Solving a Functional Equation

In this phase, starting from ρ, v and b (thus from $[\![v, b]\!] = K(\boldsymbol{x}, \boldsymbol{u})$), we compute functions $f_1, \ldots, f_r$ such that for all $\boldsymbol{x} \in \mathrm{Dom}(K)$, $K(\boldsymbol{x}, f_1(\boldsymbol{x}), \ldots, f_r(\boldsymbol{x})) = 1$.

To this aim, we follow an approach similar to the one presented in [5], which is reminiscent of early work on minimum paths by Dijkstra. Namely, we compute f_i using $f_1, \ldots, f_{i-1}$, in the following way: $f_i(\boldsymbol{x}) = \exists u_{i+1}, \ldots, u_n \quad K(\boldsymbol{x}, f_1(\boldsymbol{x}), \ldots, f_{i-1}(\boldsymbol{x}), 1, u_{i+1}, \ldots, u_n)$. Thus, function *SolveFunctionalEq*(ρ, v, b) computes and returns $\langle v_1, b_1, \ldots, v_r, b_r\rangle$ such that for all $i \in [r]$, $[\![v_i, b_i]\!] = f_i(\boldsymbol{x})$. This is effectively performed by Algorithm 2, where we use the following COBDDs manipulation functions:

- *COBDD_APP* (instantiation) such that $\langle v_{APP}, b_{APP}\rangle$ = *COBDD_APP*($x_{i_1}, \ldots, x_{i_k}, v_1, b_1, \ldots, v_k, b_k, v, b$) if and only if $[\![v_{APP}, b_{APP}]\!] = [\![v, b]\!]|_{x_{i_1} = [\![v_1, b_1]\!], \ldots, x_{i_k} = [\![v_k, b_k]\!]}$;
- *COBDD_EX* (existential quantifier elimination) such that $\langle v_{EX}, b_{EX}\rangle$ = *COBDD_EX*($x_{i_1}, \ldots, x_{i_k}, v, b$) if and only if $[\![v_{EX}, b_{EX}]\!] = \exists x_{i_1}, \ldots, x_{i_k}\ [\![v, b]\!]$.

We note that efficient (i.e., at most $O(|V| \log |V|)$) algorithms [17][18] exist to compute the above defined functions. Moreover, the above defined functions may create new COBDD nodes. We assume that such functions also properly update V, var, low, high, flip inside COBDD ρ ($\mathbf{1}$ and $\mathcal{V}$ are not affected).

Algorithm 2 Solving a boolean functional equation

```
Require: COBDD $\rho$, node $v$, boolean $b$
Ensure: SolveFunctionalEq($\rho, v, b$):
1: for all $i \in [r]$ do
2:   $[\![v_i, b_i]\!] \leftarrow$ COBDD_EX($u_{i+1}, \ldots, u_n$, COBDD_APP($u_1, \ldots, u_i, v_1, b_1, \ldots, v_{i-1}, b_{i-1}, \mathbf{1}, 0, v, b$))
3: return $\langle v_1, b_1, \ldots, v_r, b_r\rangle$
```

Correctness for function *SolveFunctionalEq* is proved in Lemma 3.

C. Synthesis Algorithm: Generating C Code

In this phase, starting from COBDD nodes $v_1, \ldots, v_r$ and flipping bits $b_1, \ldots, b_r$ for functions $f_1, \ldots, f_r$ generated in the first phase, we generate two C functions: i) `void K(int *x, int *u)`, which is the required output function for our method *Synthesize*; ii) `int K_bits(int *x, int action)`, which is an auxiliary function called by K. A call to `K_bits(x,` i`)` returns $f_i(\boldsymbol{x})$, being `x[`$j-1$`]`$= x_j$ for all $j \in [n]$. This phase is detailed in Algs. 3 (function *GenerateCCode*) and 4 (function *Translate*). In such algorithms we suppose to be able to print a node v, e.g., by printing the exadecimal value of a pointer to v.

Algorithm 3 Generating C functions

```
Require: COBDD $\rho$, $v_1, \ldots, v_r$, boolean values $b_1, \ldots, b_r$
Ensure: GenerateCCode($\rho, v_1, b_1, \ldots, v_r, b_r$):
1: print "int K_bits(int *x, int action) { int ret_b; switch(action) {"
2: for all $i \in [r]$ do
3:   print "case ", $i-1$, ":"
4:   print "ret_b = ", $\bar{b}_i$, "; goto L_", $v_i$, ";"
5: print "}" /* end of the switch block */
6: $W \leftarrow \varnothing$
7: for all $i \in [r]$ do
8:   $W \leftarrow$ Translate($\rho, v_i, W$)
9: print "} K(int* x, int* u) {int i;"
10: print "  for(i=0; i<", $r$, "; i++)"
11: print "    u[i] = K_bits(x, i);}"
```

Details of Function GenerateCCode (Algorithm 3): Given inputs ρ, v_1, b_1, $\ldots$, v_r, b_r (output by *SolveFunctionalEq*), Algorithm 3 works as follows. First, function `int K_bits(int *x, int action)` is generated. If `x[`$j-1$`]`$= x_j$ for all $j \in [n]$, the call `K_bits(x,` i`)` has to return $f_i(\boldsymbol{x})$. In order to do this, the graph $G^{(\rho_{v_i})}$ is traversed by taking, in each node v, the then edge if $\mathtt{x}[j-1] = 1$ (with j such that $\mathrm{var}(v) = x_j$) and the else edge otherwise. When node $\mathbf{1}$ is reached, then 1 is returned if and only if the integer sum $c+b_i$ is even, being c the number of complemented else edges traversed. Note that parity of $c+b_i$ may be maintained by initializing a C variable `ret_b` to $\bar{b}_i$, then complementing `ret_b` (i.e., by performing a `ret_b = !ret_b` statement) when a complemented else edge is traversed, and finally returning `ret_b`.

This mechanism is implemented inside function `K_bits` by properly translating each COBDD node $\tilde{v} \in \bigcup_{i=1}^{r} V_{v_i}$ in a C code block. Each block is labeled with a unique label depending on $\tilde{v}$, and maintains in variable `ret_b` the current parity of $c+b_i$ as described above. This is done by function *Translate*, called on line 8 and detailed in Algorithm 4.

Thus, the initial part of function `K_bits` consists of a `switch` block (generated in lines 1–5 of Algorithm 3), which initializes `ret_b` to $\bar{b}_i$ and then jumps to the label corresponding to node v_i. Then, the C code blocks corresponding to COBDD nodes are generated in lines 6–8 of Algorithm 3, by calling r times function *Translate* (see Algorithm 4) with parameters $v_1, \ldots, v_r$. Note that W maintains the already translated COBDD nodes. Since function *Translate* only translates nodes not in W, this allows us to exploit sharing not only inside each $G^{(\rho_{v_i})}$, but also inside $G^{(\rho_{v_1})}, \ldots, G^{(\rho_{v_r})}$.

Finally, function `K` is generated in lines 9–11. Function `K` simply consists in a `for` loop filling each entry `u[i]` of the output array `u` with the boolean values returned by `K_bits(x, i)`. Correctness of function *GenerateCCode* is proved in Lemma 5.

Algorithm 4 COBDD nodes translation

```
Require: COBDD $\rho$, node $v$, nodes set $W$
Ensure: Translate($\rho, v, W$):
 1: if $v \in W$ then return $W$
 2: $W \leftarrow W \cup \{v\}$
 3: print "L_", $v$, ":"
 4: if $v = \mathbf{1}$ then
 5:   print "return ret_b;"
 6: else
 7:   let $i$ be such that $\mathrm{var}(v) = x_i$
 8:   print "if(x[",$i-1$,"]==1)goto L_", $\mathrm{high}(v)$
 9:   if $\mathrm{flip}(v)$ then
10:     print "else {ret_b = !ret_b;''
11:     print "goto L_", $\mathrm{low}(v)$,";}"
12:   else
13:     print "else goto L_", $\mathrm{low}(v)$
14:   $W \leftarrow$ Translate($\rho, \mathrm{high}(v), W$)
15:   $W \leftarrow$ Translate($\rho, \mathrm{low}(v), W$)
16: return $W$
```

Details of Function Translate (Algorithm 4): Given inputs ρ, v, W, Algorithm 4 performs a recursive graph traversal of $G^{(\rho_v)}$ as follows.

The C code block for internal node v is generated in lines 3 and 7–13. The block consists of a label `L_`v`:` and an `if-then-else` C construct. Note that label `L_`v univocally identifies the C code block related to node v. This may be implemented by printing the exadecimal value of a pointer to v.

The `if-then-else` C construct is generated so as to traverse node v in graph $G^{(\rho_v)}$ in the following way. In line 8 the check `x[`$i-1$`]`$=1$ is generated, being i such that $\mathrm{var}(v) = x_i$. The code to take the then edge of v is also generated. Namely, it is sufficient to generate a `goto` statement to the C code block related to node $\mathrm{high}(v)$. In lines 10–11 and 13 the code to take the else edge is generated, in the case `x[`$i-1$`]`$=1$ is false. In this case, if the else edge is complemented, i.e., $\mathrm{flip}(v)$ holds (lines 10–11), it is necessary to complement `ret_b` and then perform a `goto` statement to the C code block related to node $\mathrm{low}(v)$ (lines 10–11). Otherwise, it is sufficient to generate a `goto` statement to the C code block related to node $\mathrm{low}(v)$ (line 13).

Thus, the block generated for an internal node v, for proper i, l and h, has one of the following forms, depending on $\mathrm{flip}(v)$:

- `L_`v`: if (x[`$i-1$`]) goto L_`h`; else goto L_`l`;`
- `L_`v`: if (x[`$i-1$`]) goto L_`h`; else {ret_b = !ret_b; goto L_`l`;}`.

There are two base cases for the recursion of function *Translate*:

- $v \in W$ (line 1), i.e., v has already been translated into a C code block as above. In this case, the set of visited COBDD nodes W is directly returned (line 1) without generating any C code. This allows us to retain COBDD node sharing;
- $v = \mathbf{1}$ (line 4), i.e., the terminal node $\mathbf{1}$ has been reached. In this case, the C code block to be generated is simply `L_1: return ret_b;`. Note that such a block will be generated only once.

In all other cases, function *Translate* ends with the recursive calls on the then and else edges (lines 14–15). Note that the visited nodes set W passed to the second recursive call is the result of the first recursive call. Correctness of function *Translate* is proved in Lemma 5.

D. An Example of Translation

Consider the COBDD ρ shown in Figure 2. Within ρ, consider mgo $K(x_0, x_1, x_2, u_0, u_1) = [\![0\text{x}17, 1]\!]$. By applying *SolveFunctionalEq*, we obtain $f_1(x_0, x_1, x_2) = [\![0\text{x}15, 1]\!]$ and $f_2(x_0, x_1, x_2) = [\![0\text{x}10, 1]\!]$. Note that 0xe is shared between $G^{(\rho_{0\text{x}15})}$ and $G^{(\rho_{0\text{x}10})}$. Finally, by calling *GenerateCCode* (see Algorithm 3) on f_1, f_2, we have the C code in Figure 3.

```
int K_bits(int *x, int action) {
  int ret_b;
  switch(action) {
    case 0: ret_b = 0; goto L_0x15;
    case 1: ret_b = 0; goto L_0x10;
  }
  L_0x15:
    if (x[0] == 1) goto L_0x13;
    else { ret_b = !ret_b; goto L_0x14;}
  L_0x13:
    if (x[1] == 1) goto L_0xe;
    else { ret_b = !ret_b; goto L_1; }
  L_0xe:
    if (x[2] == 1) goto L_1;
    else { ret_b = !ret_b; goto L_1; }
  L_0x14:
    if (x[1] == 1) goto L_0xe;
    else goto L_1;
  L_0x10:
    if (x[0] == 1) goto L_0xe;
    else { ret_b = !ret_b; goto L_0xf; }
  L_0xf:
    if (x[1] == 1) goto L_0xe;
    else { ret_b = !ret_b; goto L_0xe; }
  L_1:
    return ret_b;
}

void K(int *x, int *u) {
  int i;
  for(i = 0; i < 2; i++)
    u[i] = K_bits(x, i);
}
```

Figure 3. C code for the mgo in Figure 2 as generated by *Synthesize*

VII. Translation Proof of Correctness

In this section, we prove the correctness of our approach (Theorem 6). That is, we show that the function K we generate indeed implements the given controller K, thus resulting in a correct-by-construction control software.

We begin by stating four useful lemmata for our proof. Lemma 2 is useful to prove Lemma 3, i.e., to prove correctness of function *SolveFunctionalEq*.

Lemma 2. *Let* $K : \mathbb{B}^n \times \mathbb{B}^r \to \mathbb{B}$ *and let* $f_1, \ldots, f_r$ *be such that* $f_i(\boldsymbol{x}) = \exists u_{i+1}, \ldots, u_r\ K(\boldsymbol{x}, f_1(\boldsymbol{x}), \ldots, f_{i-1}(\boldsymbol{x}), 1, u_{i+1}, \ldots, u_r)$ *for all* $i \in [r]$. *Then,* $\boldsymbol{x} \in \mathrm{Dom}(K) \Rightarrow K(\boldsymbol{x}, f_1(\boldsymbol{x}), \ldots, f_r(\boldsymbol{x})) = 1$.

Proof: Let $\boldsymbol{x} \in \mathbb{B}^n$ be such that $\boldsymbol{x} \in \mathrm{Dom}(K)$, i.e., $\exists \boldsymbol{u}\ K(\boldsymbol{x}, \boldsymbol{u}) = 1$. We prove the lemma by induction on r. For $r = 1$, we have $f_1(\boldsymbol{x}) = K(\boldsymbol{x}, 1)$. If $f_1(\boldsymbol{x}) = 1$, we have $K(\boldsymbol{x}, f_1(\boldsymbol{x})) = K(\boldsymbol{x}, 1) = f_1(\boldsymbol{x}) = 1$. If $f_1(\boldsymbol{x}) = 0$, we have $K(\boldsymbol{x}, f_1(\boldsymbol{x})) = K(\boldsymbol{x}, 0)$, and $K(\boldsymbol{x}, 0) = 1$ since $\boldsymbol{x} \in \mathrm{Dom}(K)$ and $K(\boldsymbol{x}, 1) = 0$.

Suppose by induction that for all $\tilde{K} : \mathbb{B}^n \times \mathbb{B}^{r-1} \to \mathbb{B}$ $\tilde{K}(x, \tilde{f}_1(\boldsymbol{x}), \ldots, \tilde{f}_{r-1}(\boldsymbol{x})) = 1$, where for all $i \in [r-1]$ $\tilde{f}_i(\boldsymbol{x}) = \exists u_{i+1}, \ldots, u_{r-1}\ \tilde{K}(\boldsymbol{x}, \tilde{f}_1(\boldsymbol{x}), \ldots, \tilde{f}_{i-1}(\boldsymbol{x}), 1, u_{i+1}, \ldots, u_{r-1})$. We have that $\boldsymbol{x} \in \mathrm{Dom}(K)$ implies that either $\boldsymbol{x} \in \mathrm{Dom}(K|_{u_1=0})$ or $\boldsymbol{x} \in \mathrm{Dom}(K|_{u_1=1})$. Suppose $\boldsymbol{x} \in \mathrm{Dom}(K|_{u_1=1})$ holds. We have that $K|_{u_1=1}(\boldsymbol{x}, \tilde{f}_2(\boldsymbol{x}), \ldots, \tilde{f}_r(\boldsymbol{x})) = 1$, where for all $i = 2, \ldots, r$ $\tilde{f}_i(\boldsymbol{x}) = \exists u_{i+1}, \ldots, u_r\ K|_{u_1=1}(\boldsymbol{x}, \tilde{f}_2(\boldsymbol{x}), \ldots, \tilde{f}_{i-1}(\boldsymbol{x}), 1, u_{i+1}, \ldots, u_r)$. By construction, we have that $f_1(\boldsymbol{x}) = 1$ and $f_i(\boldsymbol{x}) = \tilde{f}_i(\boldsymbol{x})$ for $i \geq 2$, thus $1 = K|_{u_1=1}(\boldsymbol{x}, \tilde{f}_2(\boldsymbol{x}), \ldots, \tilde{f}_r(\boldsymbol{x})) = K(\boldsymbol{x}, f_1(\boldsymbol{x}), \ldots, f_r(\boldsymbol{x}))$. Analogously, if $x \notin \mathrm{Dom}(K|_{u_1=1}) \wedge \boldsymbol{x} \in \mathrm{Dom}(K|_{u_1=0})$ we have that $f_1(\boldsymbol{x}) = 0$ and $f_i(\boldsymbol{x}) = \tilde{f}_i(\boldsymbol{x})$ for $i \geq 2$, thus $1 = K|_{u_1=0}(\boldsymbol{x}, \tilde{f}_2(\boldsymbol{x}), \ldots, \tilde{f}_r(\boldsymbol{x})) = K(\boldsymbol{x}, f_1(\boldsymbol{x}), \ldots, f_r(\boldsymbol{x}))$.

■

Lemma 3 states correctness of function *SolveFunctionalEq* of Algorithm 2.

Lemma 3. *Let* $\rho = (\mathcal{V}, V, \mathbf{1}, \mathrm{var}, \mathrm{low}, \mathrm{high}, \mathrm{flip})$ *be a COBDD with* $\mathcal{V} = \mathcal{X} \cup \mathcal{U}$, $v \in V$ *be a node,* $b \in \mathbb{B}$ *be a flipping bit. Let* $[\![v, b]\!] = K(\boldsymbol{x}, \boldsymbol{u})$ *and* $r = |\mathcal{U}|$. *Then function SolveFunctionalEq*(ρ, v, b) *(see Algorithm 2) outputs nodes* $v_1, \ldots, v_r$ *and boolean values* $b_1, \ldots, b_r$ *such that for all* $i \in [r]$ $[\![v_i, b_i]\!] = f_i(\boldsymbol{x})$ *and* $\boldsymbol{x} \in \mathrm{Dom}(K)$ *implies* $K(\boldsymbol{x}, f_1(\boldsymbol{x}), \ldots, f_r(\boldsymbol{x})) = 1$.

Proof: Correctness of functions *COBDD_APP* and *COBDD_EX* (and lemma hypotheses) implies that for all $i \in [r]$ $f_i(\boldsymbol{x}) = \exists u_{i+1}, \ldots, u_r\ \ K(\boldsymbol{x}, f_1(\boldsymbol{x}), \ldots, f_{i-1}(\boldsymbol{x}), 1, u_{i+1}, \ldots, u_r)$. By Lemma 2 we have the thesis.

■

Let *Translate_dup* be a function that works as function *Translate* of Algorithm 4, but that does not take nodes sharing into account. Function *Translate_dup* may be obtained from function *Translate* by deleting line 1 (highlighted in Algorithm 4) and by replacing calls to *Translate* in lines 14 and 15 with recursive calls to *Translate_dup* (with no changes on parameters). Lemma 4 states correctness of function *Translate_dup*.

Lemma 4. *Let* $\rho = (\mathcal{V}, V, \mathbf{1}, \mathrm{var}, \mathrm{low}, \mathrm{high}, \mathrm{flip})$ *be a COBDD,* $v \in V$ *be a node,* $b \in \mathbb{B}$ *be a flipping bit, and* $W \subseteq V$ *be a set of nodes. Then function Translate_dup*(ρ, v, W) *generates a sequence of labeled C statements* $B_1 \ldots B_k$ *such that* $k \geq |V_v|$ *and for all* $w \in V_v$*: 1) label* L_w *is in* B_i *for some* i *and 2) starting an execution from label* L_w *with* $\forall i \in [n]$ x[$i-1$]= x_i *and* ret_b= $\bar{b}$, *if* $[\![w, b]\!] = f_{w,b}$ *then a* `return ret_b;` *statement is invoked in at most* $O(p)$ *steps with* ret_b $= f_{w,b}(\boldsymbol{x})$ *and* $p = \mathrm{height}(w)$.

Proof: We prove this lemma by induction on v. Let $v = \mathbf{1}$, which implies $[\![v, b]\!] = \bar{b}$ and $V_v = \{\mathbf{1}\}$. We have that function *Translate_dup*(ρ, v, W) generates a single block B_1 (thus $k = 1 = |V_1|$) such that B_1 =`L_1: return ret_b;` (lines 3–5 of Algorithm 4). Since by hypothesis we have ret_b= $\bar{b}$, and since starting from B_1 the return statement is invoked in $O(1)$ steps, the base case of the induction is proved.

Let v be an internal node with $\mathrm{var}(v) = x_i$ and let $f(\boldsymbol{x}) = [\![v, b]\!]$. Since $w \in V_v$ if and only if $w = v \vee w \in V_{\mathrm{high}(v)} \vee w \in V_{\mathrm{low}(v)}$, by induction hypothesis we only have to prove the thesis for $w = v$. We have that $f(\boldsymbol{x}) = x_i[\![\mathrm{high}(v), b]\!] + \bar{x}_i[\![\mathrm{low}(v), b \oplus \mathrm{flip}(v)]\!]$, i.e., $f(\boldsymbol{x}) = x_i[\![\mathrm{high}(v), b]\!] + \bar{x}_i[\![\mathrm{low}(v), b]\!]$ if $\mathrm{flip}(v) = 0$ and $f(\boldsymbol{x}) = x_i[\![\mathrm{high}(v), b]\!] + \bar{x}_i[\![\mathrm{low}(v), \bar{b}]\!]$ if $\mathrm{flip}(v) = 1$. Since $f(\boldsymbol{x}) = x_i f|_{x_i=1}(\boldsymbol{x}) + \bar{x}_i f|_{x_i=0}(\boldsymbol{x})$, by Theorem 1 we have that $[\![\mathrm{high}(v), b]\!] = f|_{x_i=1}(\boldsymbol{x})$, and that $[\![\mathrm{low}(v), b]\!] = f|_{x_i=0}(\boldsymbol{x})$ if $\mathrm{flip}(v) = 0$ and $[\![\mathrm{low}(v), \bar{b}]\!] = f|_{x_i=0}(\boldsymbol{x})$ if $\mathrm{flip}(v) = 1$.

By lines 3 and 8–13 of Algorithm 4, we have that function *Translate_dup*(ρ, v, W) generates blocks $BB_{11} \ldots B_{1h}B_{21} \ldots B_{2l}$ such that $B =$ `L_`v`: if (x[`$i-1$`] == 1) goto L_`$\mathrm{high}(v)$`; else` B_E where B_E is either `goto L_`$\mathrm{low}(v)$`;` if $\mathrm{flip}(v) = 0$ or `{ret_b = !ret_b; goto L_`$\mathrm{low}(v)$`;}` if $\mathrm{flip}(v) = 1$, and $B_{11} \ldots B_{1h}$ $(B_{21} \ldots B_{2l})$ are generated by the recursive call *Translate_dup*$(\rho, \mathrm{high}(v), W)$ in line 14 (*Translate_dup*$(\rho, \mathrm{low}(v), W)$ in line 15). By induction hypothesis and the above reasoning, if the execution starts at label `L_`$\mathrm{high}(v)$ and `ret_b=` $\bar{b}$, then a `return ret_b;` statement is invoked in at most $O(p-1)$ steps with `ret_b` $= f|_{x_i=1}(\boldsymbol{x})$. As for the else case, we have that starting from `L_`$\mathrm{low}(v)$ with `ret_b=` $\bar{b}$ (`ret_b=` $\bar{\bar{b}}$) if $\mathrm{flip}(v) = 0$ ($\mathrm{flip}(v) = 1$), then a `return ret_b;` statement is invoked in at most $O(p-1)$ steps with `ret_b` $= f|_{x_i=0}(\boldsymbol{x})$. By construction of block B, starting from label `L_`v, a `return ret_b;` statement is invoked in at most $O(p-1+1) = O(p)$ steps with `ret_b` $= x_i f|_{x_i=1}(\boldsymbol{x}) + \bar{x}_i f|_{x_i=0}(\boldsymbol{x}) = f(\boldsymbol{x})$. Finally, note that by induction hypothesis $h \geq |V_{\mathrm{high}(v)}|$ and $l \geq |V_{\mathrm{low}(v)}|$, thus we have that $k = 1 + h + l \geq 1 + |V_{\mathrm{high}(v)}| + |V_{\mathrm{low}(v)}| \geq |V_v|$.

■

Lemma 5 extends Lemma 4 by also considering nodes sharing, thus stating correctness of function *GenerateCCode* of Algorithm 3 and function *Translate* of Algorithm 4.

Lemma 5. *Let* $\rho = (\mathcal{V}, V, \mathbf{1}, \mathrm{var}, \mathrm{low}, \mathrm{high}, \mathrm{flip})$ *be a COBDD and* $v_1, \ldots, v_r \in V$ *be* r *nodes and* $b_1, \ldots, b_r \in \mathbb{B}$ *be* r *flipping bits. Then lines 6–8 of function GenerateCCode*$(\rho, v_1, b_1, \ldots, v_r, b_r)$ *generate a sequence of labeled C statements* $B_1 \ldots B_k$ *such that* $k = |\cup_{i=1}^{r} V_{v_i}|$ *and for all* $v \in \cup_{i=1}^{r} V_{v_i}$*: 1) the label* `L_`$v$ *is in* B_j *for some* j *and 2) starting an execution from label* `L_`v *with* $\forall j \in [n]$ `x[`$j-1$`]=` x_j *and* `ret_b=` $\bar{b}$*, if* $[\![v, b]\!] = f_{v,b}$ *then a* `return ret_b;` *statement is invoked in at most* $O(p)$ *steps with* `ret_b` $= f_{v,b}(\boldsymbol{x})$ *and* $p = \mathrm{height}(w)$.

Proof: We begin by proving that $k = |\cup_{i=1}^{r} V_{v_i}|$. To this aim, we prove that for each node $v \in \cup_{i=1}^{r} V_{v_i}$, a unique block B_v is generated. This follows by how the nodes set W is managed by function *Translate* in lines 1–3 of Algorithm 4 and by function *GenerateCCode* in lines 6–8 of Algorithm 3. In fact, function *Translate*, when called on parameters ρ, v, W, returns a set $W' \supseteq W$, and function *GenerateCCode* calls *Translate* by always passing the W resulting by the previous call. Since a block is generated for node v only if v is not in W, and v is added to W only when a block is generated for node v, this proves this part of the lemma.

As for correctness, we prove this lemma by induction on m, being m the number of times that the `return` W`;` statement in line 1 of Algorithm 4 is executed. As base of the induction, let $m = 1$ and let ρ, v, W be the parameters of the recursive call executing the first `return` W`;` statement. Then, by construction of function *Translate*, v has been added to W in some previous recursive call with parameters $\rho, v, \tilde{W}$. In this previous recursive call, a block B_v with label `L_`v has been generated. Moreover, for this previous recursive call, thus for parameters $\rho, v, \tilde{W}$, we are in the hypothesis of Lemma 4, which implies that the induction base is proved.

Suppose now that the thesis holds for the first m executions of the `return` W`;` statement in line 1 of Algorithm 4. Then, by construction of function *Translate*, v has been added to W in some previous recursive call with parameters $\rho, v, \tilde{W}$. In this previous recursive call, a block B_v with label `L_`v has been generated. Let $w_1, W_1, \ldots, w_m, W_m$, be such that the m recursive calls executing the `return` W`;` statement have parameters ρ, v_i, W_i (note that they are not necessarily distinct). By induction hypothesis, for all $i \in [m]$ starting from label `L_`w_i with $\forall j \in [n]$ `x[`$j-1$`]=` x_j and `ret_b=` $\bar{b}$, a `return ret_b;` statement is invoked in at most $O(p)$ steps with `ret_b` $= f_{w_i,b}(\boldsymbol{x})$. By Lemma 4 and its proof, the same holds for all $v \in V_v \setminus \{w_1, \ldots, w_m\}$, thus it holds for all $v \in V_v$.

■

Finally, Theorem 6 states and proves correctness for function *Synthesize* of Algorithm 1.

Theorem 6. *Let* $\rho = (\mathcal{V}, V, \mathbf{1}, \mathrm{var}, \mathrm{low}, \mathrm{high}, \mathrm{flip})$ *be a COBDD with* $\mathcal{V} = \mathcal{X} \cup \mathcal{U}$*,* $v \in V$ *be a node,* $b \in \mathbb{B}$ *be a boolean. Let* $[\![v, b]\!] = K(\boldsymbol{x}, \boldsymbol{u})$*,* $r = |\mathcal{U}|$ *and* $n = |\mathcal{X}|$*. Then function Synthesize*(ρ, v, b) *generates a C function* `void K(int *x, int *u)` *with the following property: for all* $\boldsymbol{x} \in \mathrm{Dom}(K)$*, if before a call to* `K` $\forall i \in [n]$ `x[`$i-1$`]=` x_i*, and after the call to* `K` $\forall i \in [r]$ `u[`$i-1$`]=` u_i*, then* $K(\boldsymbol{x}, \boldsymbol{u}) = 1$.

Furthermore, function `K` *has WCET* $\sum_{i=1}^{r} O(\mathrm{height}(v_i))$*, being* $v_1, \ldots, v_r$ *the nodes output by function SolveFunctionalEq.*

Proof: Let $\boldsymbol{x} \in \mathrm{Dom}(K)$ (i.e., $\exists \boldsymbol{u}\ K(\boldsymbol{x}, \boldsymbol{u}) = 1$) and suppose that for all $j \in [n]$ `x[`$j-1$`]=` x_j. By lines 9–11 of Algorithm 3, for all $i \in [r]$, `u[`$i-1$`]` will take the value returned by `K_bits(x,` i`)`. In turn, by lines 3 and 4 of Algorithm 3, each `K_bits(x,` i`)` sets `ret_b`

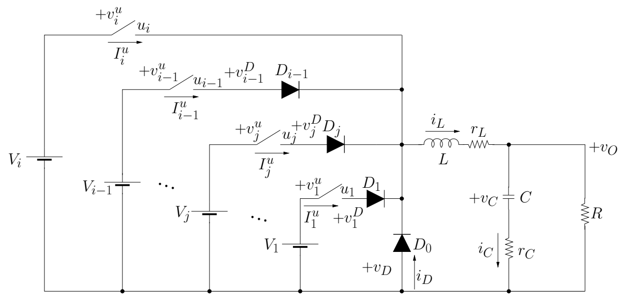

Figure 4. Multi-input Buck DC-DC converter.

to $\bar{b}_i$ and makes a jump to label `L_`v_i. By Lemma 3 and by construction of *Synthesize*, such $b_1, \ldots, b_r$ and $v_1, \ldots, v_r$ are such that that $[\![v_1, b_1]\!] = f_1(\boldsymbol{x}), \ldots, [\![v_r, b_r]\!] = f_r(\boldsymbol{x})$ and $K(\boldsymbol{x}, f_1(\boldsymbol{x}), \ldots, f_r(\boldsymbol{x})) = 1$. By Lemma 5, the sequence of calls `K_bits(x, 1)`, ..., `K_bits(x, `r`)` will indeed return, in at most $\sum_{j=1}^{r} O(\text{height}(v_i))$ steps, $f_1(\boldsymbol{x}), \ldots, f_r(\boldsymbol{x})$.

■

Corollary 7. *Let* $\rho = (\mathcal{V}, V, \mathbf{1}, \text{var}, \text{low}, \text{high}, \text{flip})$ *be a COBDD with* $\mathcal{V} = \mathcal{X} \cup \mathcal{U}$*,* $v \in V$ *be a node,* $b \in \mathbb{B}$ *be a boolean. Let* $[\![v, b]\!] = K(\boldsymbol{x}, \boldsymbol{u})$*,* $r = |\mathcal{U}|$ *and* $n = |\mathcal{X}|$*. Then the C function* `K` *output by function Synthesize*(ρ, v, b) *has WCET* $O(rn)$*.*

Proof: The corollary immediately follows from Theorem 6 and from the fact that, for all $v \in V$, $\text{height}(v) \leq n$.

■

VIII. Experimental Results

We implemented our synthesis algorithm in C programming language, using the CUDD (Colorado University Decision Diagram [19]) package for OBDD based computations and BLIF (Berkeley Logic Interchange Format [20]) files to represent input OBDDs. We name the resulting tool KSS (*Kontrol Software Synthesizer*). KSS is part of a more general tool named QKS (*Quantized feedback Kontrol Synthesizer* [6]).

A. Experimental Settings

We present experimental results obtained by using KSS on given COBDDs $\rho_1, \ldots, \rho_5$ such that for all $i \in [5]$ ρ_i represents the mgo $K_i(\boldsymbol{x}, \boldsymbol{u})$ for a *buck DC/DC converter with* i *inputs*.

The *multi-input* buck DC-DC converter [21] in Figure 4 is a mixed-mode analog circuit converting the DC input voltage (V_i in Figure 4) to a desired DC output voltage (v_O in Figure 4). As an example, buck DC-DC converters are used off-chip to scale down the typical laptop battery voltage (12-24) to the just few volts needed by the laptop processor (e.g., see [22]) as well as on-chip to support *Dynamic Voltage and Frequency Scaling* (DVFS) in multicore processors (e.g., see [23]). Because of its widespread use, control schemas for buck DC-DC converters have been widely studied. The typical software based approach (e.g., see [22]) is to control the switches $u_1, \ldots, u_i$ in Figure 4 (typically implemented with a MOSFET, i.e., a metal-oxide-semiconductor field-effect transistor [24]) with a microcontroller.

In the following experiments, we fix $n = |\boldsymbol{x}| = 20$ and we have that $r_i = |\boldsymbol{u}| = i$. Finally, K_i is an intermediate output of the QKS tool described in [6].

For each ρ_i, we run KSS so as to compute *Synthesize*(ρ_i, v_i, b_i) (see Algorithm 1), being $[\![v_i, b_i]\!] = K_i(\boldsymbol{x}, \boldsymbol{u})$. In the following, we will call $\langle v_{1i}, b_{1i}, \ldots, v_{ii}, b_{ii} \rangle$, with $v_{ji} \in V_i, b_{ji} \in \mathbb{B}$, the output of function *SolveFunctionalEq*(ρ_i, v_i, b_i). Moreover, we call $f_{1i}, \ldots, f_{ii} : \mathbb{B}^n \to \mathbb{B}$ the i boolean functions such that $[\![v_{ji}, b_{ji}]\!] = f_{ji}(\boldsymbol{x})$. All our experiments have been carried out on a 3.0 GHz Intel hyperthreaded Quad Core Linux PC with 8 GB of RAM.

B. KSS Performance

In this section, we will show the performance (in terms of computation time, memory, and output size) of the algorithms discussed in Section VI. Table I show our experimental results. The i-th row in Table I corresponds to experiments running KSS so as to compute *Synthesize*(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* shows the computation time of KSS (in secs). Column *MEM* shows the memory usage for KSS (in bytes). Column $|K|$ shows the number of nodes of the COBDD representation for $K_i(\boldsymbol{x}, \boldsymbol{u})$, i.e., $|V_{v_i}|$. Column $|F^{unsh}|$ shows the number of nodes of the COBDD representations of $f_{1i}, \ldots, f_{ii}$, without considering nodes sharing among such COBDDs. Note that we do consider nodes sharing inside each f_{ji} separately. That is, $|F^{unsh}| = \sum_{j=1}^{i} |V_{v_{ji}}|$ is the size of a trivial implementation of $f_{1i}, \ldots, f_{ii}$ in which each f_{ji} is implemented by a stand-alone C function. Column $|Sw|$ shows the size of the control software generated by KSS, i.e., the number of nodes of the COBDD representations $f_{1i}, \ldots, f_{ii}$, considering also nodes sharing among such COBDDs. That is, $|Sw| = |\cup_{j=1}^{i} V_{v_{ji}}|$ is the number of C code blocks generated by lines 6–8 of function *GenerateCCode* in Algorithm 3. Finally, Column *%* shows the gain percentage we obtain by considering nodes sharing among COBDD representations

Table I
KSS PERFORMACES

r	CPU	MEM	$\lvert K \rvert$	$\lvert F^{unsh} \rvert$	$\lvert Sw \rvert$	%
1	3.0e-02	1.0e+08	12137	2646	2646	0.0e+00
2	1.1e-01	1.3e+08	25848	5827	5076	1.3e+01
3	1.7e-01	1.8e+08	36430	10346	8606	1.7e+01
4	2.5e-01	2.4e+08	46551	15004	12285	1.8e+01
5	3.6e-01	3.3e+08	65835	21031	16768	2.0e+01

for $f_{1i}, \ldots, f_{ii}$, i.e., $(1 - \frac{|Sw|}{|F^{unsh}|})100$.

From Table I we can see that, in less than 1 second and within 350 MB of RAM we are able to synthesize the control software for the multi-input buck with $r = 5$ action variables, starting from a COBDD representation of K with about 6.6×10^4 nodes. The control software we synthesize in such a case has about 1.7×10^4 lines of code, whilst a control software not taking into account COBDD nodes sharing would have had about 2.1×10^4 lines of code. Thus, we obtain a 20% gain towards a trivial implementation.

IX. Conclusion and Future Work

In this paper, we presented an algorithm which, starting from a boolean relation K representing the set of implementations meeting the given system specifications, generates a correct-by-construction C code implementing K. This entails finding boolean functions F such that $K(x, F(x)) = 1$ holds, and then implement such F. WCET for the generated control software is linear linear in nr, being r the number of functions in F and $n = |x|$. Furthermore, we formally proved that our algorithm is correct.

We implemented our algorithm in a tool named KSS. Given our algorithm properties explained above, by using KSS it is possible to synthesize correct-by-construction control software, provided that K is provably correct with respect to initial formal specifications. This is the case in [6], thus this methodology, e.g., allows to synthesize correct-by-construction control software starting from formal specifications for DTLHSs. We have shown feasibility of our proposed approach by presenting experimental results on using it to synthesize C controllers for a multi-input buck DC-DC converter.

The WCET of the resulting control software may be too high for some systems in which nr is high, or for which the control software has to provide actions with an high frequency. In order to speed-up the WCET, a natural possible future research direction is to investigate how to parallelize the generated control software.

Acknowledgments

This work has received funding both from MIUR project TRAMP and the FP7/2007-2013 project ULISSE (grant agreement n^{o}218815).

References

[1] F. Mari, I. Melatti, I. Salvo, and E. Tronci, “From boolean relations to control software,” in *ICSEA 2011*, pp. 528–533.

[2] R. Bryant, “Graph-based algorithms for boolean function manipulation,” *IEEE Trans. on Computers*, vol. C-35, no. 8, 1986, pp. 677–691.

[3] D. Baneres, J. Cortadella, and M. Kishinevsky, “A recursive paradigm to solve boolean relations,” *IEEE Trans. on Computers*, vol. 58, no. 4, 2009, pp. 512–527.

[4] R. Wille and R. Drechsler, “Bdd-based synthesis of reversible logic for large functions,” in *DAC 2009*, pp. 270–275.

[5] E. Tronci, “Automatic synthesis of controllers from formal specifications,” in *ICFEM 1998*, pp. 134–143.

[6] F. Mari, I. Melatti, I. Salvo, and E. Tronci, “Synthesis of quantized feedback control software for discrete time linear hybrid systems,” in *CAV 2010*, ser. LNCS 6174, pp. 180–195.

[7] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. The MIT Press, 1999.

[8] F. Mari, I. Melatti, I. Salvo, and E. Tronci, “Quantized feedback control software synthesis from system level formal specifications for buck dc/dc converters,” *CoRR*, vol. abs/1105.5640, 2011.

[9] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli, E. Sentovich, and K. Suzuki, “Synthesis of software programs for embedded control applications,” *IEEE Trans. CAD*, vol. 18, 1995, pp. 834–849.

[10] T. Wongpiromsarn, U. Topcu, and R. M. Murray, “Formal synthesis of embedded control software: Application to vehicle management systems,” in *AIAA Infotech@Aerospace, 2011*.

[11] A. Cimatti, M. Roveri, and P. Traverso, “Strong planning in non-deterministic domains via model checking,” in *AIPS 1998*, pp. 36–43.

[12] V. Alimguzhin, F. Mari, I. Melatti, I. Salvo, and E. Tronci, “On model based synthesis of embedded control software,” in *EMSOFT 2012*, pp. 227–236.

[13] A. Pnueli and R. Rosner, “On the synthesis of an asynchronous reactive module,” in *ICALP 1989*, pp. 652–671.

[14] A. Girault and É. Rutten, “Automating the addition of fault tolerance with discrete controller synthesis,” *Formal Methods in System Design*, vol. 35, no. 2, 2009, pp. 190–225.

[15] ——, “From boolean functional equations to control software,” *CoRR*, vol. abs/1106.0468, 2011.

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[17] K. S. Brace, R. L. Rudell, and R. E. Bryant, “Efficient implementation of a bdd package,” in *DAC 1990*, pp. 40–45.

[18] S. Minato, N. Ishiura, and S. Yajima, “Shared binary decision diagram with attributed edges for efficient boolean function manipulation,” in *DAC 1990*, pp. 52–57.

[19] “CUDD Web Page,” http://vlsi.colorado.edu/ fabio/CUDD, last accessed 20th dec 2012

[20] “Berkeley logic interchange format (BLIF),“ bear.ces.cwru.edu/eecs_cad/sis_blif.pdf, last accessed 20th dec 2012.

[21] M. Rodriguez, P. Fernandez-Miaja, A. Rodriguez, and J. Sebastian, “A multiple-input digitally controlled buck converter for envelope tracking applications in radiofrequency power amplifiers,” *IEEE Trans. on Power Electronics*, vol. 25, no. 2, 2010, pp. 369–381.

[22] W.-C. So, C. Tse, and Y.-S. Lee, “Development of a fuzzy logic controller for dc/dc converters: design, computer simulation, and experimental evaluation,” *IEEE Trans. on Power Electronics*, vol. 11, no. 1, 1996, pp. 24–32.

[23] W. Kim, M. S. Gupta, G.-Y. Wei, and D. M. Brooks, “Enabling on-chip switching regulators for multi-core processors using current staggering,” in *ASGI 2007*.

[24] Y. Cheng and C. Hu, *MOSFET Modeling and Bsim3 User’s Guide*. Kluwer Academic Publishers, 1999.