From Boolean Relations to Control Software

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.**

***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. Such relation typically takes as input (the n-bits encoding of) a *state* x of the system 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 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* [1]) s.t. 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 [2][3], 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.

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 size of states encoding and $r = |u|$ the size of actions encoding. This allows us to synthesize correct-by-construction *control software*, provided that K is provably correct w.r.t. initial formal specifications. This is the case of [4], where an algorithm to synthesize K starting from the formal specification of a Discrete-Time Linear Hybrid System (*DTLHS* in the following) is presented. 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 $K(x, F(x)) = 1$ w.r.t. F is trivially decidable, since there are finitely many F. However, trying to explicitly enumerate all F requires time $\Omega(2^{r2^n})$ (being n the number of bits encoding state x and r the number of bits encoding state u). By using OBDD-based computations, our algorithm complexity is $O(r2^n)$ 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 [5]).

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

 ISBN: 978-1-61208-165-6

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, $F : \mathbb{B}^n \rightarrow \mathbb{B}^r$ 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, OBDDs nodes sharing will not be exploited. In our approach, we also exploit 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 70 MB of RAM we are able to synthesize the control software for a function K of 24 boolean variables, divided in $n = 20$ state variables and $r = 4$ action variables, represented by a OBDD with about 4×10^4 nodes. Such K represents the set of correct implementations for a real-world system, namely a multi-input buck DC/DC converter [6], obtained as described in [4]. The control software we synthesize in such a case has about 1.2×10^4 lines of code, whilest a control software not taking into account OBDDs nodes sharing would have had about 1.5×10^4 lines of code. Thus, we obtain a 24% 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. Finally, Section VII presents experimental results showing effectiveness of the proposed approach.

II. Related Work

Synthesis of boolean functions F satisfying a given boolean relation K in a way s.t. $K(x, F(x)) = 1$ is also addressed in [2]. However, [2] 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 [2] is not directly applicable here. An OBDD-based method for synthesis of boolean (reversible) functions is presented in [3] (see also citations thereof). Again, the method in [3] targets a hardware implementation, thus it is not applicable here.

In [4], an algorithm is presented which, starting from formal specifications of a DTLHS, synthesizes a correct-by-construction boolean relation K, and then a correct-by-construction control software implementation for K. However, in [4] the implementation of K is not described in detail. Furthermore, the implementation synthesis described in [4] has not the same size of the OBDD for F, i.e., it does not exploit OBDD node sharing.

In [7], an algorithm is presented which computes boolean functions F satisfying a given boolean relation K in a way s.t. $K(x, F(x)) = 1$. This approach is very similar to ours. However [7] does not generate the C code control software and it does not exploit OBDD node sharing.

Therefore, to the best of our knowledge this is the first time that an algorithm synthesizing correct-by-construction control software starting from a boolean relation (with the characteristics given in Section I) is presented.

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 $f : \mathbb{B}^n \rightarrow \mathbb{B}$ 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 $\{1, \ldots, n\}$.

1) 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 $K : S \times A \rightarrow \mathbb{B}$ 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 [8].

Efficient algorithms to compute mgos starting from suitable (nondeterministic) LTSs have been proposed in the literature (e.g., see [9]). Once an mgo 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.

2) 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

 ISBN: 978-1-61208-165-6

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* $K(\boldsymbol{x}, \boldsymbol{u}) = 1$ w.r.t. variables $\boldsymbol{u}$, that is we want to obtain boolean functions $f_1, \ldots, f_r$ s.t. $K(\boldsymbol{x}, f_1(\boldsymbol{x}), \ldots, f_r(\boldsymbol{x})) = K|_{u_1 = f_1(\boldsymbol{x}), \ldots, u_r = f_r(\boldsymbol{x})}(\boldsymbol{x}, \boldsymbol{u}) = 1$. 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 decide 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 [2], $f_1, \ldots, f_r$ 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.

V. OBDDs with Complemented Edges

In this section, we introduce OBDDs with complemented edges (COBDDs, Definition 1), which were first presented in [10][11]. Intuitively, they are OBDDs where else edges (i.e., edges of type $(v, low(v))$) may be complemented. Then edges (i.e., edges of type $(v, high(v))$) 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: i) $\mathcal{V} = \{x_1, \ldots, x_n\}$ is a finite set of *ordered* boolean variables; ii) V is a finite set of *nodes*; iii) $\mathbf{1} \in V$ is the *terminal* node of ρ, corresponding to the boolean constant 1 (non-terminal nodes are called *internal*); iv) for each internal node v, $\mathrm{var}(v) < \mathrm{var}(\mathrm{high}(v))$ and $\mathrm{var}(v) < \mathrm{var}(\mathrm{low}(v))$; v) $\mathrm{var}, \mathrm{low}, \mathrm{high}, \mathrm{flip}$ 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 *true 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).

COBDDs associated multigraphs: We associate to a COBDD $\rho = (\mathcal{V},\ V,\ \mathbf{1},\ \mathrm{var},\ \mathrm{low},\ \mathrm{high},\ \mathrm{flip})$ a labeled directed multigraph $G^{(\rho)} = (V, E)$ s.t. V is the same set of nodes of ρ and there is an edge $(v, w) \in E$ iff w is a child of v. Moreover, each edge $e \in E$ has a type $\mathrm{type}(e)$, indicating if e is a then, a regular else, or a complemented else edge. Figure 1 shows an example of a COBDD depicted via its associated multigraph, where edges are directed downwards. Moreover, in Figure 1 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 $\rho = (\mathcal{V},\ V,\ \mathbf{1},\ \mathrm{var},\ \mathrm{low},\ \mathrm{high},\ \mathrm{flip})$ 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 $\rho_v = (\mathcal{V},\ V_v,\ \mathbf{1},\ \mathrm{var},\ \mathrm{low},\ \mathrm{high},\ \mathrm{flip})$ s.t. $V_v = \{w \in V \mid \text{there exists a path from } v \text{ to } w \text{ in } G^{(\rho)}\}$ (note that $v \in V_v$).

Reduced COBDDs: Two COBDDs are *isomorphic* iff there exists a mapping from nodes to nodes preserving attributes var, flip, high and low. A COBDD is called *reduced* iff it contains no vertex v with $\mathrm{low}(v) = \mathrm{high}(v) \wedge \mathrm{flip}(v) = 0$, 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 $\mathrm{high}(v) = \mathrm{low}(v)$ for some $v \in V$, provided that $\mathrm{flip}(v) = 1$ (e.g., see nodes 0xf and 0xe in Figure 1). In the following, we assume all our COBDDs to be reduced.

COBDDs properties: For a given COBDD $\rho = (\mathcal{V},\ V,\ \mathbf{1},\ \mathrm{var},\ \mathrm{low},\ \mathrm{high},\ \mathrm{flip})$ the following properties follow from definitions given above: i) $G^{(\rho)}$ is a rooted directed acyclic (multi)graph (DAG); ii) each path in $G^{(\rho)}$ starting from an internal node ends in $\mathbf{1}$; iii) let $v_1, \ldots, v_k$ be a path in $G^{(\rho)}$, then $var(v_1) < \ldots < var(v_k)$.

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

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

Theor. 1 states that COBDDs are a *canonical* representation for boolean functions (see [10][11]).

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

VI. Synthesis of C Code from a COBDD

Let $K(x_1, \ldots, x_n, u_1, \ldots, u_r)$ be the mgo for a given control problem. Let $\rho = (\mathcal{V}, V, \mathbf{1}, \mathrm{var}, \mathrm{low}, \mathrm{high}, \mathrm{flip})$ be a COBDD s.t. there exist $v \in V$, $b \in \mathbb{B}$ s.t. $[\![v, b]\!] = K(x_1, \ldots, x_n, u_1, \ldots, u_r)$. Thus, $\mathcal{V} = \mathcal{X} \cup \mathcal{U} = \{x_1, \ldots, x_n\} \cup \{u_1, \ldots, u_r\}$ (we denote with $\cup$ 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*. More in-depth details may be found in [8].

A. Synthesis Algorithm: Overview

Our method *Synthesize* takes as input ρ, v and b s.t. $[\![v, b]\!] = K(\boldsymbol{x}, \boldsymbol{u})$. 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 mgo 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 $f_1, \ldots, f_r$ s.t. 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 node 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 Theor. 2.

Algorithm 1 Translating COBDDs to a C function

Require: COBDD ρ, node v, boolean b
Ensure: *Synthesize*(ρ, v, b):
1: $\langle v_1, b_1, \ldots, v_r, b_r \rangle \leftarrow$ *SolveFunctionalEq*(ρ, 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 $f_1, \ldots, f_r$ s.t. 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 [7]. 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 $\langle v_1, b_1, \ldots, v_r, b_r \rangle$ s.t. for all $i \in [r]$, $[\![v_i, b_i]\!] = f_i(\boldsymbol{x})$.

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. 2 (function *GenerateCCode*) and 3 (function *Translate*).

Given inputs ρ, v_1, b_1, $\ldots$, v_r, b_r (output by *SolveFunctionalEq*), Algs. 2 and 3 work 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, `K_bits(x,` i`)` traverses the graph $G^{(\rho_{v_i})}$ by taking, in each node v, the then edge if `x[`$j-1$`]` $= 1$ (with j s.t. $\mathrm{var}(v) = x_j$) and the else edge otherwise. When node $\mathbf{1}$ is reached, then 1 is returned iff the integer sum $c + b_i$ is even, being c the number of complemented else edges traversed. Parity of $c + b_i$ is maintained by initializing a C variable `ret_b` to $\bar{b}_i$, then complementing `ret_b` when a complemented else edge is traversed, and finally returning `ret_b`.

Thus, Algs. 2 and 3 generate `K_bits` in order to obtain the above described behavior. Namely, for all v_i output by the first phase (function *SolveFunctionalEq*), *GenerateCCode* calls *Translate* with parameters ρ, v_i, W, where W maintains the set of nodes already translated in C code. This results, for all such v_i, in a recursive graph traversal of $G^{(\rho_{v_i})}$ where, for each internal node $w \notin W$ which was not already translated, a C code block $B = B_1 B_2$ is generated s.t. B_1 is of the form `L_`w`: if (x[`$j-1$`]) goto L_`h`;`

(line 7 of Algorithm 3) and B_2 has one of the following forms: i) `else goto L_`l`;` (if $\text{flip}(w) = 0$, line 9 of Algorithm 3) or ii) `else {ret_b = !ret_b; goto L_`l`;}` (otherwise, line 8 of Algorithm 3). For the terminal node, the block `L_1: return ret_b;` is generated. Note that maintaining the set of already translated nodes W allows us to fully exploit COBDDs nodes sharing.

Algorithm 2 Generating C functions

Require: COBDD ρ, $v_1, \ldots, v_r$, 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 `", $i-1$, "`: ret_b = `", $\bar{b}_i$, "`; goto L_`", v_i,"`;`"
4: **print** "`}`" */* end of the switch block */*
5: $W \leftarrow \varnothing$
6: **for all do** $i \in [r]$ $W \leftarrow$*Translate*(ρ, v_i, W) **done**
7: **print** "`} K(int*x,int*u){int i;for(i=0;i<`",r,"`;i++)u[i]=K_bits(x,i);}`"

Algorithm 3 COBDD nodes translation

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

Algorithm Correctness: Correctness of our approach, i.e., of function *Synthesize* in Algorithm 1, is stated by Th. 2 (for the proof, see [8]).

Theorem 2. *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 $[\![v, b]\!] = K(\boldsymbol{x}, \boldsymbol{u})$. Then function Synthesize(ρ, v, b) generates a C function* `void K(int *x, int *u)` *with the following property: for all $\boldsymbol{x} \in \text{Dom}(K)$, 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 $O(nr)$.*

An Example of Translation: Consider the COBDD ρ shown in Figure 1. Within ρ, consider mgo $K(x_0, x_1, x_2, u_0, u_1) = [\![\text{0x17}, 1]\!]$. By applying *SolveFunctionalEq*, we obtain $f_1(x_0, x_1, x_2) = [\![\text{0x15}, 1]\!]$ and $f_2(x_0, x_1, x_2) = [\![\text{0x10}, 1]\!]$. Note that 0xe is shared between $G^{(\rho_{\text{0x15}})}$ and $G^{(\rho_{\text{0x10}})}$. Finally, by calling *GenerateCCode* (see Algorithm 2) on f_1, f_2, we have the C code in Figure 2.

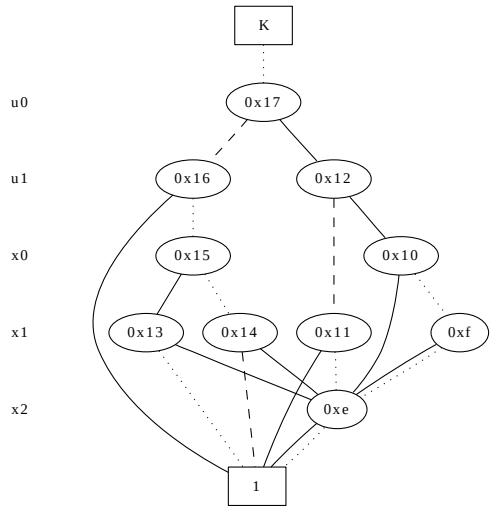

Figure 1. An mgo example

```
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 2. C code for the mgo in Figure 1 as generated by *Synthesize*

VII. Experimental Results

We implemented our synthesis 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 KSS (*Kontrol Software Synthesizer*). KSS is part of a more general tool named QKS (*Quantized feedback Kontrol Synthesizer* [4]).

1) Experimental Settings: We present experimental results obtained by using KSS on given COBDDs $\rho_1, \ldots, \rho_4$ 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 [6] for a description of this system), where $n = |\boldsymbol{x}| = 20$ and $r_i = |\boldsymbol{u}| = i$. K_i is an intermediate output of the QKS tool described in [4].

For each ρ_i, we run KSS so as to compute *Synthesize*(ρ_i, v_i, b_i) (see Algorithm 1). 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 out-

Table I
KSS PERFORMACES

r	CPU	MEM	$\|K\|$	$\|F^{unsh}\|$	$\|Sw\|$	%
1	2.2e-01	4.5e+07	12124	2545	2545	0.0e+00
2	4.2e-01	5.3e+07	25246	5444	4536	1.7e+01
3	5.2e-01	5.9e+07	34741	10731	8271	2.3e+01
4	6.3e-01	6.5e+07	43065	15165	11490	2.4e+01

put of function *SolveFunctionalEq*($\rho_i,\ v_i,\ b_i$). Moreover, we call $f_{1i}, \ldots, f_{ii} : \mathbb{B}^n \to \mathbb{B}$ the i boolean functions s.t. $[\![v_{ji}, b_{ji}]\!] = f_{ji}(\boldsymbol{x})$. All our experiments have been carried out on a 3.0 GHz Intel hyperthreaded Quad Core Linux PC with 8 GB of RAM.

2) 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., $|V_{v_{\ell_i}}|$. 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 5–6 of function *GenerateCCode* in Algorithm 2. Finally, Column % shows the gain percentage we obtain by considering node sharing among COBDD representations 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 70 MB of RAM we are able to synthesize the control software for the multi-input buck with $r = 4$ action variables, starting from a COBDD representation of K with about 4×10^4 nodes. The control software we synthesize in such a case has about 1.2×10^4 lines of code, whilest a control software not taking into account COBDD nodes sharing would have had about 1.5×10^4 lines of code. Thus, we obtain a 24% gain towards a trivial implementation.

VIII. Conclusion and Future Work

We presented an algorithm and a tool KSS implementing it 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 s.t. $K(x, F(x)) = 1$ 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 $n = |x|$. KSS allows us to synthesize correct-by-construction control software, provided that K is provably correct w.r.t. initial formal specifications. This is the case in [4], 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 buck DC-DC converter.

In order to speed-up the resulting WCET, a natural possible future research direction is to investigate how to parallelize the generated control software, as well as to improve don't-cares handling in F.

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

[2] D. Baneres, J. Cortadella, and M. Kishinevsky, "A recursive paradigm to solve boolean relations," *IEEE Trans. Comput.*, vol. 58, pp. 512–527, April 2009.

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

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

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

[6] 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.

[7] E. Tronci, "Automatic synthesis of controllers from formal specifications," in *ICFEM*. IEEE, 1998, pp. 134–143.

[8] F. Mari, I. Melatti, I. Salvo, and E. Tronci, "From boolean functional equations to control software," *CoRR*, vol. abs/1106.0468, 2011.

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

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

[11] 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.

 ISBN: 978-1-61208-165-6