From Boolean Functional Equations to Control Software

Federico Mari, Igor Melatti, Ivano Salvo, Enrico Tronci

Department of Computer Science
Sapienza University of Rome
via Salaria 113, 00198 Rome

e-mail: {mari,melatti,salvo,tronci}@di.uniroma1.it

January 27, 2013

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 WCET (Worst Case Execution Time) at most $O(nr)$, being $n = |x|$ the number of input arguments for functions in $F$ and $r$ the number of functions in $F$. 
1 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 [4]) 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], 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.

1.1 Our Contribution

In this paper we present an algorithm that, from an OBDD representation for $K$, effectively generates a C code implementation for $K$. 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 [10];
2. we generate a C code implementation for $F$ that has the same size as the OBDD for $F$ and a $O(nr)$ WCET, being $n = |x|$ the size of states encoding and $r = |u|$ the size of actions encoding. Indeed, we prove a more strict upper bound for the WCET by also considering the heights of the OBDDs representing $F$.

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 w.r.t. initial formal specifications. This is the case of [7], 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 [6]).

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 only requires $O(n + r)$ bytes of RAM for the data. As for the program size, it is linear in the size (i.e., number of nodes) of the OBDDs representing $F$, thus again we rely on the compression OBDDs achieve in many interesting cases.

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 $4 \times 10^4$ 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 [7]. The control software we synthesize in such a case has about $1.2 \times 10^4$ lines of code, whilst a control software not taking into account OBDDs nodes sharing would have had about $1.5 \times 10^4$ lines of code. Thus, we obtain a 24% gain towards a trivial implementation.

1.2 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 above), the method in [2] is not directly applicable here.

In [7] 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 [7] the implementation of $K$ is neither described in detail, nor it is proved to be correct. Furthermore, the implementation synthesis described in [7] has not the same size of the OBDD for $F$, i.e. it does not exploit OBDD node sharing.

In [10] 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 [10] does not generate the C code control software and it does not exploit OBDD node sharing. Furthermore, the algorithm is not proved to be correct.

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 Sect. 1.1) is presented and proved to be correct.
2 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 $xy = x \cdot y$), $\neg$ (logical complementation) and $\oplus$ (logical XOR). We also denote with $f|_{x_i=g}(x_1, \ldots, x_n)$ the boolean function $f(x_1, \ldots, x_{i-1}, g(x_1, \ldots, x_n), x_{i+1}, \ldots, x_n)$ and with $\exists x_i f(x_1, \ldots, x_n)$ the boolean function $f|_{x_i=0}(x_1, \ldots, x_n) + f|_{x_i=1}(x_1, \ldots, x_n)$. We will also denote vectors of boolean variables in boldface, e.g. $\mathbf{x} = (x_1, \ldots, x_n)$.

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

2.1 Feedback Control Problem for Labeled Transition Systems

In this paper we focus on solving and implementing a functional equation $K(\mathbf{x}, \mathbf{u}) = 1$. In this section we show a typical case in which such an equation needs to be solved and implemented.

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 $T : S \times A \times S \to \mathbb{B}$ is the transition relation of $\mathcal{S}$. An LTS is deterministic if $T(s, a, s') \land T(s, a, s'') \Rightarrow s' = s''$, and nondeterministic otherwise. A run or path for an LTS $\mathcal{S}$ is a sequence $\pi = s_0, a_0, s_1, a_1, s_2, a_2, \ldots$ of states $s_t$ and actions $a_t$ such that $\forall t \geq 0 T(s_t, a_t, s_{t+1})$. The length $|\pi|$ of a finite run $\pi$ is the number of actions in $\pi$. We denote with $\pi^{(s)}(t)$ the $t$-th state element of $\pi$.

A controller for an LTS $\mathcal{S}$ is a function $K : S \times A \to \mathbb{B}$ such that $\forall s \in S, \forall a \in A$, if $K(s, a) = 1$ then $\exists s' \in S T(s, a, s') = 1$. We denote with $\text{Dom}(K)$ the set of states for which a control action is defined. Formally, $\text{Dom}(K) = \{ s \in S \mid \exists a K(s, a) \}$. $\mathcal{S}^{(K)}$ denotes the closed loop system, that is the LTS $(S, A, T^{(K)})$, where $T^{(K)}(s, a, s') = T(s, a, s') \land K(s, a)$.

In the following, by assuming proper boolean encoding functions for states and actions (as it is usually done in Model Checking applications, see [6]), we may see a controller as a boolean function $K : \mathbb{B}^n \times \mathbb{B}^r \to \mathbb{B}$, with $n = \lceil \log_2 |S| \rceil$ and $r = \lceil \log_2 |A| \rceil$.

We call a path $\pi$ fullpath if either it is infinite or its last state $\pi^{(s)}(|\pi|)$ has no successors (i.e. $\text{Adm}(\mathcal{S}, \pi^{(s)}(|\pi|)) = \emptyset$). We denote with $\text{Path}(s)$ the set of fullpaths starting in state $s$, i.e. the set of fullpaths $\pi$ such that $\pi^{(s)}(0) = s$. 

5
Given a path \( \pi \) in \( \mathcal{S} \), we define the measure \( J(\mathcal{S}, G, \pi) \) on paths as the distance of \( \pi^{(S)}(0) \) to the goal on \( \pi \). That is, if there exists \( n > 0 \) s.t. \( \pi^{(S)}(n) \in G \), then \( J(\mathcal{S}, \pi, G) = \min\{n \mid n > 0 \land \pi^{(S)}(n) \in G\} \). Otherwise, \( J(\mathcal{S}, \pi, G) = +\infty \). We require \( n > 0 \) since our systems are nonterminating and each controllable state (including a goal state) must have a path of positive length to a goal state. The worst case distance (pessimistic view) of a state \( s \) from the goal region \( G \) is \( J_{\text{strong}}(\mathcal{S}, G, s) = \sup\{J(\mathcal{S}, G, s, \pi) \mid \pi \in \text{Path}(s)\} \).

**Definition 2.1.** Let \( \mathcal{P} = (\mathcal{S}, I, G) \) be a control problem and \( K \) be a controller for \( \mathcal{S} \) such that \( I \subseteq \text{Dom}(K) \).

\( K \) is a strong solution to \( \mathcal{P} \) if for all \( s \in \text{Dom}(K) \), \( J_{\text{strong}}(\mathcal{S}^{(K)}, G, s) \) is finite.

An optimal strong solution to \( \mathcal{P} \) is a strong solution \( K^* \) to \( \mathcal{P} \) such that for all strong solutions \( K \) to \( \mathcal{P} \), for all \( s \in S \) we have: \( J_{\text{strong}}(\mathcal{S}^{(K^*)}, G, s) \leq J_{\text{strong}}(\mathcal{S}^{(K)}, G, s) \).

Intuitively, a strong solution takes a pessimistic view and requires that for each initial state, all runs in the closed loop system reach the goal (no matter nondeterminism outcomes). Unless otherwise stated, we call just solution a strong solution.

**Definition 2.2.** The most general optimal (mgo) strong solution (simply mgo in the following) to \( \mathcal{P} \) is an optimal strong solution \( \bar{K} \) to \( \mathcal{P} \) such that for all other optimal strong solutions \( K \) to \( \mathcal{P} \), for all \( s \in S \), for all \( a \in A \) we have that \( K(s, a) \Rightarrow \bar{K}(s, a) \).

Efficient algorithms to compute mgos starting from suitable (nondeterministic) LTSs have been proposed in the literature (e.g. see [5]). Once an mgo \( K \) has been computed, solving and implementing the functional equation \( \bar{K}(x, u) = 1 \) allows a correct-by-construction control software to be synthesized.

### 2.2 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 \( \text{var}(v) \) (internal node) or with a boolean constant \( \text{val}(v) \in \mathbb{B} \) (terminal node). Each \( R \) internal node \( v \) has exactly two children, labeled
with high($v$) and low($v$). Let $x_1, \ldots, x_n$ be the boolean variables labeling $R$ internal nodes. Each terminal node $v$ represents the (constant) boolean function $f_v(x_1, \ldots, x_n) = \text{val}(v)$. Each internal node $v$ represents the boolean function $f_v(x_1, \ldots, x_n) = x_i f_{\text{high}(v)}(x_1, \ldots, x_n) + \bar{x}_i f_{\text{low}(v)}(x_1, \ldots, x_n)$, being $x_i = \text{var}(v)$.

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. Two OBDDs are isomorphic iff there exists a mapping from nodes to nodes preserving attributes var, val, high and low.

An OBDD is called reduced iff it contains no vertex $v$ with low($v$) = high($v$), nor does it contain distinct vertices $v$ and $v'$ such that the subgraphs rooted by $v$ and $v'$ are isomorphic. This entails that isomorphic subgraphs are shared, i.e. only one copy of them is effectively stored (see [4]).

We will only deal with reduced OBDDs, thus we will call them simply OBDDs. It can be shown [4] that each boolean function can be represented by exactly one OBDD (up to isomorphism), thus OBDD representation for boolean functions is canonical.

3 Solving a Boolean Functional Equation

Let $K(x_1, \ldots, x_n, u_1, \ldots, u_r)$ be an mgo for a given control problem $\mathcal{P} = (\mathcal{S}, I, G)$. We want to solve the boolean functional equation $K(x, u) = 1$ w.r.t. variables $u$, that is we want to obtain boolean functions $f_1, \ldots, f_r$ s.t. $K(x, f_1(x), \ldots, f_r(x)) = K|_{u_1=f_1(x),\ldots,u_r=f_r(x)}(x, u) = 1$.

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 the plant. 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 Sect. [1] 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 so-
olution would be to set up, for a given state $x$, a SAT problem instance $C = C_{K1}, \ldots, C_{Kt}, c_1, \ldots, c_n$, where $C_{K1} \land \ldots \land C_{Kt}$ is equisatisfiable to $K$ and each clause $c_i$ is either $x_i$ (if $x_i$ is 1) or $\bar{x}_i$ (otherwise). Then $C$ may be solved using a SAT solver, and the values assigned to $u$ 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 at most proportional to $rn$.

4 OBDDs with Complemented Edges

In this section we introduce OBDDs with complemented edges (COBDDs, Def. 41), which were first presented in [3, 9]. Intuitively, they are OBDDs where else edges (i.e. edges of type $(v, \text{low}(v))$) may be complemented. Then edges (i.e. edges of type $(v, \text{high}(v))$) complementation is not allowed to retain canonicity. Edge complementation usually reduce resources usage, both in terms of CPU and memory.

**Definition 4.1.** An **OBDD with complemented edges** (COBDD in the following) is a tuple $\rho = (\mathcal{V}, V, 1, \text{var}, \text{low}, \text{high}, \text{flip})$ with the following properties:

1. $\mathcal{V} = \{x_1, \ldots, x_n\}$ is a finite set of boolean variables s.t. for all $x_i \neq x_j \in \mathcal{V}$, either $x_i < x_j$ or $x_j < x_i$;

2. $V$ is a finite set of nodes;

3. $1 \in V$ is the terminal node of $\rho$, corresponding to the boolean constant $1$; any non-terminal node $v \in V, v \neq 1$ is called internal;

4. var, low, high, flip are functions defined on internal nodes, namely:

- $\text{var} : V \setminus \{1\} \to \mathcal{V}$ assigns to each internal node a boolean variable in $\mathcal{V}$;
- $\text{high} : V \setminus \{1\} \to V$ assigns to each internal node $v$ a high child (or true child), representing the case in which $\text{var}(v) = 1$;
- $\text{low} : V \setminus \{1\} \to V$ assigns to each internal node $v$ a low child (or else child), representing the case in which $\text{var}(v) = 0$;
• \(\text{flip} : V \setminus \{1\} \to \mathbb{B}\) assigns to each internal node \(v\) a boolean value; namely, if \(\text{flip}(v) = 1\) then the else child has to be complemented, otherwise it is regular (i.e. non-complemented);

5. for each internal node \(v\), \(\text{var}(v) < \text{var}(\text{high}(v))\) and \(\text{var}(v) < \text{var}(\text{low}(v))\).

**COBDDs as (labeled) DAGs** A COBDD \(\rho = (V, V, 1, \text{var}, \text{low}, \text{high}, \text{flip})\) defines a labeled directed multigraph in a straightforward way. This is detailed in Def. 4.2.

**Definition 4.2.** Let \(\rho = (V, V, 1, \text{var}, \text{low}, \text{high}, \text{flip})\) be a COBDD. The *graph associated to \(\rho\)* is a labeled directed multigraph \(G(\rho) = (V, E)\) where \(V\) is the same set of nodes of \(\rho\) and:

1. \(E = \{(v, w) \mid w = \text{high}(v) \lor w = \text{low}(v)\}\) (\(E\) is a multiset since it may happen that \(\text{high}(v) = \text{low}(v)\) for some \(v \in V\));

2. the following labeling functions are defined on nodes and edges:

   - \(\text{ind} : V \setminus \{1\} \to V\) assigns to each internal node \(v\) a boolean variable in \(V\), and is defined by \(\text{ind}(v) = \text{var}(v)\);

   - \(\text{type} : E \to \{\text{then}, \text{else}, \text{compl}\}\) assigns to each edge \(e = (v, w)\) its type, and is defined by: \(\text{type}(e) = \text{then} (\text{then edge})\) iff \(\text{high}(v) = w\), \(\text{type}(e) = \text{else} (\text{regular else edge})\) iff \(\text{low}(v) = w \land \text{flip}(v) = 0\), \(\text{type}(e) = \text{compl} (\text{complemented else edge})\) iff \(\text{low}(v) = w \land \text{flip}(v) = 1\).

**Example 4.3.** Let \(\rho = (\{x_0, x_1, x_2\}, \{0x15, 0x14, 0x13, 0xe, 1\}, 1, \text{var}, \text{low}, \text{high}, \text{flip})\) be a COBDD with: i) \(\text{var}(0x15) = x_0, \text{var}(0x14) = \text{var}(0x13) = x_1, \text{var}(0xe) = x_2\) and \(x_0 < x_1 < x_2\); ii) \(\text{high}(0x15) = 0x13, \text{low}(0x15) = 0x14, \text{high}(0x13) = \text{high}(0x14) = 0xe, \text{high}(0xe) = \text{low}(0xe) = \text{low}(0x13) = \text{low}(0x14) = 1\); iii) \(\text{flip}(0x15) = 0, \text{flip}(0x14) = 0, \text{flip}(0x13) = \text{flip}(0x13) = \text{flip}(0xe) = 1\).

Then \(G(\rho)\) is shown in Fig. 2 where edges are directed downwards. Moreover, in Fig. 2 then edges are solid lines, regular else edges are dashed lines and complemented else edges are dotted lines.
Restriction of a COBDD  The graph associated to a given COBDD may be seen as a forest with multiple rooted multigraphs. Def. 4.4 allow us to select one root vertex and thus one rooted multigraph.

Definition 4.4. Let $\rho = (\mathcal{V}, V, 1, \text{var}, \text{low}, \text{high}, \text{flip})$ be a COBDD, and let $v \in V$. The COBDD restricted to $v$ is the COBDD $\rho_v = (\mathcal{V}, V_v, 1, \text{var}_v, \text{low}_v, \text{high}_v, \text{flip}_v)$ s.t.:

- $V_v = \{w \in V \mid$ there exists a path from $v$ to $w$ in $G^\rho\}$ (note that $v \in V_v$);
- $\text{var}_v$, $\text{low}_v$, $\text{high}_v$ and $\text{flip}_v$ are the restrictions to $V_v$ of $\text{var}$, $\text{low}$, $\text{high}$ and $\text{flip}$.

Reduced COBDDs  Two COBDDs are isomorphic iff there exists a mapping from nodes to nodes preserving attributes $\text{var}$, $\text{flip}$, $\text{high}$ and $\text{low}$. A COBDD is called reduced iff it contains no vertex $v$ with $\text{low}(v) = \text{high}(v)$ and $\text{flip}(v) = 0$, nor does it contains distinct vertices $v$ and $v'$ such that $\rho_v$ 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 Fig. 3). In the following, we assume all our COBDDs to be reduced.

COBDDs Properties  For a given COBDD $\rho = (\mathcal{V}, V, 1, \text{var}, \text{low}, \text{high}, \text{flip})$ the following properties follow from definitions 4.1 and 4.2: i) $G^\rho$ is a rooted directed acyclic (multi)graph (DAG); ii) each path in $G^\rho$ starting from an internal node ends in 1; iii) let $v_1, \ldots, v_k$ be a path in $G^\rho$, then $\text{var}(v_1) < \ldots < \text{var}(v_k)$. We define the height of a node $v$ in a COBDD $\rho$ (notation $\text{height}_\rho(v)$, or simply $\text{height}(v)$ if $\rho$ is understood) as the height of the DAG $G^\rho$, i.e. the length of the longest path from $v$ to 1 in $G^\rho$.

4.1 Semantics of a COBDD

In Def. 4.5 we define the semantics $[.\!]$ of each node $v \in V$ of a given COBDD $\rho = (\mathcal{V}, V, 1, \text{var}, \text{low}, \text{high}, \text{flip})$ as the boolean function represented by $v$, given the parity $b$ of complemented edges seen on the path from a root to $v$.

Definition 4.5. Let $\rho = (\mathcal{V}, V, 1, \text{var}, \text{low}, \text{high}, \text{flip})$ be a COBDD. The semantics of a node $v \in V$ w.r.t. a flipping bit $b$ is a boolean function defined as:
• $[1, b]_{\rho} := \overline{b}$ (base of the induction)

• $[v, b]_{\rho} := x_i[\text{high}(v), b]_{\rho} + \overline{x_i}[\text{low}(v), b \oplus \text{flip}(v)]_{\rho}$ for any internal node $v$ (recursive step), being $x_i = \text{var}(v)$.

When $\rho$ is understood, we will write $[\cdot]$ instead of $[\cdot]_{\rho}$.

Note that the semantics of a node of a COBDD $\rho$ 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 (Prop. 4.6) that such boolean functions are complementary.

**Fact 4.6.** Let $\rho = (\mathcal{V}, V, 1, \text{var}, \text{low}, \text{high}, \text{flip})$ be a COBDD, let $v \in V$ be a node and $b \in \mathbb{B}$ be a flipping bit. Then $[v, b] = (v, b)$.

**Proof.** The proof is by induction on $v$. As base of the induction, we have $[1, b] = \overline{b} = \overline{\overline{b}} = [1, b]$.

As induction step, let $v$ be an internal node, and suppose by induction that $[\text{high}(v), b] = [\text{high}(v), b]$ and $[\text{low}(v), b] = [\text{low}(v), b]$.

Then, since $AB + \overline{A}C = (A + B)(A + C)$, we have: $[v, b] = x_i[\text{high}(v), b] + \overline{x_i}[\text{low}(v), b \oplus \text{flip}(v)] = (\overline{x_i} + [\text{high}(v), b])(x_i + [\text{low}(v), b \oplus \text{flip}(v)]) = (\overline{x_i} + [\text{high}(v), b])(x_i + [\text{low}(v), b \oplus \text{flip}(v)]) = x_i[\text{high}(v), b] + \overline{x_i}[\text{low}(v), b \oplus \text{flip}(v)] = [v, b]$.

**Example 4.7.** Let $\rho = (\{x_0, x_1, x_2\}, \{0x15, 0x14, 0x13, 0xe, 1\}, 1, \text{var}, \text{low}, \text{high}, \text{flip})$ be the COBDD of Ex. 4.5. If we pick nodes 0xe and 0x14 we have $[0xe, b] = x_2[1, b] + \overline{x_2}[1, b + 1] = x_2b + \overline{x_2}b = x_2 \oplus b$ and $[0x14, b] = x_1[0xe, b] + \overline{x_1}[1, b + 0] = x_1x_2b + x_1\overline{x_2}b + \overline{x_1}b = x_2 \oplus x_1\overline{x_2}b + \overline{x_1}b$.

Moreover, if we pick node 0x14, then it represents the two following boolean functions: $[0x14, 0] = x_2 + \overline{x_1}$ and $[0x14, 1] = x_1\overline{x_2}$ (note that $[0x14, 0] = [0x14, 1]$).

Theor. 4.8 states that COBDDS are a canonical representation for boolean functions (see 3 [9]).

**Theorem 4.8.** Let $f : \mathbb{B}^n \to \mathbb{B}$ be a boolean function. Then there exist a COBDD $\rho = (\mathcal{V}, V, 1, \text{var}, \text{low}, \text{high}, \text{flip})$, a node $v \in V$ and a flipping bit $b \in \mathbb{B}$ s.t. $[v, b] = f(x)$. Moreover, let $\rho = (\mathcal{V}, V, 1, \text{var}, \text{low}, \text{high}, \text{flip})$ 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 $v_1 = v_2 \land b_1 = b_2$. 

11
Efficient (i.e., at most $O(|V| \log |V|)$) algorithms exist to compute standard logical operations on COBDDs. We will assume to have available the following functions (for instantiation and existential quantifier elimination):

- $\text{COBDD\_APP}$ s.t. $\langle v_{\text{APP}}, b_{\text{APP}} \rangle = \text{COBDD\_APP}(x_{i_1}, \ldots, x_{i_k}, v_1, b_1, \ldots, v_k, b_k, v, b)$ iff $[v_{\text{APP}}, b_{\text{APP}}] = [v, b]|_{x_{i_1}=\llbracket v_1, b_1 \rrbracket, \ldots, x_{i_k}=\llbracket v_k, b_k \rrbracket}$;

- $\text{COBDD\_EX}$ s.t. $\langle v_{\text{EX}}, b_{\text{EX}} \rangle = \text{COBDD\_EX}(x_{i_1}, \ldots, x_{i_k}, v, b)$ iff $[v_{\text{EX}}, b_{\text{EX}}] = \exists x_{i_1}, \ldots, x_{i_k} [v, b]$.

Note that the above defined functions may create new COBDD nodes. We assume that such functions also properly update $V$, var, low, high, flip inside COBDD $\rho$ (1 and $\mathcal{V}$ are not affected).

5 Automatic Synthesis of C Code from a COBDD

Let $K(x_1, \ldots, x_n, u_1, \ldots, u_r)$ be an mgo for a given control problem. Let $\rho = (\mathcal{V}, V, \mathbf{1}, \text{var, low, high, flip})$ 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 $\mathcal{X} \cap \mathcal{U} = \emptyset$). 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. 3 targeting a software implementation. We do this by using a COBDD representing all our boolean functions. This allows us to exploit COBDD node sharing. This results in an improvement for the method in [10], 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 [10]. Given that $K$ is an mgo, this results in an optimal control software for the starting LTS.

5.1 Synthesis Algorithm: Overview

Our method $\text{Synthesize}$ takes as input $\rho, v$ and $b$ s.t. $[v, b] = K(x, u)$. Then, it returns as output a C function $\text{void } K(\text{int } *x, \text{ int } *u)$ 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 \( x \in \text{Dom}(K) \), and after the call to \( K \), \( \forall i \ u[i - 1] = u_i \) holds, then \( K(x, u) = 1 \). Moreover, the WCET of function \( K \) is at most \( O(nr) \).

Note that our method \( \text{Synthesize} \) 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 \( \text{Synthesize} \) is organized in two phases:

1. starting from \( \rho \), \( v \) and \( b \) (thus from \( K(x, u) \)), 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 \( x \) and computes the \( i \)-th bit \( u_i \) of an output action bit vector \( u \), where \( K(x, u) = 1 \), provided that \( x \in \text{Dom}(K) \). This computation is carried out in function \( \text{SolveFunctionalEq} \);

2. \( f_1, \ldots, f_r \) are translated inside function \( \text{void } K(\text{int } *x, \text{int } *u) \). 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 \( \text{GenerateCCode} \).

Thus function \( \text{Synthesize} \) is organized as in Alg. 1. Correctness for function \( \text{Synthesize} \) is proved by Theor. 6.5.

\begin{algorithm}
\textbf{Algorithm 1} Translating COBDDs to a C function
\begin{algorithmic}
\Require COBDD \( \rho = (V, V, 1, \text{var}, \text{low}, \text{high}, \text{flip}) \), node \( v \in V \), boolean \( b \in \mathbb{B} \)
\Ensure \( \text{Synthesize}(\rho, v, b) \):
1: \( \langle v_1, b_1, \ldots, v_r, b_r \rangle \leftarrow \text{SolveFunctionalEq}(\rho, v, b) /\!* \text{first phase } */\)
2: \( \text{GenerateCCode}(\rho, v_1, b_1, \ldots, v_r, b_r) /\!* \text{second phase } */\)
\end{algorithmic}
\end{algorithm}

\subsection{Synthesis Algorithm: Solving Functional Equation (First Phase)}

In this phase, starting from \( \rho \), \( v \) and \( b \) (thus from \( [v, b] = K(x, u) \)), we compute the COBDD nodes \( v_1, \ldots, v_r \) and flipping bits \( b_1, \ldots, b_r \) having the following properties:
• for all \( i \in [r] \), \([v_i, b_i] = f_i(x)\) (thus each \( f_i : \mathbb{B}^n \to \mathbb{B} \) does not depend on \( u \));

• for all \( x \in \text{Dom}(K) \), \( K(x, f_1(x), \ldots, f_r(x)) = 1 \).

In a hardware synthesis setting, techniques to compute \( f_1, \ldots, f_r \) satisfying the above functional equation have been widely studied (e.g. see \cite{2}). In our software synthesis setting we follow an approach similar to the one presented in \cite{10} to compute such \( f_1, \ldots, f_r \). Namely, we observe that \( f_i \) may be computed using \( f_1, \ldots, f_{i-1} \), that is \( f_i(x) = \exists u_{i+1}, \ldots, u_n K(x, f_1(x), \ldots, f_{i-1}(x), 1, u_{i+1}, \ldots, u_n) \) (see Lemma \ref{lemma:correctness}). This allows us to compute COBDD nodes \( v_1, \ldots, v_r \) and flipping bits \( b_1, \ldots, b_r \) as it is shown in function \text{SolveFunctionalEq} of Alg. \ref{alg:solve}. Correctness for function \text{SolveFunctionalEq} is proved in Lemma \ref{lemma:correctness2}.

\begin{algorithm}
\begin{algorithmic}
  \Require COBDD \( \rho = (\mathcal{V}, \mathcal{V}, 1, \text{var}, \text{low}, \text{high}, \text{flip}) \), node \( v \in \mathcal{V} \), boolean \( b \in \mathbb{B} \)
  \Ensure \text{SolveFunctionalEq}(\rho, v, b)\):
  \ForAll {\( i \in [r] \)}
    \State \([v_i, b_i] \leftarrow \text{COBDD\_EX}(u_{i+1}, \ldots, u_n, \text{COBDD\_APP}(u_1, \ldots, u_i, v_1, b_1, \ldots, v_{i-1}, b_{i-1}, 1, 0, v, b))\)
  \EndFor
  \State \textbf{return} \( \langle v_1, b_1, \ldots, v_r, b_r \rangle \)
\end{algorithmic}
\end{algorithm}

5.3 Synthesis Algorithm: Generating C Code (Second Phase)

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:

• \texttt{void K(int \*x, int \*u)}, which is the required output function for our method \texttt{Synthesize};

• \texttt{int K\_bits(int \*x, int action)}, which is an auxiliary function called by \texttt{K}. A call to \texttt{K\_bits(x, i)} returns \( f_i(x) \), being \( x[j-1] = x_j \) for all \( j \in [n] \).

This phase is detailed in Algs. \ref{alg:generate} and \ref{alg:generate2}. 

14
**Algorithm 3 Generating C functions**

**Require:** COBDD $\rho = (V, \{v_i\}, 1, \text{var}, \text{low}, \text{high}, \text{flip})$, nodes $v_1, \ldots, v_r$, 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;
2: switch(action) {}"
3: for all $i \in [r]$ do
4: print “case ”, $i - 1”, “: ret\_b = ”, $\bar{b}_i$, “; goto L”, $v_i”, “;”
5: print “}” /* end of the switch block */
6: $W \leftarrow \emptyset$
7: for all $i \in [r]$ do
8: $W \leftarrow \text{Translate}(\rho, v_i, W)$
9: print “}” /* end of K\_bits */
10: print “void K(int *x, int *u) {
11: int i; for(i=0;i<”, $r$, “;i++)
12: u[i]=K\_bits(x,i);}”

**Details of Function GenerateCCode (Alg. 3)** Given inputs $\rho, v_1, b_1, \ldots, v_r, b_r$ (output by SolveFunctionalEq), Alg. 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 $f_i(x)$. In order to do this, the graph $G(\rho_{v_i})$ is traversed by taking, in each node $v$, the then edge if $x[j - 1] = 1$ (with $j$ s.t. var($v$) = $x_j$) and the else edge otherwise. When node 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. 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. Note that formally this is equivalent to compute the flipping bit $b$ s.t. $(1, \bar{b}) = \text{COBDD\_APP}(x_1, \ldots, x_n, 1, 1-x[0], \ldots, 1, 1-x[n-1], v_i, b_i)$, being $[v_i, b_i] = f_i(x)$.

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 7 and detailed in Alg. 4.

Thus, the initial part of function K\_bits consists of a switch block (generated in lines 1-4 of Alg. 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 5–7 of Alg. 3, by calling \( r \) times function \( \text{Translate} \) (see Alg. 4) with parameters \( v_1, \ldots, v_r \). Note that \( W \) maintains the already translated COBDD nodes. Since function \( \text{Translate} \) 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 line 9. Function \( K \) simply consists in a for loop filling each entry \( u[i] \) of the output array \( u \) with the boolean values returned by \( \text{K\_bits}(x, i) \). Correctness of function \( \text{GenerateCCode} \) is proved in Lemma 6.4.

### Algorithm 4 COBDD nodes translation

**Require:** COBDD \( \rho = (V, V_0, 1, \text{var}, \text{low}, \text{high}, \text{flip}) \), node \( v \), nodes set \( W \subseteq V \)

**Ensure:** \( \text{Translate}(\rho, v, W) \):

1. if \( v \in W \) then return \( W \)
2. \( W \leftarrow W \cup \{v\} \)
3. print “L\_v”, \( v \), “;”
4. if \( v = 1 \) then
5. \hspace{1em} print “return ret\_b;”
6. else
7. \hspace{1em} let \( i \) be s.t. \( \text{var}(v) = x_i \)
8. \hspace{2em} print “if (x[i], i - 1, “] == 1) goto L\_v”, \( \text{high}(v) \), “;”
9. if \( \text{flip}(v) \) then print “else \{ret\_b = !ret\_b;goto L\_v”, \( \text{low}(v) \), “;\}”
10. else print “else goto L\_v”, \( \text{low}(v) \), “;”
11. \( W \leftarrow \text{Translate}(\rho, \text{high}(v), W) \)
12. \( W \leftarrow \text{Translate}(\rho, \text{low}(v), W) \)
13. return \( W \)

**Details of Function \( \text{Translate} \) (Alg. 4)** Given inputs \( \rho, v, W \), Alg. 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 7–10. 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 \( i \) s.t. \( \text{var}(v) = x_i \). The code to take the then edge of \( v \) is also generated. Namely, it is sufficient to generate a \texttt{goto} statement to the C code block related to node \text{high}(v). In lines \( 9 \) and \( 10 \) 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. \( \text{flip}(v) \) holds (line \( 9 \)), it is necessary to complement \( \text{ret}_b \) and then perform a \texttt{goto} statement to the C code block related to node \text{low}(v) (line \( 9 \)). Otherwise, it is sufficient to generate a \texttt{goto} statement to the C code block related to node \text{low}(v) (line \( 10 \)).

Thus, the block generated for an internal node \( v \), for proper \( i, l \) and \( h \), has one of the following forms:

- \( \mathbb{L}_v \): \texttt{if (}x[\text{i−1}]\text{)} \texttt{goto \mathbb{L}_h; else goto \mathbb{L}_l;} \\
- \( \mathbb{L}_v \): \texttt{if (}x[\text{i−1}]\text{)} \texttt{goto \mathbb{L}_h; else \{} \texttt{ret}_b = !\text{ret}_b; \texttt{goto \mathbb{L}_l;} \texttt{\}}.

There are two base cases for the recursion of function \textit{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;

- \( v = 1 \) (line \( 4 \)), i.e. the terminal node \( 1 \) has been reached. In this case, the C code block to be generated is simply \( \mathbb{L}_1: \texttt{return ret}_b; \). Note that such a block will be generated only once.

In all other cases, function \textit{Translate} ends with the recursive calls on the then and else edges (lines \( 11 \) \( 12 \)). Note that the visited nodes set \( W \) passed to the second recursive call is the result of the first recursive call. Correctness of function \textit{Translate} is proved in Lemma \( 6.4 \).

### 5.4 An Example of Translation

In this section we show how a node \( v \) and a flipping bit \( b \) of a COBDD \( \rho \) with 3 state variables and 2 action variables is translated in \( k \) and \( k\text{-bits} \) C functions. This is done by applying Algs. \( 1 \) \( 2 \) \( 3 \) and \( 4 \).

Consider COBDD \( \rho = \{u_0, u_1, x_0, x_1, x_2\}, \{0x17, 0x16, 0x15, 0x14, 0x13, 0x12, 0x11, 0x10, 0xf, 0xe, 1\}, 1, \text{var, low, high, flip} \). The corresponding \( \mathbb{G}(\rho) \) is shown in Fig. \( 1 \). Within \( \rho \), consider \( \text{mgo} \ K(x_0, x_1, x_2, u_0, u_1) = [0x17, 1] = \)
By applying SolveFunctionalEq (see Alg. 2), we obtain
\[ f_1(x_0, x_1, x_2) = \bar{u}_0 \bar{u}_1 x_0 \bar{x}_1 x_2 + \bar{u}_0 u_1 x_0 x_1 x_2 + u_0 \bar{u}_1 \bar{x}_0 \bar{x}_1 x_2 + u_0 u_1 \bar{x}_0 x_1 x_2 + u_0 u_1 x_0 \bar{x}_2. \]

By applying SolveFunctionalEq (see Alg. 2), we obtain \( f_1(x_0, x_1, x_2) = [0x15, 1] = x_0 x_1 + x_0 x_1 x_2 + x_0 x_1 + x_0 x_1 x_2 \) and \( f_2(x_0, x_1, x_2) = [0x10, 1] = x_0 x_1 x_2 + x_0 x_1 x_2 + x_0 x_2 \). COBDDs for \( f_1 \) and \( f_2 \) are depicted in Figs. 2 and 3 respectively. Note that in this simple example no new nodes have been added w.r.t. the COBDD of Fig. 1 and that node 0xe is shared between \( G(\rho_0x15) \) and \( G(\rho_0x10) \). Finally, by calling GenerateCCode (see Alg. 3) on \( f_1, f_2 \), we have the C code in Fig. 4.

6 Translation Proof of Correctness

In this section we prove the correctness of our approach (Theor. 6.5). That is, we show that the function \( K \) we generate indeed implements the given mgo \( K \), thus resulting in a correct-by-construction control software.

We begin by stating four useful lemmata for our proof. Lemma 6.1 is useful to prove Lemma 6.2, i.e. to prove correctness of function SolveFunctionalEq.

**Lemma 6.1.** Let \( K : \mathbb{B}^n \times \mathbb{B}^r \rightarrow \mathbb{B} \) and let \( f_1, \ldots, f_r \) be s.t. \( f_i(x) = \exists u_{i+1}, \ldots, u_r \ K(x, f_1(x), \ldots, f_{i-1}(x), 1, u_{i+1}, \ldots, u_r) \) for all \( i \in [r] \). Then, \( x \in \text{Dom}(K) \Rightarrow K(x, f_1(x), \ldots, f_r(x)) = 1. \)
Lemma 6.2. Let $\rho = (V, V, 1, \text{var}, \text{low}, \text{high}, \text{flip})$ be a COBDD with
$V = X \cup U$, $v \in V$ be a node, $b \in \mathbb{B}$ be a flipping bit. Let $[v, b] = K(x, u)$ and $r = |U|$. Then function $\text{SolveFunctionalEq}(\rho, v, b)$ (see Alg. 4) outputs nodes $v_1, \ldots, v_r$ and boolean values $b_1, \ldots, b_r$ s.t. for all $i \in [r]$ $[v_i, b_i] = f_i(x)$ and $x \in \text{Dom}(K)$ implies $K(x, f_1(x), \ldots, f_r(x)) = 1$.

Proof. Correctness of functions $\text{COBDD_APP}$ and $\text{COBDD_EX}$ (and lemma hypotheses) implies that for all $i \in [r]$ $f_i(x) = \exists u_{i+1}, \ldots, u_r \ K(x, f_1(x), \ldots, f_{i-1}(x), 1, u_{i+1}, \ldots, u_r)$. By Lemma 6.1 we have the thesis.

Let $\text{Translate}_\text{dup}$ be a function that works as function $\text{Translate}$ of Alg. 4 but that does not take node sharing into account. Function $\text{Translate}_\text{dup}$ may be obtained from function $\text{Translate}$ by deleting line 1 (highlighted in Alg. 4) and by replacing calls to $\text{Translate}$ in lines 11 and 12 with recursive calls to $\text{Translate}_\text{dup}$ (with no changes on parameters). Lemma 6.3 states correctness of function $\text{Translate}_\text{dup}$. 

Lemma 6.3. Let $\rho = (V, V, \mathbf{1}, \text{var}, \text{low}, \text{high}, \text{flip})$ 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 $\text{Translate}_\text{dup}(\rho, v, W)$ generates a sequence of labeled C statements $B_1 \ldots B_k$ s.t. $k \geq |V_v|$ 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 $\text{ret}_b = b$, a return statement is invoked in at most $O(p)$ steps with $\text{ret}_b = [w, b] = f_{w,b}(x)$ and $p = \text{height}(w)$.

Proof. We prove this lemma by induction on $v$. Let $v = \mathbf{1}$, which implies $[v, \mathbf{1}] = \mathbf{1}$ and $V_v = \{\mathbf{1}\}$. We have that function $\text{Translate}_\text{dup}(\rho, v, W)$ generates a single block $B_1$ (thus $k = 1 = |V_1|$) s.t. $B_1 = \text{L-1: return } \text{ret}_b$; (lines 2,3 of Alg. 4). Since by hypothesis we have $\text{ret}_b = \mathbf{1}$, 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 $\text{var}(v) = x_i$ and let $f(x) = [v, b]$. Since $w \in V_v$ iff $w = v \lor w \in V_{\text{high}(v)} \lor w \in V_{\text{low}(v)}$, by induction hypothesis we only have to prove the thesis for $w = v$. We have that $f(x) = x_i[\text{high}(v), b] + \bar{x}_i[\text{low}(v), b \oplus \text{flip}(v)]$, i.e. $f(x) = x_i[\text{high}(v), b] + \bar{x}_i[\text{low}(v), b]$ if $\text{flip}(v) = 0$ and $f(x) = x_i[\text{high}(v), b] + \bar{x}_i[\text{low}(v), b]$ if $\text{flip}(v) = 1$. Since $f(x) = x_i f|_{x_i=1}(x) + \bar{x}_i f|_{x_i=0}(x)$, by Theor. 4.8 we have that $[\text{high}(v), b] = f|_{x_i=1}(x)$, and that $[\text{low}(v), b] = f|_{x_i=0}(x)$ if $\text{flip}(v) = 0$ and $[\text{low}(v), b] = f|_{x_i=0}(x)$ if $\text{flip}(v) = 1$. 

20
By lines 8 and 10 of Alg. 4, we have that function \( \text{Translate}_\text{dup}(\rho, v, W) \) generates blocks \( BB_{11} \ldots B_{i1}B_{21} \ldots B_{2i} \) s.t. \( B = L_v \): if \( x[i - 1] == 1 \) goto \( L_{\text{high}}(v) \); else \( B_E \) where \( B_E \) is either goto \( L_{\text{low}}(v) \); if \( \text{flip}(v) = 0 \) or \( \{ \text{ret}_b = \!\text{ret}_b; \text{goto} L_{\text{low}}(v) \} \) if \( \text{flip}(v) = 1 \), and \( B_{11} \ldots B_{i1} \ (B_{21} \ldots B_{2i}) \) are generated by the recursive call \( \text{Translate}_\text{dup}(\rho, \text{high}(v), W) \) in line 11 (\( \text{Translate}_\text{dup}(\rho, \text{low}(v), W) \) in line 12). By induction hypothesis and the above reasoning, if the execution starts at label \( L_{\text{high}}(v) \) and \( \text{ret}_b = \bar{b} \), then a return \( \text{ret}_b \); statement is invoked in at most \( O(p - 1) \) steps with \( \text{ret}_b = f|_{x_i=1}(x) \). As for the else case, we have that starting from \( L_{\text{low}}(v) \) with \( \text{ret}_b = \bar{b} \) (\( \text{ret}_b = \bar{b} \) if \( \text{flip}(v) = 0 \), with \( \text{flip}(v) = 1 \), then a return \( \text{ret}_b \); statement is invoked in at most \( O(p - 1) \) steps with \( \text{ret}_b = f|_{x_i=0}(x) \). By construction of block \( B \), starting from label \( L_v \), a return \( \text{ret}_b \); statement is invoked in at most \( O(p - 1 + 1) = O(p) \) steps with \( \text{ret}_b = x_if|_{x_i=1}(x) + \bar{x}_i f|_{x_i=0}(x) = f(x) \). Finally, note that by induction hypothesis \( h \geq |V_{\text{high}}(v)| \) and \( l \geq |V_{\text{low}}(v)| \), thus we have that \( k = 1 + h + l \geq 1 + |V_{\text{high}}(v)| + |V_{\text{low}}(v)| \geq |V_v| \).

Lemma 6.4 extends Lemma 6.3 by also considering node sharing, thus stating correctness of function \( \text{GenerateCCode} \) of Alg. 4 and function \( \text{Translate} \) of Alg. 4.

**Lemma 6.4.** Let \( \rho = (V, V, 1, \text{var}, \text{low}, \text{high}, \text{flip}) \) 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 2-7 of function \( \text{GenerateCCode}(\rho, v_1, b_1, \ldots, v_r, b_r) \) generate a sequence of labeled \( C \) statements \( B_1 \ldots B_k \) s.t. \( k = \big| \cup_{i=1}^r V_{v_i} \big| \) 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 \( \text{ret}_b = \bar{b} \), a return \( \text{ret}_b \); statement is invoked in at most \( O(p) \) steps with \( \text{ret}_b = [v, b] = f_{v,b}(x) \) and \( p = \text{height}(w) \).

**Proof.** We begin by proving that \( k = \big| \cup_{i=1}^r V_{v_i} \big| \). 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 \( \text{Translate} \) in lines 11-13 of Alg. 4 and by function \( \text{GenerateCCode} \) in lines 5-7 of Alg. 3. In fact, function \( \text{Translate} \), when called on parameters \( \rho, v, W \), returns a set \( W' \supseteq W \), and function \( \text{GenerateCCode} \) calls \( \text{Translate} \) 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 \texttt{return} \( W \); statement in line 1 of Alg. 4 is executed. As base of the induction, let \( m = 1 \) and let \( \rho, v, W \) be the parameters of the recursive call executing the first \texttt{return} \( W \); statement. Then, by construction of function \textit{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 6.3, which implies that the induction base is proved.

Suppose now that the thesis holds for the first \( m \) executions of the \texttt{return} \( W \); statement in line 1 of Alg. 4. Then, by construction of function \textit{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 s.t. the \( m \) recursive calls executing the \texttt{return} \( W \); statement have parameters \( \rho, v, W_i \) (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 \( \texttt{ret}_b = \bar{b} \), a \texttt{return} \( \texttt{ret}_b \); statement is invoked in at most \( O(p) \) steps with \( \texttt{ret}_b = f_{w_i, b}(x) \). By Lemma 6.3 and its proof, the same holds for all \( v \in V_v \) \( \setminus \{w_1, \ldots, w_m\} \), thus it holds for all \( v \in V_v \).

We are now ready to give our main correctness theorem for function \textit{Synthesize} of Alg. 1.

\textbf{Theorem 6.5.} Let \( \rho = (V, V, \mathbf{1}, \text{var, low, high, flip}) \) be a COBDD with \( V = \mathcal{X} \cup \mathcal{U}, v \in V \) be a node, \( b \in \mathbb{B} \) be a boolean. Let \( [[v, b]] = K(x, u), r = |\mathcal{U}| \) and \( n = |\mathcal{X}| \). Then function \textit{Synthesize}(\( \rho, v, b \)) generates a C function \texttt{void} \( K(int *x, int *u) \) with the following property: for all \( x \in \text{Dom}(K) \), 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 \( K(x, u) = 1 \).

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

\textit{Proof.} Let \( x \in \text{Dom}(K) \) (i.e. \( \exists u \ K(x, u) = 1 \)) and suppose that for all \( j \in [n] \ x[j - 1] = x_j \). By line 9 of Alg. 3 for all \( i \in [r] \), \( u[i - 1] \) will take the value returned by \( K_{\text{bits}}(x, i) \). In turn, by line 3 of Alg. 3 each \( K_{\text{bits}}(x, i) \) sets \texttt{ret}_b \) to \( \bar{b}_i \) and makes a jump to label \( L_{v_i} \). By Lemma 6.2 and by construction of \textit{Synthesize}, such \( b_1, \ldots, b_r \) and \( v_1, \ldots, v_r \) are s.t. that
\([v_1, b_1] = f_1(x), \ldots, [v_r, b_r] = f_r(x)\) and \(K(x, f_1(x), \ldots, f_r(x)) = 1\). By Lemma 6.4, the sequence of calls \(K\_bits(x, 1), \ldots, K\_bits(x, r)\) will indeed return, in at most \(\sum_{j=1}^{r} O(\text{height}(v_i))\) steps, \(f_1(x), \ldots, f_r(x)\).

**Corollary 6.6.** Let \(\rho = (V, V, 1, \text{var}, \text{low}, \text{high}, \text{flip})\) be a COBDD with \(V = X \cup U\), \(v \in V\) be a node, \(b \in \mathbb{B}\) be a boolean. Let \([v, b] = K(x, u)\), \(r = |U|\) and \(n = |X|\). Then the C function \(K\) output by function \(\text{Synthesize}(\rho, v, b)\) has WCET \(O(rn)\).

*Proof.* The corollary immediately follows from Theor. 6.5 and from the fact that, for all \(v \in V\), \(\text{height}(v) \leq n\).

\[\square\]

### 7 Experimental Results

We implemented our synthesis algorithm in C programming language, using the CUDD package for OBDD based computations. We name the resulting tool KSS (Kontrol Software Synthesizer). KSS is part of a more general tool named QKS (Quantized feedback Kontrol Synthesizer [7]). KSS takes as input a BLIF file which encodes the OBDD for an mgo \(K(x, u)\). Such BLIF file also contains information about how to distinguish from state variables \(x\) and action variables \(u\). Then KSS generates as output a C code file containing functions \(K\) and \(K\_bits\) as described in Sect. 5. In this section we present our experiments that aim at evaluating effectiveness of KSS.

#### 7.1 Experimental Settings

We present experimental results obtained by using KSS on given COBDDs \(\rho_1, \ldots, \rho_4\) s.t. for all \(i \in [4]\):

- \(\rho_i = (V_i, V_i, 1, \text{var}_i, \text{low}_i, \text{high}_i, \text{flip}_i)\), with \(V_i = X_i \cup U_i = \{x_1, \ldots, x_{20}\} \cup \{u_1, \ldots, u_i\}\); thus \(n_i = 20\) and \(r_i = i\) (note that \(V_i \subset V_j\) for \(j > i\));

- there exists \(v_i \in V_i, b_i \in \mathbb{B}\) s.t. \([v_i, b_i] = K_i(x, u)\), being \(K_i(x, u)\) the COBDD representation of the mgo for a buck DC/DC converter with \(i\) inputs (see [8] for a description of this system). \(K_i\) is an intermediate output of the QKS tool described in [7].

23
For each $\rho_i$, we run KSS so as to compute $\text{Synthesize}(\rho_i, v_i, b_i)$ (see Alg. 1). In the following, we will call $(w_{i1}, b_{i1}, \ldots, w_{ii}, b_{ii})$, with $w_{ji} \in V_i, b_{ji} \in \mathbb{B}$, the output of function $\text{SolveFunctionalEq}(\rho_i, v_i, b_i)$ of Alg. 2. Moreover, we call $f_{i1}, \ldots, f_{ii} : \mathbb{B}^n \to \mathbb{B}$ the $i$ boolean functions s.t. $[w_{ji}, b_{ji}] = f_{ji}(x)$. Note that, by Lemma 6.2, for all $x \in \text{Dom}(K)$, $K_i(x, f_{i1}(x), \ldots, f_{ii}(x)) = 1$.

All our experiments have been carried out on a 3.0 GHz Intel hyper-threaded Quad Core Linux PC with 8 GB of RAM.

### 7.2 KSS Performance

In this section we will show the performance (in terms of computation time, memory, and output size) of the algorithms discussed in Sect. 5. Tab. 1 shows our experimental results. The $i$-th row in Tab. 1 corresponds to experiments running KSS so as to compute $\text{Synthesize}(\rho_i, v_i, b_i)$. Columns in Tab. 1 have the following meaning. Column $r$ shows the number of action variables, i.e. $|U_i|$ (note that $|X_i| = 20$ for all $i \in [4]$). 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 $K_i(x, u)$, i.e. $|V_v|$. Column $|F^{unsh}|$ shows the number of nodes of the COBDD representations of $f_{i1}, \ldots, f_{ii}$, without considering nodes sharing among such COBDDs. Note that we do consider nodes sharing inside each $f_{ji}$ separately. That is, $|F^{unsh}| = \sum_{j=1}^{i} |V_{w_{ji}}|$ is the size of a trivial implementation of $f_{i1}, \ldots, f_{ii}$ 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 $f_{i1}, \ldots, f_{ii}$, considering also nodes sharing among such COBDDs. That is, $|Sw| = |\cup_{j=1}^{i} V_{w_{ji}}|$ is the number of C code blocks generated by lines 5-7 of function $\text{GenerateCCode}$ in Alg. 3. Finally, Column $\%$ shows the gain.

| $r$ | CPU | MEM | $|K|$ | $|F^{unsh}|$ | $|Sw|$ | $\%$ |
|-----|-----|-----|------|----------|--------|------|
| 1   | 2.20e-01 | 4.53e+07 | 12124 | 2545 | 2545 | 0.00e+00 |
| 2   | 4.20e-01 | 5.29e+07 | 25246 | 5444 | 4536 | 1.67e+01 |
| 3   | 5.20e-01 | 5.94e+07 | 34741 | 10731 | 8271 | 2.29e+01 |
| 4   | 6.30e-01 | 6.50e+07 | 43065 | 15165 | 11490 | 2.42e+01 |
percentage we obtain by considering node sharing among COBDD representations for \( f_{i_1}, \ldots, f_{i_2} \), i.e. \( 1 - \frac{|Sw|}{|Fun|} \) \times 100.

From Tab. 1 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 \( 4 \times 10^4 \) nodes. The control software we synthesize in such a case has about \( 1.2 \times 10^4 \) lines of code, whilst a control software not taking into account COBDD nodes sharing would have had about \( 1.5 \times 10^4 \) lines of code. Thus, we obtain a 24% gain towards a trivial implementation.

8 Conclusions

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 at most linear in \( nr \), being \( n = |x| \) the number of input arguments for functions in \( F \) and \( r \) the number of functions in \( F \). Furthermore, we formally proved that our algorithm is correct.

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 [7], 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 \( F \).

References

[1] Paul C. Attie, Anish Arora, and E. Allen Emerson. Synthesis of fault-tolerant concurrent programs. *ACM Trans. on Program. Lang. Syst.*, 26(1):125–185, 2004.
[2] David Baneres, Jordi Cortadella, and Mike Kishinevsky. A recursive paradigm to solve boolean relations. *IEEE Trans. Comput.*, 58:512–527, April 2009.

[3] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a bdd package. In *DAC*, pages 40–45, 1990.

[4] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, 1986.

[5] Alessandro Cimatti, Marco Roveri, and Paolo Traverso. Strong planning in non-deterministic domains via model checking. In *AIPS*, pages 36–43, 1998.

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

[7] Federico Mari, Igor Melatti, Ivano Salvo, and Enrico Tronci. Synthesis of quantized feedback control software for discrete time linear hybrid systems. In *CAV*, LNCS 6174, pages 180–195, 2010.

[8] Federico Mari, Igor Melatti, Ivano Salvo, and Enrico Tronci. Quantized feedback control software synthesis from system level formal specifications for buck dc/dc converters. Technical Report arXiv:1105.5640v1, arXiv, 2011.

[9] Shin-ichi Minato, Nagisa Ishiura, and Shuzo Yajima. Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In *DAC*, pages 52–57, 1990.

[10] Enrico Tronci. Automatic synthesis of controllers from formal specifications. In *ICFEM*, pages 134–143. IEEE, 1998.