Rolf Drechsler
Sebastian Huhn *Editors*

Advanced Boolean Techniques

Selected Papers from the 15th International Workshop on Boolean Problems

Springer

Advanced Boolean Techniques

Rolf Drechsler • Sebastian Huhn
Editors

Advanced Boolean Techniques

Selected Papers from the 15th International Workshop on Boolean Problems

Springer

Editors
Rolf Drechsler
University of Bremen/DFKI
Bremen, Bremen, Germany

Sebastian Huhn
University of Bremen/DFKI
Bremen, Germany

ISBN 978-3-031-28915-6 ISBN 978-3-031-28916-3 (eBook)
https://doi.org/10.1007/978-3-031-28916-3

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

For decades, Boolean functions have been significantly contributing to computer science and, by this, paved the way for the design and verification of state-of-the-art circuits and systems.

The *International Workshop on Boolean Problems* (IWSBP) is a bi-annually held and well-established forum to discuss the recent advances in problems related to Boolean logic and Boolean algebra. After two exhaustive years of the COVID-19 pandemic, the workshop took place again in person from September 22 to 23 in 2022 and was hosted at the University of Bremen, Germany. Such an in-person meeting undoubtedly offers an environment to spark engaging discussions and supports fruitful collaborations in the exciting field of Boolean problems.

The workshop addresses the scientific exchange of problems related to Boolean logic and Boolean algebra. It also includes problems of a discrete mathematical nature. The workshop provides a forum for researchers and engineers from different disciplines to exchange ideas and to discuss problems and solutions. The workshop is devoted to both theoretical discoveries and practical applications. One important aim is to initiate collaborative research and to find new areas of application.

The book's first chapter is a contribution resulting from the invited keynote at the workshop. Here, Martin Fränzle presents "Arithmetic Satisfiability-Modulo-Theory Solving Applied to Nonstandard Analysis Problems of Cyber-Physical Systems". The following nine chapters are extended manuscripts based on the workshop submissions. In the second chapter, Annika Heil and Oliver Keszocze write about the "Fast AIG-Based Approximate Logic Synthesis". Afterward, in the third chapter, Siang-Yun Lee, Heinz Riener, and Giovanni De Micheli consider "External Don't Cares in Logic Synthesis". Claudio Moraga, Radomir S. Stanković, and Milena Stanković target "Maiorana-McFarland Boolean Bent Functions Characterized by Their Reed-Muller Spectra" in the fourth chapter. In the fifth chapter, Muhammad Hassan, Thilo Vörtler, Karsten Einwich, Rolf Drechsler, and Daniel Große investigate "Toward System-Level Assertions for Heterogeneous Systems". In the sixth chapter, Marcel Merten, Mohammed E. Djeridane, Sebastian Huhn, and Rolf Drechsler write about "SAT-Based Key Determination Attack for Improving the Quality Assessment of Logic Locking Mechanisms". Anna Bernasconi,

Valentina Ciriani, and Licia Monfrini address “Autosymmetric and D-reducible Functions: Theory and Application to Security” in the seventh chapter. In the eighth chapter, Danila Gorodecky and Leonel Sousa focus on “Two-Operand Modular Multiplication to Small Bit Ranges”. Ahmad Al-zoubi and Goerschwin Fey present “Low-Latency Real-Time Inference for Multilayer Perceptrons on FPGAs” in the nineth chapter. In the tenth chapter, Bernd Steinbach and Christian Posthoff investigate “Thirty-Six Officers of Euler-New Insights Computed Using XBOOLE”. Finally, the book is concluded in the eleventh chapter by an invited article from Khushboo Qayyum, Alireza Mahzoon, and Rolf Drechsler about “Start Small But Dream Big: On Choosing a Static Variable Order for Multiplier BDDs”.

We would like to express our thanks to the program committee of the 15th IWSBP and to the organizational team, in particular, Lisa Jungmann and Kristiane Schmitt. Furthermore, we thank all the authors of contributed chapters who did a great job in submitting their manuscripts of very high quality. A special thanks goes to the keynote speakers of the workshop, Prof. Martin Fränzle (Carl von Ossietzky University of Oldenburg, Germany) and Dr. Stefan Frehse (formerly matched.io, Germany). Finally, we would like to thank Dhivya Savariraj, Hemalatha Velarasu, Zoe Kennedy, Brian Halm, and Charles Glaser from Springer. All this would not have been possible without their steady support.

Bremen, Germany — Rolf Drechsler
May, 2023 — Sebastian Huhn

Contents

Arithmetic Satisfiability-Modulo-Theory Solving Applied to Nonstandard Analysis Problems of Cyber-Physical Systems

Martin Fränzle

1 Introduction

Cyber-physical systems (CPSes) joining a physical environment and numerous embedded computational devices via digital networking into a tightly coupled system are rapidly becoming reality. They are at the heart of the recent push toward so-called smart environments, be it "smart cities" as denoting anticipated forms of CPS-enabled urban structures, or "smart grids," "smart transportation," and "smart health" advancing energy supplies, transportation systems, and medical technology, respectively, or "Industry 4.0" revolutionizing manufacturing technology. Most of these applications are inherently safety-critical in that malfunctions may endanger life, property, or the environment. The quest for ensuring the predictable, reliable, and safe operation of complex cyber-physical infrastructures thus becomes pronounced [26].

This quest does, however, induce the need to reason about the joint dynamic behavior of computational devices and physical (in a broad sense) processes. Such reasoning naturally involves hybrid discrete-continuous state, with the physical phenomena spanning a multidimensional continuous state space that is subject to continuous-time dynamics, while computational processes give rise to discrete state and behavior. Pertinent models reflecting the joint dynamics of such hybrid-state systems supporting their analysis are hybrid automata [1] and related formalisms,

Supported by Deutsche Forschungsgemeinschaft under grant no. FR 2715/4-1 and by the State of Lower Saxony within the Zukunftslabor Mobilität.

M. Fränzle (✉)
Department of Computing Science, Carl von Ossietzky University of Oldenburg, Oldenburg, Germany
e-mail: martin.fraenzle@uol.de

R. Drechsler, S. Huhn (eds.), *Advanced Boolean Techniques*,
https://doi.org/10.1007/978-3-031-28916-3_1

which induce complex verification conditions that can in general only be solved approximately due to undecidability of even the most basic dynamical problems of restricted subclasses of hybrid automata [19]. Such approximations have historically given rise to a vast and diverse set of different automated verification technologies [13].

All these analysis techniques do have to reason about infinite state due to the hybrid discrete-continuous nature of the state space. Adopting set-based reasoning is a must when it comes to automatic reasoning about such infinite domains, either directly or indirectly by constructing finite-state abstractions lumping together sets of infinite cardinality into a single abstract item. Various computational procedures have been devised for such set-based reasoning, with the most prominent ones being reach-set computation on the one hand and satisfiability-modulo-theory (SMT) solving on the other. Reach-set computation provides over- or—more rarely—under-approximations of the state set reachable within a bounded or unbounded time or step horizon within a computational representation of subsets of hybrid discrete-continuous state spaces of finite dimension. A specific such representation has to be chosen, as arbitrary sets are not representable due to cardinality reasons: only countable many sets are computer-representable, while the subsets of a finite-dimensional hybrid state space have the cardinality of the powerset of the continuum. Various computational representations and with them algorithms for the approximate computation of reach sets of hybrid automata have been devised, among them (computational) interval boxes, zonotopes and polyhedra, support functions, Taylor forms, or sub-level sets of algebraic functions (cf. [13] for an overview).

An alternative approach, which we employ on the examples in this note, is satisfiability-modulo-theory (SMT) solving [3] over arithmetic theories, as pioneered by the LP-SAT hybrid-state planner [28]. Being based on automated proof search over the underlying arithmetic theory and thereby implicitly defining and enumerating relevant subsets of the problem domain, it avoids explicit computation of reachable state sets while solving—mostly depth-bounded, though extensions to unbounded problems exist based on inductive proof rules or Craig interpolation [24]—reachability problems of infinite-state or hybrid-state systems [9]. While SMT was originally confined to decidable theories, where a theory solver deciding arbitrary conjunctions of theory atoms exists and the DPLL(T) or CDCL(T) paradigm, inheriting conflict-driven clause learning (CDCL) from propositional satisfiability [21], adds the ability to reason about arbitrary Boolean combinations of theory atoms, solvers like iSAT [16] and dReal [18] have later added the ability to (approximately yet safely) reason about undecidable fragments of arithmetic involving transcendental functions. Their reasoning is based on a tight integration of interval constraint propagation [5] with CDCL [21].

iSAT [15] and its subsequent extensions for direct coverage of nonlinear ordinary differential equations [10] and of stochastic SMT problems [16] thus exploit the Boolean algebra structure of the powerset—or rather of computationally representable subsets of the powerset—of the domains reasoned about, namely, the Booleans, the integers, the reals, the computational reals implemented as binary

floating-point numbers [25], and their combinations. Within this note, we cannot explain the algorithmic underpinnings of the iSAT solver family in any detail, but we explain their input languages and their use for encoding hybrid discrete-continuous state dynamics (Sect. 2), and we demonstrate recent applications outside the traditional domains of automated verification [17] and test automation [27]. With respect to those recent applications, we focus on the two sample applications of exact monitoring of cyber-physical processes under epistemic and aleatory uncertainty (Sect. 3.1, based on [12]) and of quantitative safety analysis of autonomous systems featuring brain-computer interfaces (Sec. 3.2, related to [8]).

2 iSAT and SiSAT

As we are aiming at the automated analysis of phenomena involving hybrid discrete-continuous state, we are faced with the general problem of mechanically solving equational and inequational arithmetic constraints over mixed discrete-continuous domains. These constraints do not only naturally involve linear and polynomial (often ambiguously denoted as "nonlinear" in the satisfiability-modulo-theory community) arithmetic but also transcendental functions and linear as well as nonlinear differential equations.

In order to understand how such constraint systems evolve from an analysis problem for a hybrid discrete-continuous state system modeled as a hybrid automaton [1], we take a look at the probabilistic hybrid automaton depicted in Fig. 1. In order to pursue *qualitative* bounded reachability analysis for hybrid-state systems via constraint solving [2, 14], where bounded reachability analysis constitutes the simplest instance of bounded model checking [7], with respect to a given set of goal or target states like, e.g., those satisfying $T > -12$, one generates the following constraints:

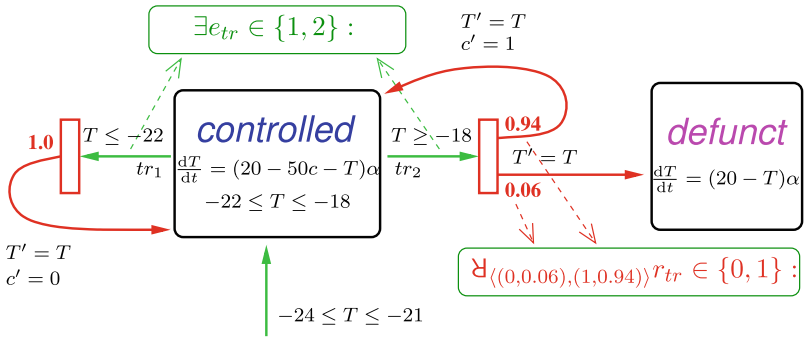


Fig. 1 A small probabilistic hybrid automaton, involving both nondeterministic transition selection (in green) and probabilistic transition selection (red), i.e., constituting a hybrid-state Markov decision process

Table 1 The symbolic transition relation for the PHA from Fig. 1 used in qualitative bounded model checking (BMC, [7])

Source	Guard	Action	Target
Controlled	$(T \leq -22)$	$(T' = T \wedge c' = 0)$	*Controlled'*
Controlled	$(T \geq -18)$	$(T' = T)$	*Defunct'*
Controlled	$(T \geq -18)$	$(T' = T \wedge c' = 1)$	*Controlled'*
Source	Flow	Invariant	Target
Controlled	$\left(\frac{\mathrm{d}T}{\mathrm{d}t} = (20 - 50c - T)\alpha\right)$	$(-22 \leq T \leq -18)$	*Controlled'*
Defunct	$\left(\frac{\mathrm{d}T}{\mathrm{d}t} = (20 - T)\alpha\right)$	True	*Defunct'*

- Initial conditions of the automaton map to corresponding constraints, yielding the initial-state condition $I \equiv \text{controlled} \wedge -24 \leq T \wedge T \leq -18$ for the automaton from Fig. 1.
- Transitions map to constraints between the values of state variables before and after the transition, the latter here denoted by decorating the state variable names with a prime. For the automaton from the example, the corresponding transition constraint is given in the upper part of Table 1.
- Durational stays in locations are interpreted as transitions from pre- to post-states connected by the differential equation associated with the location, as shown in the lower part of Table 1.

  Both parts of the transition relation together form the symbolic transition relation T
- The specification of the goal set simply is copied as $G \equiv T > -12$.

These constraints constitute a symbolic representation that is in one-to-one correspondence to qualitative, non-probabilistic behavior of the infinite-state transition system induced by the hybrid automaton.

"Unwinding" this symbolic transition to a desired depth $k \in \mathbb{N}$ is then achieved by k times copying the transition constraint under appropriate renaming, thereafter adding (accordingly renamed) versions of the initial state constraint and the target constraint. This yields a constraint $\Phi_k = I[\mathbf{x}/\mathbf{x}_0] \wedge T[\mathbf{x}/\mathbf{x}_0][\mathbf{x}'/\mathbf{x}_1] \wedge T[\mathbf{x}/\mathbf{x}_1][\mathbf{x}'/\mathbf{x}_2] \wedge \ldots \wedge T[\mathbf{x}/\mathbf{x}_{k-1}][\mathbf{x}'/\mathbf{x}_k] \wedge G[\mathbf{x}/\mathbf{x}_k]$, where $\mathbf{x}$ denotes the variable set of the above constraints and $\phi[\mathbf{x}/\mathbf{y}]$ the substitution of $\mathbf{x}$ by $\mathbf{y}$ in ϕ. Satisfiablity of the resulting constraint system Φ_k then corresponds to reachability of a state satisfying the target constraint within exactly k steps of the hybrid automaton [7], with any satisfying valuation of Φ_k constituting a path from an initial state to a target state of k steps in length.

Note that the constraint system Φ_k not only refers to a rich (and undecidable in general) fragment of arithmetic but also comprises arbitrary Boolean combinations of the arithmetic atoms involved. This necessity is induced by encoding transition systems and is in stark contrast to many settings of arithmetic constraint solving, e.g., interval constraint propagation [5], where conjunctive constraint systems tend to be addressed.

In order to address the solving of such hybrid-domain constraint systems involving complex Boolean connective structure, the author's group has together with Bernd Becker's group at Albert Ludwigs Universität Freiburg and Christoph Weidenbach's group at the Max-Planck-Institut für Informatik at Saarbrücken developed a number of arithmetic constraint solvers within the Transregional Research Center AVACS (Automatic Verification and Analysis of Complex Systems, DFG SFB-TRR 14, [4]). Most notably are iSAT [15, 24, 25] addressing the above fragment of linear, polynomial, and transcendental arithmetic constraints, yet without ordinary differential equations, iSAT(ODE) [10, 11] adding the respective support for ordinary differential equation constraints, and SiSAT [16, 17] adding stochastic quantification and a corresponding quantitative semantics akin to propositional stochastic SAT (SSAT, [20, 22]), yet lifting such to stochastic satisfiability modulo an arithmetic theory. Generalizing satisfiability-modulo-theory (SMT) based bounded model checking of hybrid automata [2, 14], such stochastic satisfiability-modulo-theory (SSMT) permits the direct analysis of probabilistic bounded reachability problems of probabilistic hybrid automata.

An SSMT formula $\phi = Q : \pi$ input to SiSAT hereby comprises a (possibly empty) quantifier prefix Q containing a sequence of existential ($\exists x \in \{v_1, \ldots, v_n\} : \ldots$) and random ($\text{⅁}x \in \{(v_1, p_1), \ldots, (v_n, p_n)\} : \ldots$ describing a probability distribution with finite carrier) quantifications over finite domains, followed by a quantifier-free "matrix" π, which is an arithmetic constraint formula as in iSAT or iSAT(ODE). In contrast to the Boolean satisfaction semantics of SMT formulae, the semantics of such an SSMT formula ϕ takes the form of a probability of satisfaction P_ϕ and is defined recursively over its quantifier prefix:

- If ϕ is quantifier-free, then $P_\phi = 1$ if ϕ is satisfiable and $P_\phi = 0$ else.
- If ϕ is of the form $\exists x \in \{v_1, \ldots, v_n\} : \psi$, where v_1 to v_n are theory constants, then $P_\phi = \max_{v \in \{v_1, \ldots, v_n\}} P_{\psi[x/v]}$.
- If ϕ is of the form $\text{⅁}x \in \{(v_1, p_1), \ldots, (v_n, p_n)\} : \psi$, where v_1 to v_n are (pairwise different) theory constants and d_1 to d_n rational numbers with $\sum_{i=1}^{n} p_i = 1$, then $P_\phi = \sum_{(v,p) \in \{(v_1,p_1),\ldots,(v_n,d_n)\}} p \cdot P_{\psi[x/v]}$.

Semantically, an SSMT formula thus constitutes a $1\frac{1}{2}$-player game, or equivalently a Markov decision process (MDP), where the existential player seeks to maximize the satisfaction probability of the matrix in response to the preceding random draws by the random player.

Using such SSMT constraint encodings, we can compute the maximum, w.r.t. optimal resolution of nondeterministic choices, probability of reaching a target state in a probabilistic hybrid automaton (cf. Fig. 1) as follows:

- I and G remain as before.
- The symbolic transition relation is enriched by recourse to variables e_{tr} and r_{tr} encoding the resolution of nondeterministic choices and probabilistic choices, respectively, as shown in Table 2.
- The unwinding Φ_k is extended by a quantifier prefix comprising existential and randomized quantification and thereby encoding the sequence

Table 2 The symbolic transition relation for the PHA from Fig. 1 used in quantitative bounded model checking (PBMC; [16])

Source	Guard	Trans	Distr	Action	Target
Controlled	$(T \leq -22)$	$(e_{tr} = 1)$	true	$(T' = T \wedge c' = 0)$	*Controlled'*
Controlled	$(T \geq -18)$	$(e_{tr} = 2)$	$(r_{tr} = 0)$	$(T' = T)$	*Defunct'*
Controlled	$(T \geq -18)$	$(e_{tr} = 2)$	$(r_{tr} = 1)$	$(T' = T \wedge c' = 1)$	*Controlled'*
Source	Flow		Invariant		Target
Controlled	$\left(\frac{\mathrm{d}T}{\mathrm{d}t} = (20 - 50c - T)\alpha\right)$		$(-22 \leq T \leq -18)$		*Controlled'*
Defunct	$\left(\frac{\mathrm{d}T}{\mathrm{d}t} = (20 - T)\alpha\right)$		True		*Defunct'*

of (and consequently the dependencies among) resolution of nondeterministic choices and probabilistic choices, rendering an SSMT formula $\Psi_k = \exists e_{tr\,0} \mathbin{\mathbb{U}} r_{tr\,0} \exists e_{tr\,1} \mathbin{\mathbb{U}} r_{tr\,1} \ldots \exists e_{tr\,(k-1)} \mathbin{\mathbb{U}} r_{tr\,(k-1)} : \Phi_k$.

The probability of satisfiability of the resulting constraint system Ψ_k then corresponds to the maximum probability of reaching a state satisfying the target constraint within exactly k steps of the probabilistic hybrid automaton.

3 Sample Applications

We will now demonstrate two recent sample applications of such arithmetic constraint solving, both of which go well beyond classical static analysis of design models of embedded or cyber-physical systems. The first deals with the stringent online monitoring of safety properties of cyber-physical systems when state observation is uncertain, the other with the quantitative safety analysis of critical cyber-physical systems basing their decisions on (highly uncertain) brain-computer interfaces.

3.1 Exact Monitoring of Cyber-Physical Systems Under Uncertainty[1]

Cyber-physical systems (CPS) joining a physical environment and numerous embedded computational devices via digital networking into a tightly coupled system are rapidly becoming reality. They are at the heart of the recent push toward so-called smart environments. Most of these applications are inherently safety-

[1] This section is based on joint work with Bernd Finkbeiner, Florian Kohn, and Paul Kröger published in [12].

critical in that malfunctions may endanger life, property, or the environment. The quest for ensuring the predictable, reliable, and safe operation of complex cyber-physical infrastructures is often addressed via stringent run-time monitoring. The applications pose high demands on the accuracy of the monitoring mechanisms, as lacking detection of an anomaly in system behavior may induce the aforementioned risks, while spurious signaling of a potential problem may lead to performance-degrading exception handling up to full system lockdown. Such accuracy, however, is hard to attain when observing physical state through actual sensor devices, thereby facing inevitable and significant inaccuracies and uncertainties in the state observation in the form of epistemic as well as aleatory uncertainties.

This provokes a quest for monitoring algorithms which are optimally exact given these inaccuracies and the partiality of the sensory equipment. *Exact* hereby means that they are both sound and maximally complete w.r.t. the monitoring problem under uncertainty. *Soundness* implies that the monitor will never provide monitoring verdicts that are artifacts of the uncertain observation, yet that all its verdicts invariantly hold true in any ground truth consistent with the observed noisy and partial measurements. If it provides a verdict whenever such verdict invariantly holds true in any measurement-consistent ground truth, then we call the monitor *complete*.

For qualitative models of sensory uncertainty, arithmetic SMT solving can provide such monitoring algorithms for spatiotemporal properties, as we demonstrate by means of an example. To this end assume that we model measurement error qualitatively (i.e., non-stochastical) as a nondeterministic measurement outcome, characterized by:

- an unknown yet fixed sensor offset that is bounded by a sensor-specific constant $\varepsilon > 0$,
- an independent per-sample error that is bounded by sensor-specific constant $\delta > 0$.

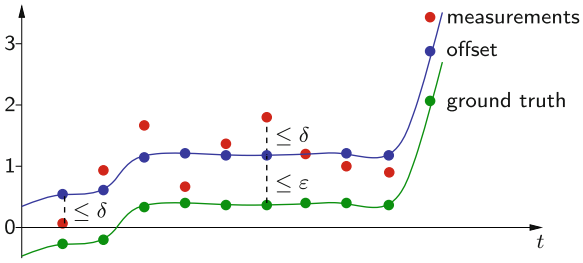
The upper bounds on these two values refer directly to the two terms trueness and precision used by the pertinent ISO norm 5725 to describe the accuracy of a measurement method.

Figure 2 explains the relationship between ground-truth values of physical entities and the related measurements under this model of measurement error. Given a ground-truth trajectory τ, where τ maps the various names s of physical signals to their actual signal $\tau(s) : \mathbb{R} \to \mathbb{R}$, a measurement time series m_s thus is possible iff:

$$\exists o \in [-\varepsilon, \varepsilon] : \forall t \in T : \exists e \in [-\delta, \delta] : \tau(s)(t) + o + e = m_s(t), \tag{1}$$

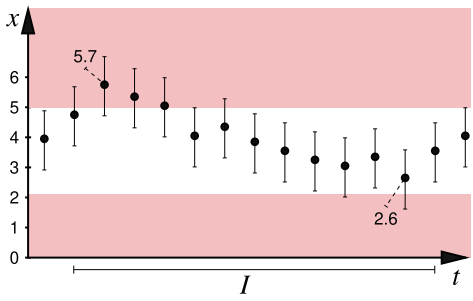
where T is the set of time instants where measurements are taken. Vice versa, ground truth τ is *consistent* with measurement series $m_{S_1}, \ldots m_{s_n}$, denoted by $m_{s_1}, \ldots, m_{s_n} \vdash \tau$, iff all m_{s_i} are possible w.r.t. τ. $\mathrm{GT}(m_{s_1}, \ldots, m_{s_n}) = \{\tau \mid m_{s_1}, \ldots, m_{s_n} \vdash \tau\}$ is the possible ground truth given $m_{S_1}, \ldots m_{s_n}$.

Now assume that our monitoring obligation is to at time $t = 13$ find out about the truth value of the signal temporal logic [23] formula $\mathrm{G}_{\leq 12}(x \geq 2 \wedge x \leq 5)$ at $t = 1$



- All measurements of the physical quantity are affected by the same sensor offset o with $|o| \leq \varepsilon$ (in blue),
- additionally each measurement is affected by a per-sample error e_i with $|e_i| \leq \delta$ (in red).

Fig. 2 Imprecise measurement (red dots) of a physical quantity (green line)



- x is the physical quantity observed,
- black dots denote inexact measurements m_i taken at time instances $i = 1 \ldots 15$,
- perpendicular intervals indicate error margins due to offset and per-sample error of ± 0.5,
- red areas indicate critical values for x satisfying $x \leq 2 \vee x \geq 5$.

Fig. 3 A time series of imprecise measurements (after [12])

given the time series of measurements depicted in Fig. 3. This time series has been obtained via a sensor observing physical state x with an offset error bound $\varepsilon = 0.5$ and per-sample error bound $\delta = 0.5$.

Given the relation between ground-truth values for x and the imprecise measurements m_i expressed in Eq. (1), we can reduce the question whether $\phi = \mathrm{G}_{\leq 12}(x \geq 2 \wedge x \leq 5)$ holds at time $t = 13$ to an arithmetic SMT problem as follows: We first rewrite Eq. (1) into an existential SMT problem by the following sequence of satisfiability-preserving transformation:

$$\exists o \in [-\varepsilon, \varepsilon] : \forall t \in T : \exists e \in [-\delta, \delta] : \tau(x)(t) + o + e = m_x(t) \tag{2}$$

$$\downarrow \text{ eliminate } \forall \text{ by specialisation to time domain referenced by } \phi$$

$$\exists o \in [-\varepsilon, \varepsilon] : \bigwedge_{t=1}^{13} \exists e \in [-\delta, \delta] : \tau(x)(t) + o + e = m_x(t) \tag{3}$$

$$\downarrow \text{ Skolemization, quantifier lifting, bound renaming}$$

$$\exists o \in [-\varepsilon, \varepsilon] : \exists e_1, \ldots, e_{13} \in [-\delta, \delta] : \bigwedge_{t=1}^{13} \tau(x)(t) + o + e_i = m_x(t) \tag{4}$$

$$\downarrow \text{ interpret as satisfiability problem, drop } \exists \text{ quantifier prefix}$$

$$\bigwedge_{t=1}^{13} \tau(x)(t) + o + e_i = m_x(t) \tag{5}$$

$$\downarrow \text{ FO reduction by renaming}$$

$$\bigwedge_{t=1}^{13} x_i + o + e_i = m_{x_i} \tag{6}$$

$$\downarrow \text{ add actual measurements}$$

$$\bigwedge_{t=1}^{13} x_i + o + e_i = m_{x_i} \wedge \underbrace{m_{x_0} = 3.95 \wedge m_{x_1} = 4.8 \wedge m_{x_2} = 5.7 \wedge \ldots \wedge m_{x_{12}} = 2.6 \wedge m_{x_{13}} = 3.66}_{\text{measurement values}} \tag{7}$$

As Eq. (7) expresses consistency between the measurements m_i and possible ground-truth values x_i, we then simply add the bounded model checking [7] tableaux for ϕ or alternatively for $\neg\phi$ at time $t = 1$ to Eq. (7), obtaining

$$\psi := \underbrace{(7)}_{\text{consistency}} \wedge \underbrace{\bigvee_{t=1}^{13} (x_i < 2 \vee x_i > 5)}_{\neg\phi \text{ holds at } t=1} \tag{8}$$

$$\psi' := \underbrace{(7)}_{\text{consistency}} \wedge \underbrace{\bigwedge_{t=1}^{13} (x_i \geq 2 \vee x_i \leq 5)}_{\phi \text{ holds at } t=1} \tag{9}$$

It now is easy to see that ψ is satisfiable iff there is a possible ground truth consistent with the actual measurements that violates ϕ, implying that ψ is unsatisfiable iff each ground truth consistent with the actual measurements is guaranteed to satisfy ϕ at time $t = 1$. Analogously, ψ' is satisfiable iff there is a possible ground truth consistent with the actual measurements that satisfies ϕ, implying that ψ is unsatisfiable iff each ground truth consistent with the actual measurements is guaranteed to violate ϕ at time $t = 1$.

Pursuing arithmetic SMT solving on both ψ and ψ' and reporting

- "ϕ holds at $t = 1$" iff ψ is unsatisfiable,
- "ϕ is violated at $t = 1$" iff ψ' is unsatisfiable,
- "the measurements are inconclusive about ϕ at $t = 1$" iff both ψ and ψ' are satisfiable

thus provides a sound (no wrong verdicts are given ever) and complete (verdicts are provided whenever possible) monitoring procedure based on SMT solving. The induced proof obligation of deciding satisfiability of the arithmetic constraint systems ψ and ψ' can be discharged by various SMT solvers covering linear arithmetic, like MathSAT [6]. When the exactness of results is to be further refined by also incorporating a model of the system dynamics into the state estimation (see [12] for details) and if this model is involves transcendental arithmetic, then solvers like iSAT [15] and dReal [18] become the method of choice.

3.2 *Quantitative Safety Analysis of BCI-Enabled Autonomous Systems*[2]

Our second example deals with the use of brain-computer interfaces to inform the embedded decision logic of a cyber-physical system about the imminent, potentially conflictory actions of a human. The scenario depicted in Fig. 4 entails the use of blood-oxygen-level-dependent imaging via a functional near-infrared spectroscopy (fNIRS) helmet to detect cortical activity, the use of a computer-vision system based on a convolutional neural network trained to spot signs of stress in the fNIRS image, a Bayesian network trained to predict human behavior dependent on stress levels (and some other factors like age and gender of the subject), and a robust control component basing driving decisions of an autonomous car on these observations. Though uncertainties as well as signal latencies are high (see the respective marking in Fig. 4), these together are nevertheless meant to provide the following safety-enhancing functionality:

- The signal processing and classification chain comprising of the fNIRS and the image classifier provide indications of whether the driver of the manually operated car (marked "M" in the image) while waiting for a left turn through oncoming traffic has built up stress due to extended waiting or due to preexisting conditions.
- The human behavior prediction determines the stress-dependent likelihood of the driver of car "M" filtering through the current gap in front of the automated red car (marked "A" in the image), as well the dependence of this likelihood on gap size variation.
- The control prompts car "A" to vary gap size to avoid risks should a stressed human driver in car "M" start to filter through oncoming traffic, while at the same time not compromising performance by always or unnecessarily frequently opening the gap in front of "A."

[2] This section is based on joint work with Werner Damm, Andreas Lüdtke, Jochem W. Rieger, Alexander Trende, and Anirudh Unni published in [8].

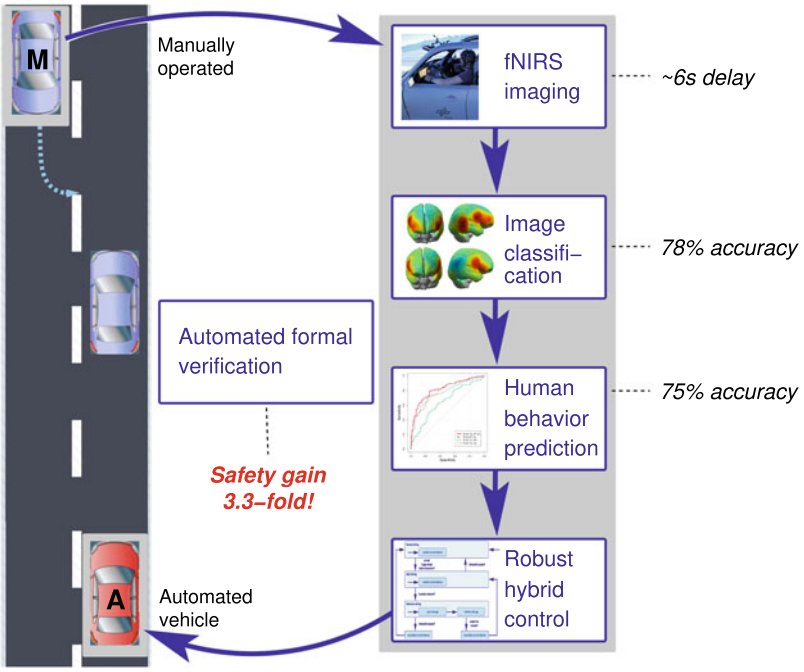


Fig. 4 Brain-computer interfaces supporting the decision-making in autonomous vehicles (after [8])

Full details of this scenario can be found in [8].

Given the latencies and inaccuracies in the fNIRS-based human state detection, the actual safety impact of such a system remains unclear. We have therefore encoded the system model into SiSAT, facilitating its rigorous analysis, as follows:

1. We have encoded a state-based model of the dynamic buildup of frustration in a waiting driver directly from the experimental findings obtained on human subjects in a driving simulator (and thus obviously needing further empirical justification for a transfer to reality, but being used herein cum grano salis as an example of the argument logic). This model tracks the buildup of frustration levels based on the number of gaps that the manual driver has waited for, with critical levels being reached from a non-frustrated state within 6 ± 2 gaps, but preexisting frustration being possible. The slope of stress buildup is nondeterministic in its start state and varies stochastically in speed within the aforementioned boundaries.
2. We have encoded probabilistic processes modeling the likelihood of misclassifications of stress detection due to an optimally adjusted detector, adjusting it conservatively, as required in a safety-oriented design, for a moderately high sensitivity even if that may come at the price of compromised specificity. We could thus calibrate the detection system for frustration to a sensitivity of 0.78

and obtained a similar reliability for the prediction of gap acceptance, which was encoded by an analogous probabilistic process. The stress detection process reads actual stress levels due to the dynamic model from 1. and reports an unreliable stress detection based on this reflecting the actual sensitivity; likewise the modeled actual gap acceptance varies stochastically around the predictions of the behavioral model.
3. We have built physical first principles models of the car movements.
4. We have for the sake of comparison once added the control component to the model of car "A" and once replaced it with a controller for keeping gap size in front of "A" constant.

Combining these models, we generated a corresponding symbolic SSMT representation of a hybrid-state Markov decision process (MDP) in SiSAT syntax using the coding scheme sketched in Sect. 2. Within this MDP, gap sizes in traffic on the HAV's lane as well as occurrence times of manually driven cars in opposing traffic were existential variables, forcing SiSAT to construct a worst-case (i.e., maximally risky) scenario, while gap acceptance, build-up slopes for frustration, and the frustration detection were random variables as described above. SiSAT was thus asked to construct a worst-case scenario of short and long gaps leading to maximum risk. The probabilities for the random variables were directly taken from the experimental findings obtained on male subjects, i.e., gap acceptance rates for short gaps in condition of frustration were 0.37 if the oncoming traffic was manually operated and 0.97 if it was an HAV; likewise, detection rate of frustration was 0.78.

In the uncontrolled setting of a constant gap size in front of "A," SiSAT based on this MDP computed the risk of traversing through a too short gap in front of car "A" as being in the interval [0.96999999, 0.97000001] for the worst possible scenario. With the robust control strategy in place, the worst-case risk of traversing through a too short gap was computed as [0.29584999, 0.29585001], implying a risk reduction for worst-case scenarios by a factor of approximately 3.3 despite the uncertainties in sensing frustration by neurophysiological measurements. It should be noted that these figures apply to the mutual worst-case scenarios (which need not even coincide) and are not reflecting the average case, which is dominated by non-risky situations and should not exhibit significant changes in behavior with associated impact on performance. It should also be clear that these figures are currently only meant to demonstrate feasibility of the analysis method and cannot directly be transferred to realistic driving situations due to lack of a sufficiently dense empirical basis of some model elements.

4 Conclusion

Satisfiability solving modulo theories (SMT) [3] has evolved into a stable and now rather scalable algorithmic foundation for the automated analysis of diverse computational as well as cyber-physical phenomena, provided that they are encodable

in (mostly existential) fragments of logic over appropriate theories. Within this note, we have demonstrated how SMT solving over rich arithmetic domains can automatically discharge proof obligations of hybrid-state cyber-physical systems that are induced by analysis problems outside the traditional domains of automated verification [17] and test automation [27]. The two examples provided were exact monitoring of cyber-physical processes under epistemic and aleatory uncertainty [12] and quantitative safety analysis of autonomous systems featuring brain-computer interfaces [8], shedding light on the breadth of potential application areas of arithmetic SMT solving.

Acknowledgments Numerous colleagues who have over more than a decade contributed to the work reported herein deserve sincere thanks, most notably Bernd Becker, Andreas Eggers, Christian Herde, Holger Hermanns, Stefan Kupferschmidt, Stefan ratschan, Karsten Scheibler, Tino Teige, and Christoph Weidenbach for contributing to iSAT and its descendants; Bernd Finkbeiner, Florian Kohn, and Paul Kröger for cooperating on monitoring under uncertainty; and Werner Damm, Andreas Lüdtke, Jochem Rieger, Alexander Trende, and Anirudh Unni for transdisciplinary collaboration on brain-computer-interfaces enabling cyber-physical systems to assess human state.

My thanks also go to the organizers of the 15th International Workshop on Boolean Problems, Rolf Drechsler and Sebastian Huhn, for inviting me as a keynote speaker, and to Alberto Lluch Lafluente and the Software Systems Engineering Section at the Technical University of Denmark for hosting me during a research semester and thus providing a pleasant environment for preparing this note.

References

1. Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.: Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In: Grossman, R.L., Nerode, A., Ravn, A.P., Rischel, H. (eds.) Hybrid Systems. Lecture Notes in Computer Science, vol. 736, pp. 209–229. Springer (1992). https://doi.org/10.1007/3-540-57318-6_30
2. Audemard, G., Bozzano, M., Cimatti, A., Sebastiani, R.: Verifying industrial hybrid systems with mathsat. Electron. Notes Theor. Comput. Sci. **119**(2), 17–32 (2005). https://doi.org/10.1016/j.entcs.2004.12.022
3. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability - Second Edition, Frontiers in Artificial Intelligence and Applications, vol. 336, pp. 1267–1329. IOS Press (2021). https://doi.org/10.3233/FAIA201017
4. Becker, B., Podelski, A., Damm, W., Fränzle, M., Olderog, E., Wilhelm, R.: SFB/TR 14 AVACS - automatic verification and analysis of complex systems (der sonderforschungsbereich/transregio 14 AVACS - automatische verifikation und analyse komplexer systeme). Inf. Technol. **49**(2), 118–126 (2007). https://doi.org/10.1524/itit.2007.49.2.118
5. Benhamou, F., Granvilliers, L.: Continuous and interval constraints. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming, Foundations of Artificial Intelligence, vol. 2, pp. 571–603. Elsevier (2006). https://doi.org/10.1016/S1574-6526(06)80020-9
6. Cimatti, A., Griggio, A., Schaafsma, B., Sebastiani, R.: The MathSAT5 SMT Solver. In: Piterman, N., Smolka, S. (eds.) 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16–24. Lecture Notes in Computer Science, vol. 7795. Springer (2013)

7. Clarke, E.M., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods Syst. Des. **19**(1), 7–34 (2001). https://doi.org/10.1023/A:1011276507260
8. Damm, W., Fränzle, M., Lüdtke, A., Rieger, J.W., Trende, A., Unni, A.: Integrating neurophysiological sensors and driver models for safe and performant automated vehicle control in mixed traffic. In: 2019 IEEE Intelligent Vehicles Symposium, IV 2019, Paris, France, June 9–12, 2019. pp. 82–89. IEEE (2019). https://doi.org/10.1109/IVS.2019.8814188
9. de Moura, L.M., Rueß, H., Sorea, M.: Lazy theorem proving for bounded model checking over infinite domains. In: Voronkov, A. (ed.) Automated Deduction - CADE-18, 18th International Conference on Automated Deduction, Copenhagen, Denmark, July 27–30, 2002, Proceedings. Lecture Notes in Computer Science, vol. 2392, pp. 438–455. Springer (2002). https://doi.org/10.1007/3-540-45620-1_35
10. Eggers, A., Fränzle, M., Herde, C.: SAT modulo ODE: A direct SAT approach to hybrid systems. In: Cha, S.D., Choi, J., Kim, M., Lee, I., Viswanathan, M. (eds.) Automated Technology for Verification and Analysis, 6th International Symposium, ATVA 2008, Seoul, Korea, October 20–23, 2008. Proceedings. Lecture Notes in Computer Science, vol. 5311, pp. 171–185. Springer (2008). https://doi.org/10.1007/978-3-540-88387-6_14
11. Eggers, A., Ramdani, N., Nedialkov, N.S., Fränzle, M.: Improving the SAT modulo ODE approach to hybrid systems analysis by combining different enclosure methods. Softw. Syst. Model. **14**(1), 121–148 (2015). https://doi.org/10.1007/s10270-012-0295-3
12. Finkbeiner, B., Fränzle, M., Kohn, F., Kröger, P.: A truly robust signal temporal logic: Monitoring safety properties of interacting cyber-physical systems under uncertain observation. Algorithms **15**(4), 126 (2022)
13. Fränzle, M., Chen, M., Kröger, P.: In memory of oded maler: automatic reachability analysis of hybrid-state automata. ACM SIGLOG News **6**(1), 19–39 (2019). https://doi.org/10.1145/3313909.3313913
14. Fränzle, M., Herde, C.: Efficient proof engines for bounded model checking of hybrid systems. Electron. Notes Theor. Comput. Sci. **133**, 119–137 (2005). https://doi.org/10.1016/j.entcs.2004.08.061
15. Fränzle, M., Herde, C., Teige, T., Ratschan, S., Schubert, T.: Efficient solving of large nonlinear arithmetic constraint systems with complex boolean structure. J. Satisf. Boolean Model. Comput. **1**(3-4), 209–236 (2007). https://doi.org/10.3233/sat190012
16. Fränzle, M., Hermanns, H., Teige, T.: Stochastic satisfiability modulo theory: A novel technique for the analysis of probabilistic hybrid systems. In: Egerstedt, M., Mishra, B. (eds.) Hybrid Systems: Computation and Control, 11th International Workshop, HSCC 2008, St. Louis, MO, USA, April 22–24, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4981, pp. 172–186. Springer (2008). https://doi.org/10.1007/978-3-540-78929-1_13
17. Fränzle, M., Teige, T., Eggers, A.: Engineering constraint solvers for automatic analysis of probabilistic hybrid automata. J. Log. Algebraic Methods Program. **79**(7), 436–466 (2010). https://doi.org/10.1016/j.jlap.2010.07.003
18. Gao, S., Kong, S., Clarke, E.M.: dreal: An SMT solver for nonlinear theories over the reals. In: Bonacina, M.P. (ed.) Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7898, pp. 208–214. Springer (2013). https://doi.org/10.1007/978-3-642-38574-2_14
19. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata? J. Comput. Syst. Sci. **57**(1), 94–124 (1998). https://doi.org/10.1006/jcss.1998.1581
20. Littman, M.L., Majercik, S.M., Pitassi, T.: Stochastic boolean satisfiability. J. Autom. Reason. **27**(3), 251–296 (2001). https://doi.org/10.1023/A:1017584715408
21. Mahajan, Y.S., Fu, Z., Malik, S.: Zchaff2004: An efficient SAT solver. In: Hoos, H.H., Mitchell, D.G. (eds.) Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10–13, 2004, Revised Selected Papers. Lecture Notes in Computer Science, vol. 3542, pp. 360–375. Springer (2004). https://doi.org/10.1007/11527695_27

22. Majercik, S.M.: Stochastic boolean satisfiability. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability - Second Edition, Frontiers in Artificial Intelligence and Applications, vol. 336, pp. 1331–1369. IOS Press (2021). https://doi.org/10.3233/FAIA201018
23. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Lakhnech, Y., Yovine, S. (eds.) Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004. Lecture Notes in Computer Science, vol. 3253, pp. 152–166. Springer (2004). https://doi.org/10.1007/978-3-540-30206-3_12
24. Scheibler, K., Kupferschmid, S., Becker, B.: Recent improvements in the SMT solver isat. In: Haubelt, C., Timmermann, D. (eds.) Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV), Warnemünde, Germany, March 12–14, 2013, pp. 231–241 (2013)
25. Scheibler, K., Neubauer, F., Mahdi, A., Fränzle, M., Teige, T., Bienmüller, T., Fehrer, D., Becker, B.: Accurate ICP-based floating-point reasoning. In: Piskac, R., Talupur, M. (eds.) 2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3–6, 2016, pp. 177–184. IEEE (2016). https://doi.org/10.1109/FMCAD.2016.7886677
26. Sztipanovits, J., Koutsoukos, X., Karsai, G., Sastry, S., Tomlin, C., Damm, W., Fränzle, M., Rieger, J., Pretschner, A., Köster, F.: Science of design for societal-scale cyber-physical systems: challenges and opportunities. Cyber Phys. Syst. **5**(3), 145–172 (2019). https://doi.org/10.1080/23335777.2019.1624619
27. Teige, T., Eggers, A., Scheibler, K., Stasch, M., Brockmeyer, U., Holberg, H.J., Bienmüller, T.: Two decades of formal methods in industrial products at BTC embedded systems. In: Huisman, M., Pasareanu, C.S., Zhan, N. (eds.) 24th International Symposium on Formal Methods, FM 2021. Lecture Notes in Computer Science, vol. 13047, pp. 725–729. Springer (2021). https://doi.org/10.1007/978-3-030-90870-6_40
28. Wolfman, S.A., Weld, D.S.: The LPSAT engine & its application to resource planning. In: 16th International Joint Conference on Artifical Intelligence - Volume 1. p. 310–316. IJCAI'99, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1999)

Fast AIG-Based Approximate Logic Synthesis

Annika Heil and Oliver Keszocze

1 Introduction

A major objective of the technical industry is to provide its customers with small and fast devices which are simultaneously energy-efficient. System designers focus on three major aspects while designing a system: area, latency, and power consumption.

In terms of area, it is widely known to researchers in the field of digital technology that the number of transistors, which are able to fit on an integrated circuit, has risen steadily since the early 1960s, thanks to technological advance (also known as *Moore's law*) [1]. This trend of minimization has been declining and is expected to end in 2025 [2].

The question is, how to address the three aspects when "simply" minimizing the transistors will not be possible in the foreseeable future any more. It turns out that many applications, especially in the domain of digital signal processing, do not require strictly correct computations [3]. This is due to the fact that the human perception itself is not perfect. In some other situations, it might even be the case that the customer is willing to accept incorrect results in favor of having a faster, smaller, or less energy-hungry system [4].

A design paradigm known as *approximate computing* [5, 6] exploits this. The basic idea is to trade off computational accuracy for gains in nonfunctional aspects such as reduced area, smaller latency, and power reduction.

In the literature, two main approaches to introduce approximations to the design in order to achieve gains on one or multiple of the aspects mentioned above are used (see, e.g., [7]): (a) physical changes to the design including voltage over-scaling or overclocking or (b) altering the functionality. We will pursue the latter approach

A. Heil · O. Keszocze (✉)
Friedrich-Alexander Universität Erlangen-Nürnberg (FAU), Nürnberg, Germany
e-mail: annika.heil@fau.de; oliver.keszoecze@fau.de

R. Drechsler, S. Huhn (eds.), *Advanced Boolean Techniques*,
https://doi.org/10.1007/978-3-031-28916-3_2

in this work. More precisely, we will present a novel and fast *approximate logic synthesis* (ALS) technique. Our optimization goal is the circuit area. We refer the reader to [8] for a general survey on ALS techniques.

In this work we propose an ALS method that (a) aims to minimize the area of an approximated circuit, (b) specifically targets arithmetic circuits, (c) operates on (X)AIG representations of Boolean functions, and (d) has a small execution time due to a fast method of evaluating the error introduced by the approximation.

2 Related Work

Approximate logic synthesis has been performed on many different representations of Boolean functions using very different means of approximation.

Initial work has been done by using the structural information of a given circuit, e.g., by cutting the carry chain of adders or multipliers [9].

On a higher level of abstraction, researchers extended programming languages [10] and hardware description languages [11] with constructs to automatically compile/synthesize approximated systems.

Preliminary work on approximations on graph structures has mostly been done on BDDs [12–14]. In this work, we use the AIG data structure. The works that are closest to the work presented in this manuscript are [15] where the authors find cuts within an AIG that are replaced by approximations. The introduced error is bound by a miter structure that is evaluated using SAT. While the authors in [14] work on BDDs, we employ their idea of exploiting properties of the approximation operation to speed up the error metric computation process. We further also use their algorithmic approach for AIG approximation.

3 Background

3.1 Notation and Conventions

In this paper, all functions will be of type $f : \mathbb{B}^n \to \mathbb{B}^m$. The m individual output functions are denoted as f_i. The interpretation of $f(x)$ as a natural number with the usual binary encoding is denoted by $\text{val}(f(x))$.

For a function f with $m = 1$, i.e., a Boolean function, its ON/OFF-set is denoted by ON / OFF(f), i.e.

$$\text{ON}(f) := \{x \mid f(x) = 1\} \text{ and } \text{OFF}(f) := \{x \mid f(x) = 0\}.$$

The size of a ON/OFF-set is denoted $\#\,\text{ON}(f)/\#\,\text{OFF}(f)$.

Given a Boolean function f, an approximated version of it is denoted by a hat, i.e., $\hat{f}$. Primary inputs are labeled $A, B, \ldots$ and primary outputs X, Y, Z. Within this manuscript, the approximations do not alter the number of input or output variables.

The *truth density* $\mathrm{td}(f)$ of a function f is the ratio between the size of the ON-set of f and the total number of inputs, i.e.

$$\mathrm{td}(f) = \frac{1}{2^n} \cdot \#\mathrm{ON}(f).$$

The name stems from the fact that the truth density gives information about the probability of f being 1, i.e., true.

3.2 (XOR-)AND-Inverter Graphs

To efficiently represent Boolean functions, many representations have been presented. This work focuses on AND-Inverter Graphs (AIGs) [16] and XOR-AND-Inverter Graphs [17]. These structures are directed acyclic graphs. In both representations, nodes without incoming edges represent primary inputs, and nodes without outgoing edges represent primary outputs. For AIGs, the internal nodes represent the logical AND operation, whereas in a XAIG, the nodes can represent either the logical AND or the logical XOR operation. In both types of graphs, edges might be negated. We denote the size, i.e., the number of nodes, of an (X)AIG G by $\#G$.

Example 1 Consider the addition of the two-bit numbers $(CA), (DB) \in \mathbb{B}^2$, i.e., $(ZYX) = (CA) + (DB)$. An AIG representing the adder is shown in Fig. 1a. The nodes are AND operations, while dashed edges indicate negations. The output X is computed as

$$X = \underbrace{\neg \underbrace{(\neg B \wedge \neg A)}_{\text{Node6}} \wedge \neg \underbrace{(A \wedge B)}_{\text{Node5}}}_{\text{Node7}}. \tag{1}$$

Figure 1b shows an XAIG representing the same functionality, i.e., a two-bit adder. The gray nodes are XOR nodes. Note that the computation of X in Eq. (1) is actually an XOR operation, i.e., $X = A \oplus B$. This is reflected by the XAIG in node 7 that completely represents the computation of X. This shows that XAIGs may save nodes compared to AIGs. The AIG used the three nodes 5, 6, and 7 to represent the computation of X (see Fig. 1a).

While there is no one-to-one correspondence between the number of nodes in an (X)AIG and the resulting circuit size, the rule of thumb "less nodes lead to smaller circuits" does often hold and is used within this work.

In this work, we expect the functionality to be optimized by ALS to be given as an AIG. Hence, instead of optimizing a circuit, represented by an AIG, directly for

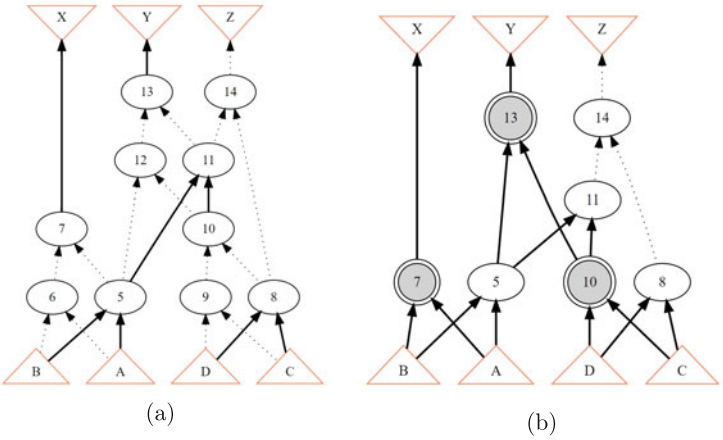


Fig. 1 AIG and XAIG for a two-bit adder. (**a**) AIG for a two-bit adder computing $(CA)+(DB) = (ZYX)$. Each node represents an AND operation. Dashed lines indicate negation. (**b**) XAIG representing the same functionality as the AIG in (**a**). The gray nodes are XOR nodes; the other nodes are AND nodes

the area used by an actual physical realization, we aim to minimize the number of AIG nodes instead.

3.3 Error Metrics

To evaluate systems in terms of the quality of the computed values, many different error metrics have been proposed. Each of these metrics measures different aspects of the approximated functionality (see [18] for an overview of commonly used metrics). Some examples of error metrics are

$$\mathrm{er}(f, \hat{f}) = \frac{1}{2^n} \cdot \sum_{x \in \mathbb{B}^n} f(x) \neq \hat{f}(x), \tag{2}$$

$$\mathrm{wce}(f, \hat{f}) = \max_{x \in \mathbb{B}^n} |\mathrm{val}(f(x)) - \mathrm{val}(\hat{f}(x))|, \text{ and} \tag{3}$$

$$\mathrm{whd}(f, \hat{f}) = \sum_{i=0}^{m-1} 2^i \sum_{x \in \mathbb{B}^n} \left(f_i(x) \oplus \hat{f}_i(x) \right). \tag{4}$$

The *error rate* (Eq. 2) counts how often the approximated function $\hat{f}$ computes an incorrect result. This metrics is not well-suited for evaluating approximations of arithmetic circuits as it does not take into account at all how severe the errors are as it completely ignores the actual function values. As this metrics is rather simple to evaluate (or compute an estimate using Monte Carlo simulations), it is often used

Table 1 Error metric values for the error rate, the worst-case error, and the weighted Hamming distance for an exemplary function f and its approximated function $\hat{f}$

x	$f(x)$	$\hat{f}(x)$	$f(x) \neq \hat{f}(x)$	$\lvert \text{val}(f(x)) - \text{val}(\hat{f}(x)) \rvert$	$\sum_{i=0}^{m-1} 2^i \cdot \left(f(x) \oplus \hat{f}(x) \right)$
000	00	10	1	2	2
001	10	10	0	0	0
010	10	10	0	0	0
011	00	00	0	0	0
100	01	10	1	1	3
101	11	00	1	3	3
110	11	11	0	0	0
111	01	01	0	0	0
			$\text{er}(f, \hat{f}) = 3/8$	$\text{wce}(f, \hat{f}) = 3$	$\text{whd}(f, \hat{f}) = 8$

in the literature. The *worst-case error* (Eq. 3) does take the values of f and $\hat{f}$ into account and returns the largest error. The last error metric (Eq. 4) is a weighted variant of the Hamming distance metric derived from the mean Hamming distance as presented in [19]. The weight parameters 2^i ensure that the bit position of an error is taken into account. Therefore, we have that errors in the more significant bits have a larger influence on the error than the lower significant bits. The metrics (3) and (4) are well-suited for arithmetic circuits.

Example 2 Table 1 shows the truth table for a function f, an approximation $\hat{f}$ of f, and the error metric values for the three error metrics introduced above.

All these error metrics have in common that they are computationally expensive to determine [20], making iterative ALS techniques that rely on repeatedly evaluating an error metric infeasible. It is possible to accelerate the error metric computation when properties of the approximation operation on a specific data structure can be exploited [14]. In this work, we adopt the greedy bucket-based algorithm from [14] to operate on (X)AIGs and choose the weighted Hamming distance as our error metric. We will use the truth density propagation from [21] to quickly compute (an estimate of) whd (see Sect. 4.3).

4 Fast AIG Approximate Logic Synthesis

4.1 Bucket-Based Approximation Algorithm

We first describe the presented ALS technique, a bucket-based approximation algorithm, on a high level of abstraction before explaining the technical details in the following sections.

The main idea behind the algorithm is, given an AIG G, to define multiple buckets that contain approximations of G that have less nodes than G. Each bucket has an error threshold. Only approximated AIGs that have an whd error lass than

the threshold are stored in the bucket. All buckets are sorted in ascending order of the threshold. The algorithm iterates over the AIGs currently stored in the buckets and tries to further approximate them without exceeding the error threshold of the last bucket. When no further approximations are possible, the algorithm terminates and returns the buckets.

The returned buckets form the Pareto front for the optimization criteria number of AIG nodes (which we use as a stand-in for the circuit's area) and the weighted Hamming distance error metric.

The algorithm is depicted in Algorithm 1. In lines 1–3 the buckets are set up. They are initialized with copies of the AIG that is to be approximated; the first bucket (having the smallest error threshold) is selected as the first AIG to be approximated. The algorithm runs as long as approximations have been performed (lines 4–20). For the current bucket, nodes and corresponding approximation operations that can be applied are found (line 6). Each of these approximations

Algorithm 1: Fast approximate AIG synthesis

```
   Input  : AIG to approximate A, number of buckets n with corresponding thresholds
   Output : Array bucket containing the approximate AIGs
   ▷ Initialize the buckets
1  buckets ← ⟨A, ... A⟩
2  bucket[0].changed ← true                ▷ Ensure to approximate at least once
3  currBucket ← bucket[0]
4  while currBucket.changed do
5      currBucket.changed ← false
6      approxCandidates ← findApproximationCandidates(currBucket); foreach
       Candidate c ∈ approxCandidates do
7          approx ← approximate(currBucket, c)
8          e ← error(approximated, A)
           ▷ Find bucket repBucket with
             • error(approx, A) ≤ error(repBucket, A) and
             • #approx < #repBucket.
9          repBucket ← findFittingBucket(approxB, buckets, A)
10         if repBucket exists then
11             repBucket ← approx
12             repBucket.changed ← true
13             if repBucket has a lower error threshold than currBucket then
14                 currBucket ← repBucket                ▷ Continue with repBucket
15
16             else
                   ▷ Continue with next bucket
17                 currBucket ← next(currBucket)
18         else
19             currBucket ← next(currBucket)             ▷ Continue with next bucket
20
21 return buckets
```

is applied (line 7), and the result is evaluated for whether it can be put into one of the buckets, i.e., whether there is a bucket containing an AIG with a larger error and more nodes (line 9). If that is the case, the corresponding bucket is updated (lines 11–12). If the updated bucket has a lower error threshold than the currently used bucket, this bucket is used in the next iteration (lines 13–14); otherwise, the next bucket is used (lines 17 and 19).

We implemented the proposed ALS method in the state-of-the-art logic synthesis tool ABC [22].

4.2 Approximation Operations

In this work, we make use of the two different approximation operations, *XOR replacement* and *constant replacement*, as they can be efficiently implemented on the AIG data structure. After a replacement has been conducted, the structure of the AIG has changed, and new optimization rules may apply. Therefore, after each replacement, the AIG is again optimized by ABC.

Example 3 After replacing an input A of an AND node v (i.e., v represents $A \wedge B$) with a constant 0, e.g., allows to further replace the node v with the constant 0 as we have

$$A \wedge B \overset{\text{replace A with 0}}{\rightsquigarrow} = 0 \wedge B = 0.$$

XOR Replacement The idea behind XOR replacement is to first identify nodes in the initial AIG G that form an XOR operation and then to replace them by a single node only.

In order to identify the nodes forming an XOR operation, the AIG G is transformed into an equivalent XAIG G' (see step (a) in Fig. 2). This step is handled automatically by ABC. Note that the transformation does not necessarily replace *all* AND nodes by XOR nodes.

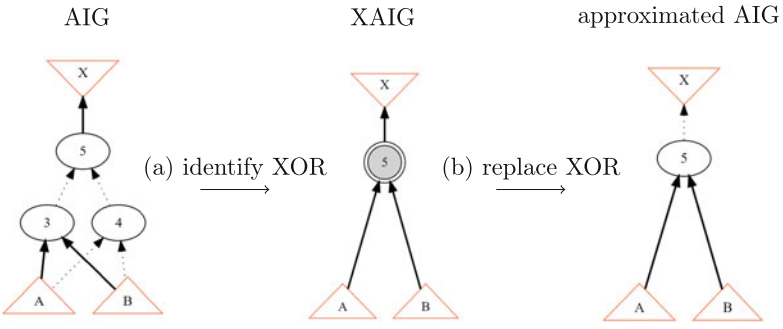


Fig. 2 Exemplary XOR replacement example

Table 2 XOR Replacements based on the truth densities of A and B

$A \oplus B$		td(A)		
		25%	50%	75%
td(B)	25%	$A \vee B$	$A \vee B$	$\neg(A \wedge B)$ / $A \vee B$
	50%	$A \vee B$	$\neg(A \wedge B)$ / $A \vee B$	$\neg(A \wedge B)$
	75%	$\neg(A \wedge B)$ / $A \vee B$	$\neg(A \wedge B)$	$\neg(A \wedge B)$

The second step (see step (b) in Fig. 2) then replaces the found XOR node by a single AND node. Note that one or multiple edges in the graph might be negated in this process (see the outgoing edge of node 5 on the right of Fig. 2).

In order to find suitable replacements for the XOR node, we investigated the XOR behavior depending on the truth densities of the inputs of the XOR operation. Table 2 shows the replacements introducing the smallest error. We obtained the replacements via exhaustive testing.

The tie breaker in the case when both NAND and OR are suitable replacements, we chose the NAND replacement when either $\text{td}(A) > (1 - \text{td}(B))$ or $\text{td}(B) > (1 - \text{td}(A))$ holds. We replace the XOR node with an OR node otherwise.

Example 4 Consider the AIG on the left of Fig. 2 and assume $\text{td}(A) = 0.7$ and $\text{td}(B) = 0.5$. The nodes 3, 4, and 5 form an XOR operation and, hence, can be replaced according to Table 2. As both NAND and OR are valid replacements, we have to check the tie breaker to decide on the actual replacement. As we have $\text{td}(B) = 0.5 > 0.3 = (1 - 0.7) = (1 - \text{td}(A))$, the three nodes are replaced by a single NAND node.

Replacing any XOR node v in the AIG of a function f according to Table 2 yields an approximation $\hat{f}$ where

$$\text{ON}(f_v) \subseteq \text{ON}(\hat{f}_v) \vee \text{ON}(\hat{f}_v) \subseteq \text{ON}(f_v) \tag{5}$$

holds. Here $f_v / \hat{f}_v$ is the function represented by the node v. Equation 5 describes over-/underapproximations, respectively. Note that the property in Eq. (5) holds only locally at the replaced node.

Constant Replacement When a node in the AIG has a truth density close to either 0 or 1, it can be considered a constant 0 or 1 node. To make this decision, the user can specify a corresponding decision threshold. As long as this threshold is less than 0.5, i.e., replace the node v with a constant 0/1 when $\text{td}(v) < 0.5 / \text{td}(v) > 0.5$, the constant replacement operation also has the property in Eq. (5).

For this replacement operation, the AIG G does not need to be transformed into an XAIG.

4.3 Fast Computation of the Weighted Hamming Distance

We review the definition of the weighted Hamming distance error metric from Eq. (4)

$$\text{whd}(f, \hat{f}) = \sum_{i=0}^{m-1} 2^i \cdot \left(\sum_{x \in \mathbb{B}^n} \left(f_i(x) \oplus \hat{f}_i(x) \right) \right) \tag{6}$$

and note that the computation of the Hamming distance on the individual output functions f_i can be computed using the truth density as follows:

$$= \sum_{i=0}^{m-1} 2^i \cdot \left(2^n \cdot |\text{td}(f_i) - \text{td}(\hat{f}_i)| \right) = 2^n \cdot \sum_{i=0}^{m-1} 2^i \cdot |\text{td}(f_i) - \text{td}(\hat{f}_i)|. \tag{7}$$

For this equality to hold, the function $\hat{f}$ must have been obtained by applying an approximation operation for which the property in Eq. (5) holds.

Example 5 Consider the two approximations $\hat{\text{X}}$ and $\tilde{\text{X}}$ shown in the truth table in Table 3. For the approximation operation yielding $\hat{\text{X}}$, property (5) holds, i.e., we have that $\text{ON}(\text{X}) \subset \text{ON}(\hat{\text{X}})$ holds. This property does not hold for the approximation $\tilde{\text{X}}$. Computing the whd using Eq. (7) shows that the over-/underapproximation property is crucial:

$$\text{whd}(\text{X}, \hat{\text{X}}) = 2^2 \cdot |0.50 - 0.75| = 4 \cdot 0.25 = 1$$

$$\text{whd}(\text{X}, \tilde{\text{X}}) = 2^2 \cdot |0.50 - 0.50| = 4 \cdot 0.00 = 0$$

The value $\text{whd}(\text{X}, \tilde{\text{X}}) = 0$ is clearly incorrect.

The advantage of computing whd using Eq. (7) instead of using the initial definition of Eq. (4) is that the actual time necessary to determine the value can be greatly reduced if the computation of the truth densities can be done quickly. We will see in Sect. 4.4 how this is possible.

Table 3 Approximating X using operations for which the over-/underapproximation property Eq. (5) does hold ($\hat{\text{X}}$) and does not hold ($\tilde{\text{X}}$)

A	B	X	$\hat{\text{X}}$	$\tilde{\text{X}}$
1	1	0	0	0
0	1	1	1	1
1	0	1	1	0
0	0	0	1	1

As the proposed ALS method (see Algorithm 1) is an iterative approach, many whd values have to be computed during a synthesis run. When the total number of inputs does not exceed 16, the AIG can be fully evaluated and exact results can be computed. When the AIG grows beyond this, the truth density and, hence, the whd are computed iteratively by determining the whd locally for the approximated node only. We then adopt an additive model accumulating the locally computed errors until the output node is reached. This additive model along with the fact that consecutive errors that might cancel each other out (a situation also known as *error masking*) are not taken into account leads to an overestimation of the total error. The upside of this simplification is that it allows for a very fast estimation of the total whd.

4.4 Truth Density Computation

So far, we used the truth density values of all (X)AIG nodes without considering how to actually compute them. In this work, we make use of two different means of obtaining the truth densities of the nodes.

The first means of obtaining the truth density is to directly use ABC. The tool estimates the truth density values of the nodes by running a number of simulations of the graph, i.e., evaluating the graph for a given number of randomly generated inputs. The quality of the result greatly varies with the number of simulations and, hence, the time one is willing to spend on the estimation.

As the computation of the densities is crucial for both, the decision on which node to replace and the computation of the weighted Hamming distance error metric, we chose to use the error propagation method presented in [21]. While its intended use is to propagate the error rate through a general Boolean network, it can easily be applied for our use case as (X)AIGs are nothing but a specific Boolean network and the truth density is already computed by the approach as a "by-product."

The speed of the approach from [21] stems from not having to perform full simulations of the (X)AIGs but computes the truth density using symbolic variables. It should be noted that computed densities are only exact in case when there is no fanout reconvergence in the (X)AIG. Nevertheless, extensive tests have shown that the degradation of the results in case of reconvergences is negligible.

5 Experimental Evaluation

5.1 Experimental Setup

We implemented the proposed ALS technique in the state-of-the-art logic synthesis tool ABC [22] using the probabilistic error propagation tool from [21].

As benchmark circuits, generic n-bit adders and multipliers as well as the EvoApproxLib$^{\text{LITE}}$ library [18] are used.

Instead of directly specifying the whd value of the buckets, we define a *threshold* $t \in [0, 1]$ that reflects how large the error in the most significant bit of the output is allowed to be in percent. This allows to define buckets that capture similar error behavior for circuits of different size, i.e., one does not have to (manually) compute different bucket values for an 8-bit and an 16-bit multiplier. A threshold value t can be translated in an estimate on the whd via $\text{whd}(f, \hat{f}) \approx t \cdot 2^n \cdot 2^{m-1}$.

All experiments were executed on an AMD Ryzen 5 3600XT 6-Core CPU with 3.80-GHz and 16-GB memory running Ubuntu 20.04 in WSL 2 on Windows 10 Build 19044.1645.

5.2 Scalability

To assess the scalability of our approach, we performed approximate logic synthesis on adders of increasing bit width using 5 buckets with threshold values 0.0156, 0.0.03125, 0.0625, 0.125, and 0.25. In the experiments, it turned out that the error propagation implementation has a memory leak preventing it to be used for AIGs with more than $\approx$ 300 nodes. Therefore, the following results were obtained using ABC's simulation method.

The results of the synthesis runs are presented in Table 4. For each bit width, the results for each bucket are listed in a separate line. The approximated AIGs were converted to a list of logic gates using ABC. Afterward, the area and delay have been computed by ABC using the `mcnc.genlib` gate library. For each physical aspect, the number of gates, the area, and the delay, we present the reduction/increase in the aspect in percent after the absolute values in the table. We further report the whd for the AIGs.

Using the number of AIG nodes as a stand-in for the circuit area works well: the reduction in nodes is qualitatively reflected in the reduction in the number of gates and the reduction of the area. As can be seen, the goal of optimizing circuits for area has been achieved. It is interesting to see the reduction remains in the range $\approx$ 65%–75% for threshold values up to 0.125. Only after allowing for 25% weighted errors in the most significant bit, further size reductions are achieved.

While our method is capable of reducing the area, it does, in turn, increase the delay of the circuit (usually in the $\approx$ 112%–125% range). This value, again, drops when a large threshold is used. As we do not explicitly optimize for delay, this is an acceptable trade-off.

As can be seen, the actual whd values for the buckets increase with increasing bit width of the adders. This shows that choosing a means to describe buckets that abstracts away the bit width is helpful.

Table 4 ALS results for adders of varying bit width. For each bucket, the number AIG node, number of gates, area, delay, and whd are reported. For the number of nodes/gates, the area, and the delay, the relative change to the unapproximated AIG is also shown

Threshold	Nodes	%	Gates	%	Area	%	Delay	%	whd
16 bit [Run-time: 34.64 s]									
Unapprox.	158	–	105	–	215	–	33.70	–	–
0.0156	141	89%	66	63%	170	79%	39.30	117%	2.75
0.03125	139	88%	66	63%	166	77%	38.80	115%	3.00
0.0625	137	87%	66	63%	162	65%	38.30	114%	3.25
0.125	135	85%	66	63%	158	73%	37.80	112%	3.50
0.25	131	83%	61	58%	151	70%	35.30	104%	4.00
32 bit [Run-time: 365.54 s ≈ 5 m]									
Unapprox.	318		217	–	439	–	65.70	–	–
0.0156	270	85%	131	60%	319	73%	77.40	118%	7.00
0.03125	269	85%	132	61%	315	72%	77.20	118%	7.25
0.0625	266	84%	133	61%	309	70%	78.00	119%	7.75
0.125	264	83%	132	61%	306	70%	77.70	118%	8.00
0.25	193	60%	83	38%	176	40%	44.60	66%	16.64
64 bit [Run-time: 2527.54 s ≈ 42 m]									
Unapprox.	638	–	441	–	887	–	129.70	–	–
0.0156	523	82%	261	59%	598	67%	160.30	124%	15.50
0.03125	521	82%	261	59%	594	67%	159.80	123%	15.75
0.0625	519	81%	261	59%	590	67%	159.30	123%	16.00
0.125	517	81%	261	59%	586	66%	158.80	122%	16.25
0.25	461	72%	223	51%	483	54%	134.40	103%	21.45
128 bit [Run-time: 22738.28 s ≈ 6 h]									
Unapprox.	1278	–	889	–	1783	–	257.70	–	–
0.0156	1034	81%	516	58%	1179	66%	315.70	123%	32.48
0.03125	1032	81%	516	58%	1175	66%	315.20	123%	32.73
0.0625	1030	81%	516	58%	1171	66%	314.70	122%	32.98
0.125	1028	80%	516	58%	1167	65%	314.20	122%	33.23
0.25	501	39%	194	22%	211	12%	57.20	22%	61.72

5.3 *Multi-Objective Optimization for Area and* whd

The benchmark library EvoApproxLib$^{\text{LITE}}$[1] [18, 23] provides a selection of approximate adders and multipliers. They have been synthesized via exhaustive search with respect to various error metrics (including er and wce) as well as area and power consumption. The benchmark set does not evaluate the whd error metric.

As the final buckets of the presented approach form the Pareto front of the multi-objective optimization problem with the optimization criteria whd and area, we

[1] The benchmark library is publicly available at https://ehw.fit.vutbr.cz/evoapproxlib/.

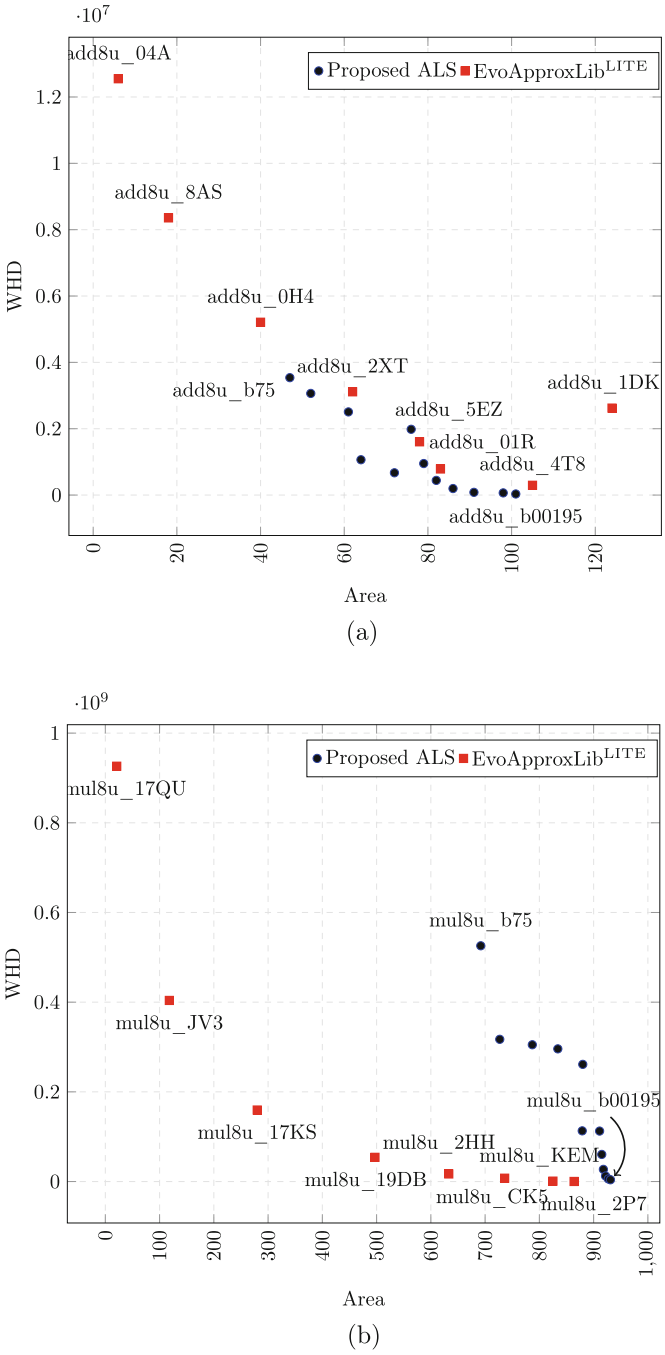


Fig. 3 Comparison of the synthesis results of the proposed approach (dark blue dots) and EvoApproxLib$^{\text{LITE}}$ (red squares) with respect to area and the weighted Hamming distance error metric for (**a**) 8-bit unsigned adders and (**b**) 8-bit unsigned multipliers

compare our results with the exhaustive results from EvoApproxLib^{LITE}. We select the adders and multipliers from the benchmark set that form the Pareto front with respect to area and the wce error metric and compute the whd values for them so that we can compare the benchmark circuits to our results. The circuits optimized for this metric were chosen as wce does take into account the order of the output bits, and, therefore, the corresponding circuits allow for the fairest comparison.

The comparison for 8-bit unsigned adders and multipliers is shown in Fig. 3. The notation for our circuits is as follows: "add8u_b75" refers to an unsigned 8-bit adder from the bucket with a threshold of 0.75. For EvoApproxLib^{LITE}, the naming scheme is of the form "add8u_⟨ID⟩" and directly taken from their website.

For the adders (Fig. 3a), the proposed ALS method clearly produces better results than EvoApproxLib^{LITE}. These results can be explained, in part, by the fact that EvoApproxLib^{LITE} optimized for a different error and in part by the fact that the computation of the sum bits in an adder basically is a large XOR gate. When looking at the results for the multipliers (Fig. 3b), one can see that the applied approximations are not resulting in points close to the Pareto front any more. When investigating what approximation operations have been chosen by the proposed ALS algorithm (see Table 5), one can see that the ratio of XOR replacement over constant replacements for the adder is higher than for the multiplier. This further hints that XOR replacement is well-suited for adders while multipliers do not benefit from this particular kind of approximation.

5.4 *Truth Density Computation*

To investigate the difference in execution time between the ABC simulation-based truth density estimation and the method from [21], we synthesized adders of increasing bit width using 5 buckets with threshold values 0.0156, 0.0.03125, 0.0625, 0.125, and 0.25. The results are reported in Fig. 4. As can be seen, the error propagation approach clearly excels with respect to the execution time. Due to the memory leakage issue (see Sect. 5.2), we can not show results for larger circuits. While using ABC for the error estimation already is fast, using error propagation shows a great potential to further accelerate our proposed ALS technique.

Table 5 Number of XOR/Constant replacements for 8-bit adders and multipliers

Circuit	Replace XOR	Replace Constant	Ratio
8-bit adder	37	4	9.25
8-bit multiplier	136	38	3.6

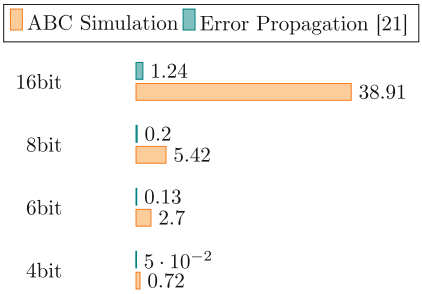


Fig. 4 Execution time of the proposed ALS method in seconds for adders of varying bit width

6 Conclusion and Outlook

We presented a novel and fast greedy, bucket-based approximate logic synthesis technique working on AIGs that aims to minimize both the area of the resulting circuit and, at the same time, the error introduced by the approximations. We chose the weighted Hamming distance error metric whd to assess the functional quality of the circuit as it takes into account the order of the output bits. We found a means of effectively computing whd via computing truth densities and exploiting properties of the used approximation operations. The effectiveness of the presented method has been evaluated in a set of experiments.

The next step is to find the memory leakage in the fast error propagation tool to further enhance the speed of the proposed method.

Acknowledgments The paper has been partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)—450987171.

References

1. Moore, G.E.: Cramming more components onto integrated circuits. Electronics **38**(8), pp. 114 (1965)
2. Waldrop, M.M.: The chips are down for Moore's law. Nature News **530**(7589), 144–147 (2016). Visited on 06/20/2022
3. Zhu, N., et al.: Design of low-power high-speed truncation-error-tolerant adder and its application in digital signal processing. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **18**(8), 1225–1229 (2010)
4. Venkatesan, R., et al.: MACACO: modeling and analysis of circuits for approximate computing. In: International Conference On Computer Aided Design (2011). Visited on 09/19/2018
5. Han, J., Orshansky, M.: Approximate computing: an emerging paradigm for energy-efficient design. In: European Test Symposium (2013)
6. Mittal, S.: A survey of techniques for approximate computing. ACM Comput. Surv. **48**(4), 1–33 (2016). Visited on 02/28/2019
7. Venkataramani, S., et al.: Approximate computing and the quest for computing efficiency. In: Design Automation Conference (2015). Visited on 04/11/2019

8. Scarabottolo, I., et al.: Approximate logic synthesis: a survey. Proc. IEEE **108**(12), 2195–2213 (2020)
9. Shafique, M., et al.: A low latency generic accuracy configurable adder. In: Design Automation Conference (2015). Visited on 02/08/2019
10. Barbareschi, M., Iannucci, F., Mazzeo, A.: A pruning technique for B&B based design exploration of approximate computing variants. In: International Symposium on VLSI (2016)
11. Keszocze, O., Kiessling, M.: Approximate computing extensions for the clash HDL compiler. In: Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (2021)
12. Soeken, M., et al.: BDD minimization for approximate computing. In: Asia and South Pacific Design Automation Conference (2016)
13. Shirinzadeh, S., et al.: An adaptive prioritized e-preferred evolutionary algorithm for approximate BDD optimization. In: Proceedings of the Genetic and Evolutionary Computation Conference (2017). Visited on 10/19/2018
14. Keszocze, O.: BDD-based error metric analysis, computation and optimization. IEEE Access **10**, 14013–14028 (2022)
15. Chandrasekharan, A., et al.: Approximation-aware rewriting of AIGs for error tolerant applications. In: International Conference On Computer Aided Design (2016). Visited on 10/24/2018
16. Mishchenko, A., Chatterjee, S., Brayton, R.: DAG-aware AIG rewriting: A fresh look at combinational logic synthesis. In: Design Automation Conference (2006). Visited on 07/31/2019
17. Háleček, I., Fišer, P., Schmidt, J.: On XAIG rewriting. In: International Workshop on Logic & Synthesis (2017)
18. Mrazek, V., et al.: EvoApprox8B: library of approximate adders and multipliers for circuit design and benchmarking of approximation methods. In: Design, Automation and Test in Europe (2017). Visited on 10/17/2018
19. Vasicek, Z.: Formal methods for exact analysis of approximate circuits. IEEE Access **7**, 177309–177331 (2019)
20. Keszocze, O., Soeken, M., Drechsler, R.: The complexity of error metrics. Inf. Process. Lett. **139**, 1–7 (2018). Visited on 08/08/2018
21. Echavarria, J., et al.: Probabilistic error propagation through approximated Boolean networks. In: Design Automation Conference (2020)
22. Brayton, R., Mishchenko, A.: ABC: An academic industrial-strength verification tool. In: Touili, T., Cook, B., Jackson, P. (eds.) Computer Aided Verification. University of California, Berkeley (2010)
23. Mrazek, V., Vasicek, Z., Sekanina, L.: EvoApproxLib: extended library of approximate arithmetic circuits. In: Workshop on Open-Source EDA Technology (2019)

External Don't Cares in Logic Synthesis

Siang-Yun Lee, Heinz Riener, and Giovanni De Micheli

1 Introduction

Logic synthesis, or, more specifically, technology-independent logic optimization, is a step in the VLSI design flow after RTL synthesis and before technology mapping, attempting to optimize combinational circuits on technology-independent representations, such as *AND-inverter graphs* (AIGs). As a bottom line, the produced result of a logic synthesis algorithm must respect the given functionality of the circuit. To date, this means that the output circuit should be functionally equivalent to the original one, and is usually verified by performing *combinational equivalence checking* (CEC) [5] on the two circuits. However, this requirement might be too strong in some cases. Further high-effort optimization can be enabled by relaxing the requirement of exact functional equivalence and allowing flexibilities external to the combinational circuit under optimization.

Don't cares are flexibilities in logic functions or logic networks where output values of some (local) functions can be changed without violating the (global) specification [3]. Don't-care conditions may be derived on various scales, from interconnections of logic gates within a combinational network [4] to interactions between submodules in a system [12]. Computation and utilization of don't-care conditions in combinational logic synthesis have often been formulated using incompletely specified functions [2], also known as permissible functions [11]. Don't cares play a central role in logic synthesis. However, due to the intrinsically high computational complexity of don't-care computation, methods to (under-)

S.-Y. Lee (✉) · G. De Micheli
École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland
e-mail: siang-yun.lee@epfl.ch; giovanni.demicheli@epfl.ch

H. Riener
Cadence Design Systems, Munich, Germany

R. Drechsler, S. Huhn (eds.), *Advanced Boolean Techniques*,
https://doi.org/10.1007/978-3-031-28916-3_3

approximate them were developed [9, 14, 15]. Nowadays, more powerful and scalable computation of don't cares enabled by *satisfiability* (SAT) solving and simulation is commonly used, but consideration of don't cares is still limited to those within a combinational network [10].

In contrast to internal don't cares computed within a network, external don't cares are flexibilities arising from outside of the combinational network under optimization, derived from a higher-level perspective of the system. For example, cascaded finite-state machines may produce don't-care input sequences for each other [12]. As another example, sometimes the system is partitioned into submodules and optimized separately. While their boundaries are intended to be kept, flexibilities on the input-output relations of individual submodules due to their interactions are allowed. Considering external don't cares essentially changes the problem from optimizing a (completely specified) Boolean function into optimizing a Boolean relation. The solution space is enlarged and the problem complexity is much higher; thus, there is currently no open-source logic synthesis tool that supports taking and utilizing external don't cares. Nevertheless, with the increased computation power affordable nowadays, solving such optimization problems should be possible on smaller benchmarks. Moreover, in some applications, users of logic synthesis tools crave to optimize their circuit as much as possible and are willing to afford higher runtime.

This paper serves as a pioneer toward support of external don't cares in logic synthesis. During this journey, we will lay the foundation with mathematical definitions of don't-care conditions in general, explore different flavors of external don't cares, view the general problem of logic synthesis from a Boolean relation perspective, and finally take the first step of considering external don't cares in logic optimization. We will show with experimental demonstrations that external don't cares indeed open up more optimization opportunities that would have been impossible without them. In the end, we will also point out possible directions for future research.

2 Background and Terminologies

2.1 Boolean Functions and Boolean Relations

A *Boolean variable* is a variable taking values in the *Boolean domain* $\mathbb{B} = \{0, 1\}$. The (n-dimensional) *Boolean space* $\mathbb{B}^n$ is an n-ary Cartesian power of the Boolean domain. An (n-input, single-output, completely specified) *Boolean function* is a function $f : \mathbb{B}^n \to \mathbb{B}$ of n Boolean variables. Multi-output Boolean functions can be seen as an ordered set of single-output functions.

A *Boolean relation* $\mathcal{R}$ is a binary relation over two Boolean spaces $\mathcal{R} \subseteq \mathbb{B}^n \times \mathbb{B}^m$, a *domain* ($\mathbb{B}^n$) and a *codomain* ($\mathbb{B}^m$). Boolean functions are special cases of Boolean relations. More specifically, they can be classified into two types:

- *Completely specified* functions are special cases of Boolean relations where the relations are *functional* (i.e., an element in the domain maps into one unique element in the codomain) and *total* (i.e., every element in the domain maps into an element in the codomain). When describing Boolean functions as Boolean relations, an element in the domain, which is a value assignment to all the function's input variables, is also called a *minterm*.
- *Incompletely specified* functions are Boolean functions for which the output values under some minterms are not specified. In other words, for some minterm $\mathbf{b} \in \mathbb{B}^n$, the output value can be either 0 or 1. An incompletely specified function can be represented as a nonfunctional Boolean relation $\mathcal{R}$ having, for some minterms $\mathbf{b}$, both $(\mathbf{b}, 0) \in \mathcal{R}$ and $(\mathbf{b}, 1) \in \mathcal{R}$.

Given an incompletely specified function as a Boolean relation $\mathcal{R} \subseteq \mathbb{B}^n \times \mathbb{B}^m$, a completely specified function $f : \mathbb{B}^n \to \mathbb{B}^m$ is *compatible* with $\mathcal{R}$ if

$$\forall \mathbf{b} \in \mathbb{B}^n, (\mathbf{b}, f(\mathbf{b})) \in \mathcal{R}. \tag{1}$$

When not explicitly noted, *functions* in the remaining of this paper refer to single-output, completely specified Boolean functions.

2.2 Logic Networks and Functions in a Network

Logic networks (or simply *networks*) are technology-independent representations of digital circuits. A logic network N is defined by a four-tuple $N = (I, V, E, O)$, where the two sets (V, E) define a directed acyclic graph. The first set I is the set of *primary inputs* (PIs) to the network. Each element in the vertex set V, referred to as a *node* n, models either a logic gate or a PI. Thus, $I \subseteq V$. Each element (n_i, n_o, c) in the edge set $E \subseteq V \times V \times \mathbb{B}$ models a wire from node n_i to node n_o with a complementation tag $c \in \{0 = \text{regular}, 1 = \text{complemented}\}$ recording the existence of an inverter on the wire. n_i is said to be a *fanin* of n_o and n_o is said to be a *fanout* of n_i. Finally, each *primary output* (PO) in O is a tagged node (n, c) modeling an outgoing wire from a gate or a PI, with or without an inverter.

Cuts A *cut* in a network, defined over a given set $R \subseteq V$ of *root* nodes, is a set C of nodes such that any path from a PI to a root includes a node in C. Let $\textsc{Cuts}(R)$ denote the set of all cuts for the set R:

$$C \in \textsc{Cuts}(R) \text{ if } \forall i \in I, r \in R, \forall p : i \overset{p}{\rightsquigarrow} r, \exists n \in C : n \in p. \tag{2}$$

When R contains only one node n, $\textsc{Cuts}(R)$ may be abbreviated as $\textsc{Cuts}(n)$ and is also referred to as a cut of n:

$$\textsc{Cuts}(n) \triangleq \textsc{Cuts}(\{n\}). \tag{3}$$

Conversely, given a set C of nodes, a node n is said to be *supported* by C if C is a cut of n. A *logic cone* between a cut $C \in \textsc{Cuts}(n)$ and a node n is the set of all nodes on any path from a node in C to n. All nodes in the logic cone are supported by C.

A cut of a network N is a cut where R is the set of nodes referenced by POs:

$$\textsc{Cuts}(N) \triangleq \textsc{Cuts}(\{n : \exists c, (n, c) \in O\}). \tag{4}$$

Given any set R of roots, the identical set $C = R$ is always a cut by definition; thus, such cut is said to be a *trivial cut*. Also, the set I of PIs is always a cut in a network for any possible R.

Global Function of Nodes Each node n in a network computes a Boolean function $f_n : \mathbb{B}^{|I|} \to \mathbb{B}$ in terms of the PIs, called the node's *global function*. To express the global functions, a Boolean variable x_i is associated with each PI $i \in I$. Let $\mathbf{x} = (x_1, \ldots, x_{|I|})$ be the set of all PI variables. By definition, the function of a PI node $i \in I$ is $f_i(\mathbf{x}) = x_i$. Then, in a topological order, the functions of all nodes in the network can be computed by composing the functions of a node's fanins with the function of the corresponding logic gate. Finally, the PO functions are computed by taking the function of a PO node and inverting if the PO is complemented.

Node Function in Terms of a Cut The function of a node may also be expressed in terms of a cut supporting it. Given a node n and a cut $C \in \textsc{Cuts}(n)$, the *local function* $f_n^C : \mathbb{B}^{|C|} \to \mathbb{B}$ is the Boolean function derived by associating a Boolean variable with each node in C and computing the local functions of each node in the logic cone between C and n in a topological order. The global functions are a special case of local functions using the PI set I as the cut:

$$f_n \triangleq f_n^I. \tag{5}$$

2.3 Don't-Care Conditions

A *don't care* for an incompletely specified function is a minterm for which the output value is not specified. In a logic network, although all node functions (in terms of any cut) are completely specified, for some nodes, there may be some minterms where the output values of their functions are *flexible*. In other words, the function f_n^C of a node n in terms of cut C may be modified by changing its output value under some minterms without affecting the global functions of any PO. As a consequence, an incompletely specified function where these minterms are don't cares and the output values under the other minterms are the same as f_n^C can be used to resynthesize the logic cone between C and n. Two types of internal don't cares, arising from different reasons, may appear in logic networks:

- Given a cut $C \in \textsc{Cuts}(R)$ supporting a set R of nodes[1] and let $\mathbf{x} = (x_1, \ldots, x_{|C|})$ be Boolean variables associated with each node in C, a value assignment $\mathbf{b}_C \in \mathbb{B}^{|C|}$ to $\mathbf{x}$ (i.e., a minterm of the local functions f_n^C of any node $n \in R$) is a *satisfiability don't care* (SDC) if this value combination never appears under any PI value assignment:

$$\nexists \mathbf{b}_I \in \mathbb{B}^{|I|}, (f_n(\mathbf{b}_I) : n \in C) = \mathbf{b}_C. \tag{6}$$

- Given a node n and a cut $C \in \textsc{Cuts}(n)$ and let $\mathbf{x} = (x_1, \ldots, x_{|C|})$ be Boolean variables associated with each node in C, a value assignment $\mathbf{b}_C \in \mathbb{B}^{|C|}$ to $\mathbf{x}$ (i.e., a minterm of the local function f_n^C) is an *observability don't care* (ODC) with respect to n if none of the PO functions are affected by flipping the output value of f_n^C under $\mathbf{b}_C$:

$$\forall \mathbf{b}_I \in \mathbb{B}^{|I|}, (f_n(\mathbf{b}_I) : n \in C) = \mathbf{b}_C \implies \forall o \in O, f_o^*(\mathbf{b}_I) = f_o(\mathbf{b}_I), \tag{7}$$

  where f_o^* is the PO function derived by replacing any regular outgoing edge of n with a complemented one and replacing any complemented outgoing edge of n with a regular one.

3 Computation of Internal Don't Cares

Appearance of "don't care" as a technical term in the literature dates back to as early as the 1980s [3]. Pioneering research attempted to derive don't cares in multi-level networks and use them in two-level minimization to resynthesize part of the network [2]. Theories on don't-care computation were formulated based on symbolic computations propagated through the network [4, 11]. Until the late 1990s, computation of don't cares had been implemented using *binary decision diagrams* (BDDs). Due to scalability concerns, approximated computation was adopted [9], and the compatibility of ODCs was studied to avoid recomputation of ODCs in the network once an ODC is used to change the function of a node [14]. Since the early 2000s, computation tools of don't cares have moved from BDDs to SAT, enabling using complete, instead of approximate, don't cares while maintaining scalability [10].

[1] The supported set R is not involved in the definition of SDCs, so it can, in theory, be empty and C is not necessarily a cut. Although one may define and compute SDCs for any set C of nodes, in practice, SDCs are only meaningful when C is indeed a cut, as SDCs are used to optimize nodes in R.

In many modern logic synthesis tools, internal don't cares are derived locally (under-approximated) using bit-parallel circuit simulation:

- To compute the SDCs for a given set C of nodes, we first find another cut $C_0 \in \text{CUTS}(C)$ supporting C. Then, we perform circuit simulation by assigning projection functions to nodes in C_0 and obtain the local functions of nodes in C in terms of C_0, represented as truth tables. Finally, by analyzing each bit in the truth tables, we identify the value combinations at C that do not happen, which are the SDCs at C.
- To compute the ODCs with respect to a node n, we first mark the transitive fanout cone of n for a predefined number of levels and collect the set R of nodes having fanouts outside of this transitive fanout cone. Then, we find a cut $C \in \text{CUTS}(R)$ supporting R and perform circuit simulation to obtain the local functions f_R of nodes in R in terms of C. After adding a temporary inverter at the output of n, we perform another simulation to obtain f_R^*. Finally, we compare the two simulation results to identify the minterms where f_R and f_R^* have identical values, which are the ODCs with respect to n.

4 Definition and Representation of External Don't Cares

The general problem of technology-independent combinational logic synthesis asks for generating a logic network that implements the desired output functions and is optimized according to some predefined cost objective. Often, the desired functionalities are given as an unoptimized network. Besides improving the cost objective, a logic synthesis algorithm must preserve the functionalities of the given network. More precisely, the global PO functions must not change after optimization.

However, the desired functionalities may not be completely specified, and there may be don't-care conditions external to the network under synthesis. For example, due to the interplay between the network and its environment (other cascaded circuits, previous- and next-stage sequential circuits, or user inputs), some input value combinations may never appear, or some output values are not used ("observed") under certain conditions. These *external don't cares* (EXDCs) can be leveraged to further optimize the network. As it is impossible to derive external don't cares from the network alone, they have to be given to a combinational optimization algorithm from a higher-level algorithm.

4.1 External Controllability Don't Cares (External SDCs)

Extending the definition of SDC to the input boundary, a value assignment to the PIs that will never appear is called an *external controllability don't care* (EXCDC). These don't cares are controlled by the environment external to the network.

Mathematically, EXCDCs are essentially a special case of SDCs where the cut C is the set of PIs. The set of EXCDCs of a network N may be given as a function $f^{\mathrm{CDC}} : \mathbb{B}^{|I|} \to \mathbb{B}$:

$$f^{\mathrm{CDC}}(\mathbf{b}_I) = 1 \iff \mathbf{b}_I \text{ is an EXCDC.} \tag{8}$$

4.2 External Observability Don't Cares

Extending the definition of ODCs to the output boundary, external ODCs are conditions under which some PO values are not of interest. Depending on the reasons of such situations, there are several ways one may wish to define external ODCs.

As a Function of PIs For each PO $o \in O$, the condition under which the value of o is not observed may be specified as a function of PI values. For example, when the network describes the transition and output logic of a Mealy finite-state machine, it may appear that for some previous states (PIs of the network), an output is not used. In this case, the external ODCs are described as a multi-output function $f^{\mathrm{ODCI}} : \mathbb{B}^{|I|} \to \mathbb{B}^{|O|}$:

$$\text{For each } o \in O,\ f_o^{\mathrm{ODCI}}(\mathbf{b}_I) = 1 \iff \mathbf{b}_I \text{ is an EXODC for } o. \tag{9}$$

As a Function of Other POs For each PO $o \in O$, the condition under which the value of o is not observed may be specified as a function of other PO values. For example, when the outputs of the network are used in the next stage as a series of cascaded conditional statements such that if a PO of higher priority evaluates to 1, then the lower-priority POs do not matter. In this case, the external ODCs are described as a multi-output function $f^{\mathrm{ODCO}} : \mathbb{B}^{|O|} \to \mathbb{B}^{|O|}$:

$$\text{For each } o \in O,\ f_o^{\mathrm{ODCO}}(\mathbf{b}_O) = 1 \iff \mathbf{b}_O \text{ is an EXODC for } o. \tag{10}$$

The i-th output of f^{ODCO} should not depend on its i-th input. Note that in this case, the don't-care conditions depend on the actual implementation of the network. Using one ODC to optimize and change the function of a PO may invalidate opportunities of using another ODC to optimize some other POs.

As Equivalence Classes Instead of specifying external ODCs separately for each PO, the flexible conditions might be some value combinations of a subset of POs. Figure 1 gives an example. Because of the cascaded next-stage logic at the output of N, the value combinations $o_1 = 0, o_2 = 1$, and $o_1 = 1, o_2 = 0$ have the same effect as seen from the system output (both map into $y_1 = 1, y_2 = 1$; red edges). Thus, these two PO value combinations may be classified into the same *external observability equivalence class* (EXOEC), and PI minterms that map to one of them are flexible to be re-mapped to either one (pink edges are added).

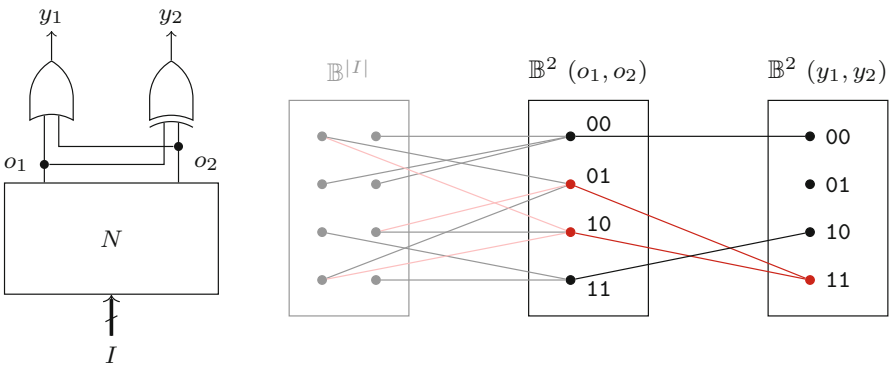


Fig. 1 Example of external observability equivalence classes

More generally, two PO value combinations are *observably equivalent* (in the same EXOEC) if their difference may not be observed when the network is immersed in a larger system. By definition, this is an equivalence relation and is *reflexive* (i.e., if a is observably equivalent to b, then b is observably equivalent to a [a and b are indistinguishable]), *symmetric* (i.e., any PO value combination is observably equivalent to itself [trivial]), and *transitive* (i.e., if a is observably equivalent to b and b is observably equivalent to c, then a is observably equivalent to c [a, b and c are indistinguishable]).

EXOECs can be given as a function $f^{\text{OEC}} : \mathbb{B}^{2\cdot|O|} \to \mathbb{B}$:

$$f^{\text{OEC}}(\mathbf{a}_O, \mathbf{b}_O) = 1 \iff \mathbf{a}_O \text{ and } \mathbf{b}_O \text{ are observably equivalent.} \tag{11}$$

Because f^{OEC} describes an equivalence relation, it must fulfill the reflexivity, symmetry and transitivity properties as described above.

4.3 Logic Synthesis from a Boolean Relation Perspective

A logic network computes a multi-output Boolean function at its primary outputs (i.e., the collection of PO global functions). Hence, it can be described as a Boolean relation. The task of logic synthesis is thus finding an (optimized) network whose output function is compatible with a given Boolean relation $\mathcal{R}$. The presence of external don't cares adds more elements into $\mathcal{R}$.

More generally, given a set C_1 of nodes and a cut $C_0 \in \text{CUTS}(C_1)$ supporting it, a Boolean relation $\mathcal{R}_{01}$ can be derived to describe the network functionality between C_0 and C_1. Moreover, if C_1 is also a cut supporting another set C_2, another Boolean relation $\mathcal{R}_{12}$ can be derived and cascaded with $\mathcal{R}_{01}$.

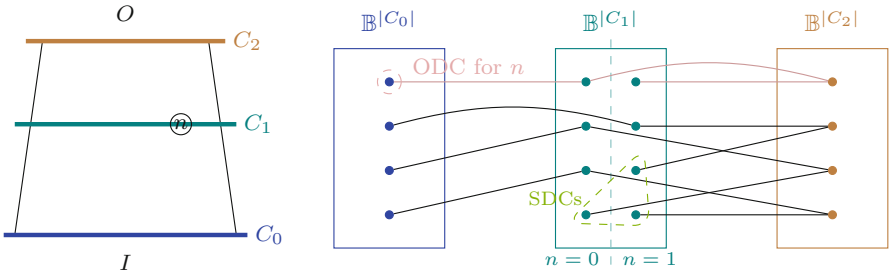


Fig. 2 Illustration of Example 1

Example 1 Let $C_1 \in \textsc{Cuts}(N)$ be a cut of the network. Let $C_0 = I$ and let $C_2 = \{n : \exists c, (n, c) \in O\}$. We may derive two Boolean relations:

$$\mathcal{R}_{01} = \{(\mathbf{b}_0, f_{C_1}^{C_0}(\mathbf{b}_0)) : \mathbf{b}_0 \in \mathbb{B}^{|C_0|}\} \tag{12}$$

$$\mathcal{R}_{12} = \{(\mathbf{b}_1, f_{C_2}^{C_1}(\mathbf{b}_1)) : \mathbf{b}_1 \in \mathbb{B}^{|C_1|}\}, \tag{13}$$

where $f_{C_1}^{C_0}$ is the function the nodes in C_1 compute in terms of C_0, and similarly for $f_{C_2}^{C_1}$.

Figure 2 illustrates the example. According to the definitions in Sect. 2.3, an (internal) SDC is an element $\mathbf{b}_1 \in \mathbb{B}^{|C_1|}$ such that

$$\nexists \mathbf{b}_0 \in \mathbb{B}^{|C_0|}, (\mathbf{b}_0, \mathbf{b}_1) \in \mathcal{R}_{01}. \tag{14}$$

Whereas an (internal) ODC for a node $n \in C_1$ is an element $\mathbf{b}_0 \in \mathbb{B}^{|C_0|}$ such that, let $\mathbf{b}_1 = f_{C_1 - \{n\}}(\mathbf{b}_0)$ be the values at $C_1 - \{n\}$ under $\mathbf{b}_0$:

$$\text{if } ((\mathbf{b}_1, 0), \mathbf{b}_2) \in \mathcal{R}_{12}, \text{ then also } ((\mathbf{b}_1, 1), \mathbf{b}_2) \in \mathcal{R}_{12}. \tag{15}$$

Generalizing internal and external don't cares, SDCs are elements in a Boolean space (which corresponds to any cut in the network) that are not mapped to by any element in a previous-stage Boolean space. In contrast, ODCs arise from two elements in a Boolean space that map to the same element in a next-stage Boolean space.

4.4 Boolean Relation as Unified Representation of External Don't Cares

We observe that none of f^{ODCI}, f^{ODCO}, f^{OEC} is general enough to express the other two. More concretely:

- f^{ODCI} cannot be represented using f^{ODCO} or f^{OEC} because the latter ones lack conditioning on the PI values. There can be multiple PI value combinations leading to the same PO value, but only some of them are don't cares.
- The example in Fig. 1 cannot be represented using f^{ODCI} or f^{ODCO} because the condition is not simply ignoring the value of a single PO, but flipping the values of both POs.

It is possible to convert f^{ODCO} into f^{OEC}, but the conversion is not straightforward, nor efficient. Starting from $f^{\text{OEC}}(\mathbf{a}_O, \mathbf{b}_O) = \mathbf{a}_O \leftrightarrow \mathbf{b}_O$, for each $\mathbf{b}_O \in \mathbb{B}^{|O|}$ such that $f_o^{\text{ODCO}}(\mathbf{b}_O) = 1$, we make $f^{\text{OEC}}(\mathbf{b}_O, \mathbf{b}_O^*) = 1$, where $\mathbf{b}_O^*$ is derived by flipping the value corresponding to o in $\mathbf{b}_O$. The complication comes from propagating the equivalence and keeping the transitivity property of the equivalence relation during the process.

As discussed in Sect. 4.3, the specification of a logic synthesis problem can be seen as a Boolean relation. In the presence of external don't-care conditions, representation using Boolean relations is inevitable because there are more than one compatible completely specified multi-output Boolean functions. To represent the *specification Boolean relation* $\mathcal{R}_{\text{spec}}$, we write its characteristic function, called the *specification function* $f^{\text{spec}} : \mathbb{B}^{|I|+|O|} \to \mathbb{B}$, which asks if a certain pair of PI and PO minterms is in $\mathcal{R}_{\text{spec}}$:

$$\begin{aligned} f^{\text{spec}}(\mathbf{b}_I, \mathbf{b}_O) = 1 &\iff \text{Under } \mathbf{b}_I, \mathbf{b}_O \text{ is acceptable at POs.} \\ &\iff (\mathbf{b}_I, \mathbf{b}_O) \in \mathcal{R}_{\text{spec}} \end{aligned} \tag{16}$$

Given an original network N_{ori}, computing the function f_O^I, and the external don't-care conditions as any subset of representations discussed in this section, f^{spec} may be derived:

$$\begin{aligned} f^{\text{spec}}(\mathbf{b}_I, \mathbf{b}_O) = & f^{\text{CDC}}(\mathbf{b}_I) \\ & \vee \bigwedge_{o \in O} \left(f_o^{\text{ODCI}}(\mathbf{b}_I) \vee f_o^{\text{ODCO}}(\mathbf{b}_O) \right) \\ & \vee f^{\text{OEC}}(f_O^I(\mathbf{b}_I), \mathbf{b}_O). \end{aligned} \tag{17}$$

In Eq. (17), if f^{CDC}, f^{ODCI}, or f^{ODCO} are not given, they are substituted with 0 (i.e., the term is removed); if f^{OEC} is not given, it is substituted with a negated miter function $\neg \bigvee_{o \in O} \left(f_o^I(\mathbf{b}_I) \oplus \mathbf{b}_O \right)$.

A network is *compatible* if its global PO function f^{impl} fulfills:

$$\forall \mathbf{b} \in \mathbb{B}^{|I|},\ f^{\text{spec}}(\mathbf{b}, f^{\text{impl}}(\mathbf{b})) = 1. \tag{18}$$

After logic optimization, a verification step is usually done to ensure the functional correctness of the optimized circuit. Classical CEC verifies if the optimized circuit computes exactly the same global PO function as the original circuit. However,

when optimization is performed with external don't cares, such exact equivalence requirement is too strong. Verification must be modified to use a network representing f^{spec} instead of a miter network.

5 Optimization with External Don't Cares

To utilize both internal and external don't-care conditions, a *Boolean method*, which considers Boolean functions of the nodes instead of analyzing the network as algebraic expressions (i.e., an *algebraic method*), must be used. As it is computationally too hard to synthesize (or resynthesize) the entire network from a Boolean function or Boolean relation, modern Boolean methods often perform resynthesis and substitution locally within a smaller region, called a *window*.

However, in order to leverage the flexibilities provided by external don't cares, these conditions must be propagated from the boundaries of the network inward to the windows being resynthesized. For this purpose, we propose to adopt the simulation-guided paradigm [7]. In this paradigm, node functions are approximated by their *simulation signatures*, obtained by performing global simulations using a non-exhaustive set of *simulation patterns* (value assignments to primary inputs). An optimization flow adopting the simulation-guided paradigm consists of the following key steps:

1. Generate a set of simulation patterns.
2. Simulate the network to obtain simulation signatures and use the signatures to compute optimization candidates. The resynthesis computation can be done in a window of any size. Optionally, ODCs may be computed by re-simulating the transitive fanout cone, similar to the method described in Sect. 3.
3. As the simulation is not exhaustive, a candidate needs to be formally verified before it can be substituted into the network. This is done by solving a SAT instance converted from the network. If a satisfiable assignment is derived by the SAT solver, it is a counterexample proving that the candidate produces unwanted output under a certain PI assignment. The counterexample is added into the simulation patterns. Otherwise, an unsatisfiable result proves that the candidate is valid and thus it is used to substitute the original sub-network.

Using global simulation, internal SDCs are accumulated and propagated within the network as missing bit patterns in the simulation signatures. EXCDCs can be easily integrated by removing simulation patterns that are don't cares in Step 1. In contrast, EXODCs may only be used when ODC computation is enabled in Step 2 and is considered until primary outputs. In such case, ODC computation is modified as follows: To compute ODCs of a node n, two sets S and S^* of PO simulation signatures are obtained, one (S) by normal simulation and the other (S^*) by adding an inverter at the output of n. For each bit in the simulation signatures (corresponding to a PI simulation pattern), instead of checking if all POs have the same value in S and in S^*, we check if the PO value combination in S^* is in the

Boolean relation $\mathcal{R}_{\text{spec}}$. The SAT instance in Step 3 also needs to be relaxed to take external don't cares into account. The modified SAT instance now encodes the complement of Eq. (18) instead of a miter. A satisfiable assignment to the instance is a counterexample violating the Boolean relation $\mathcal{R}_{\text{spec}}$.

6 Experimental Demonstration

To demonstrate the effectiveness of considering external don't cares in logic synthesis, we present some experimental results in this section. As external don't cares are not provided along with commonly used benchmarks, we have to generate them by ourselves. The algorithm presented in Sect. 5 is implemented in the open-source C++ logic synthesis library *mockturtle*[2] [16].

We select *ten* medium-sized (comparing to other benchmarks in the same suite) benchmarks from the IWLS'22 programming contest.[3] These benchmarks are originally provided as truth tables of PO functions in terms of PIs (i.e., completely specified functions). In this experiment, we use the best (smallest in terms of the number of gates) synthesized AIGs we have obtained in participation of the contest as the starting point. Without external don't cares, they cannot be optimized any further using the highest-effort (using the entire network as windows, considering internal ODCs until POs, and no limitation on the size of dependency circuits) simulation-guided resubstitution [7].

Table 1 summarizes the optimization results using randomly generated external don't cares. All of the 10 benchmarks have 12 PIs and 3 POs. Column #Gates lists

Table 1 Optimization results of using randomly generated external don't cares on highly optimized benchmarks

Benchmark				EXCDC			EXODC			Both		
Name	#PIs	#POs	#Gates	Δ	%	Time	Δ	%	Time	Δ	%	Time
ex70	12	3	263	15	5.70	0.24	0	0.00	0.27	15	5.70	0.35
ex71	12	3	369	2	0.54	0.70	13	3.52	0.75	13	3.52	0.70
ex72	12	3	456	83	18.20	2.03	38	8.33	1.80	35	7.68	2.13
ex73	12	3	208	1	0.48	0.36	1	0.48	0.28	1	0.48	0.24
ex74	12	3	468	40	8.55	3.78	0	0.00	3.78	37	7.91	3.78
ex75	12	3	489	78	15.95	1.43	114	23.31	1.20	132	26.99	1.03
ex76	12	3	246	2	0.81	0.22	1	0.41	0.24	4	1.63	0.27
ex77	12	3	319	89	27.90	0.37	25	7.84	0.32	98	30.72	0.29
ex78	12	3	369	42	11.38	0.36	56	15.18	0.35	52	14.09	0.35
ex79	12	3	365	0	0.00	0.92	20	5.48	0.70	17	4.66	0.78

[2] Available: https://github.com/lsils/mockturtle.

[3] https://www.iwls.org/iwls2022/.

the number of gates before optimization using EXDCs, columns Δ list the reduction on the number of gates after optimization, columns % list the reduction percentage, and columns Time list the runtime in seconds. All benchmarks use the same external don't-care conditions. Column EXCDC is optimized providing only a randomly generated f^{CDC} having 248 minterms evaluating to 1, column EXODC is optimized providing only $f^{\text{ODCO}} = (f_{y_1}^{\text{ODCO}} = 0, f_{y_2}^{\text{ODCO}} = \neg y_1, f_{y_3}^{\text{ODCO}} = 0)$, and column Both is optimized with both f^{CDC} and f^{ODCO}.

This experiment shows that providing external don't cares indeed enables further optimization opportunities, and that the presented optimization technique works in practice.

7 Conclusion and Future Work

This paper aims primarily at raising and defining the problem of logic synthesis with external don't cares. It provides a review on the theoretical definition of don't-care conditions in general, and identifies different ways of representing external don't cares. An emphasis is made on the relation of don't cares and Boolean relations. Finally, using partial simulation and SAT-based verification, we present how external don't cares may be considered in logic optimization. In conclusion, this paper is the first step toward involving external don't cares in logic synthesis. While the theoretical formulations serve as a foundation for future research, the optimization technique is still limited in achievable optimization quality and scalability. In the following, we discuss some future research directions.

7.1 Multi-Target Resynthesis

From the Boolean relation point of view, the classical definition of internal ODCs (Eq. 7) is additionally restricted to pairs of elements that only differ in one bit (corresponding to the node under consideration) instead of any pair that map to the same next-stage minterm. The advantage of this approach is that the don't-care conditions are used to optimize one node at a time without the need to modify the other nodes. However, it is possible to generalize this class of don't cares by grouping all elements that map to the same element in the next-stage Boolean space together as an OEC and drop the dependency of the definition on a certain node. In this case, multiple nodes need to be optimized together and change their output values.

It is shown in [8] that considering the resynthesis problem of multiple nodes at the same time is necessary for some optimization opportunities to emerge, and the work provides algorithms to describe internal DCs as Boolean relations and to resynthesize windows from Boolean relations. The problem of multi-target

resynthesis specified by a Boolean relation is intrinsically more complex than the well-researched single-target resynthesis [6, 13]. While [1] discusses Boolean relation solving based on divide and conquer, further investigation still has potential. With such Boolean relation solver available, logic optimization with external don't cares can be further enhanced.

7.2 Propagation and Management of Observability Equivalence Classes

The biggest problem encountered in the utilization of external don't cares is to properly and efficiently propagate these conditions into the network. Propagation of EXCDCs by partial simulation is relatively straightforward without scalability concern. In contrast, propagation of external ODCs as presented in Sect. 5 is not scalable. On the one hand, computation of ODCs involves re-simulating the entire transitive fanout cone of the node, and verification with EXODCs requires duplicating at least the transitive fanout cone, if not the entire network, in the SAT instance. One possibility to address this issue is to develop methods to propagate external OECs into a cut in the network. On the other hand, management of the OECs is not scalable with respect to the number of POs if PO minterms are explicitly represented. Thus, symbolic representations of OECs and their management methods (especially, merging equivalence classes according to the transitivity rule) need to be developed.

References

1. Bañeres, D., Cortadella, J., Kishinevsky, M.: A recursive paradigm to solve Boolean relations. IEEE Trans. Comput. **58**(4), 512–527 (2009)
2. Bartlett, K.A., Brayton, R.K., Hachtel, G.D., Jacoby, R.M., Morrison, C.R., Rudell, R.L., Sangiovanni-Vincentelli, A.L., Wang, A.R.: Multi-level logic minimization using implicit don't cares. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **7**(6), 723–740 (1988)
3. Brand, D.: Redundancy and don't cares in logic synthesis. IEEE Trans. Comput. **32**(10), 947–952 (1983)
4. Damiani, M., Micheli, G.D.: Don't care set specifications in combinational and synchronous logic circuits. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **12**(3), 365–388 (1993)
5. Goldberg, E.I., Prasad, M.R., Brayton, R.K.: Using SAT for combinational equivalence checking. In: W. Nebel, A. Jerraya (eds.) Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2001, Munich, Germany, March 12–16, 2001, pp. 114–121. IEEE Computer Society (2001)
6. Lee, S.Y., Riener, H., Micheli, G.D.: Logic resynthesis of majority-based circuits by top-down decomposition. In: M. Shafique, A. Steininger, L. Sekanina, M. Krstic, G. Stojanovic, V. Mrazek (eds.) 24th International Symposium on Design and Diagnostics of Electronic Circuits & Systems, DDECS 2021, Vienna, Austria, April 7–9, 2021, pp. 105–110. IEEE (2021)

7. Lee, S.Y., Riener, H., Mishchenko, A., Brayton, R.K., De Micheli, G.: A simulation-guided paradigm for logic synthesis and verification. IEEE Trans. CAD Integr. Circuits Syst. (2021). https://doi.org/10.1109/TCAD.2021.3108704
8. Lee, T.Y., Wu, C.C., Lin, C.C., Chen, Y.C., Wang, C.Y.: Logic optimization with considering Boolean relations. In: J. Madsen, A.K. Coskun (eds.) 2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19–23, 2018, pp. 761–766. IEEE (2018)
9. McGeer, P.C., Brayton, R.K.: The observability don't-care set and its approximations. In: Proceedings of the 1990 IEEE International Conference on Computer Design: VLSI in Computers and Processors, ICCD 1990, Cambridge, MA, USA, 17–19 September, 1990, pp. 45–48. IEEE Computer Society (1990)
10. Mishchenko, A., Brayton, R.K.: SAT-based complete don't-care computation for network optimization. In: 2005 Design, Automation and Test in Europe Conference and Exposition (DATE 2005), 7–11 March 2005, Munich, Germany, pp. 412–417. IEEE Computer Society (2005)
11. Muroga, S., Kambayashi, Y., Lai, H.C., Culliney, J.N.: The transduction method-design of logic networks based on permissible functions. IEEE Trans. Comput. **38**(10), 1404–1424 (1989)
12. Rho, J.K., Somenzi, F.: Don't care sequences and the optimization of interacting finite state machines. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **13**(7), 865–874 (1994)
13. Riener, H., Lee, S.Y., Mishchenko, A., Micheli, G.D.: Boolean rewriting strikes back: Reconvergence-driven windowing meets resynthesis. In: 27th Asia and South Pacific Design Automation Conference, ASP-DAC 2022, Taipei, Taiwan, January 17–20, 2022, pp. 395–402. IEEE (2022)
14. Saluja, N., Khatri, S.P.: A robust algorithm for approximate compatible observability don't care (CODC) computation. In: S. Malik, L. Fix, A.B. Kahng (eds.) Proceedings of the 41th Design Automation Conference, DAC 2004, San Diego, CA, USA, June 7–11, 2004, pp. 422–427. ACM (2004)
15. Savoj, H., Brayton, R.K.: The use of observability and external don't cares for the simplification of multi-level networks. In: R.C. Smith (ed.) Proceedings of the 27th ACM/IEEE Design Automation Conference, Orlando, Florida, USA, June 24–28, 1990, pp. 297–301. IEEE Computer Society Press (1990)
16. Soeken, M., Riener, H., Haaswijk, W., Testa, E., Schmitt, B., Meuli, G., Mozafari, F., Lee, S.Y., Tempia Calvino, A., Marakkalage, D.S., De Micheli, G.: The EPFL logic synthesis libraries. Preprint (2022). arXiv:1805.05121

Maiorana-McFarland Boolean Bent Functions Characterized by Their Reed-Muller Spectra

Claudio Moraga, Radomir S. Stanković, and Milena Stanković

1 Introduction

Bent functions were introduced by Oscar Rothaus in 1976 [16]. These functions are at the largest Hamming distance $(2^{n-1} - 2^{(n/2)-1})$ from affine Boolean functions, however restricted to an even number of variables. Due to the high nonlinearity, these functions attracted the interest of researchers particularly in coding theory and in cryptography. It is simple to understand that also due to the high nonlinearity, the number of such functions is reduced, increasing however, super exponentially. There are 8 bent functions on 2 variables, 896 bent functions on 4 variables, and, as it will be shown below, there are $1.37 \cdot 10^{18}$ Maiorana-McFarland Boolean bent functions on 8 variables out of almost $1.158 \cdot 10^{77}$ Boolean functions on 8 variables. (See, [6], for the exact number of Boolean bent functions on 8 variables.)

In his original paper, Rothaus introduced a simple method to generate a class of bent functions from $\mathbb{F}_2^{2k}$ to $\mathbb{F}_2$:

$$f(x, y) = x \cdot y + g(y),$$

where g is an arbitrary Boolean function from $\mathbb{F}_2^k$ to $\mathbb{F}_2$. This method was strongly improved by J. A. Maiorana [8] as follows:

C. Moraga (✉)
Faculty of Computer Science, TU Dortmund University, Dortmund, Germany
e-mail: claudio.moraga@udo.edu; claudio.moraga@tu-dortmund.de

R. S. Stanković
Mathematical Institute, Serbian Academy of Sciences and Arts and Department of Computer Science, Faculty of Electronic Engineering, Belgrade, Serbia

M. Stanković
Department of Computer Science, Faculty of Electronic Engineering, Niš, Serbia

R. Drechsler, S. Huhn (eds.), *Advanced Boolean Techniques*,
https://doi.org/10.1007/978-3-031-28916-3_4

$$f(x, y) = x \cdot (\pi(y)) + g(y), \tag{1}$$

where π denotes a permutation of $\mathbb{F}_2^k$.

Independently of Maiorana's work, L. R. McFarland developed the same method [9]. For this reason, this class of functions is known as the Maiorana-McFarland class. A first analysis of the Maiorana-McFarland generation method was disclosed in [2]. Later on, many other classes have been developed. (See, e.g., [1, 2, 10, 18, 20].)

The Transform, which today is known as the Reed-Muller transform, was originally developed in 1928 by I. L. Zhegalkin [21, 22], (in Russian). Due to the language barrier, this transform remained unknown and was "rediscovered" only in 1954 by S. M. Reed [15] and D. E. Muller [13]. The basic Reed-Muller transform, which here we call $\mathbf{RM}(1)$, has the matrix representation $\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$, and the transform has a Kronecker product structure, i.e. $\mathbf{RM}(n) = \left(\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}\right)^{\otimes n}$, the n-fold Kronecker product of $\mathbf{RM}(1)$ with itself. The Reed-Muller spectrum of a Boolean n-place function f with value vector $\mathbf{F}$ is obtained as $\mathbf{RM}(n) \cdot \mathbf{F}$, with computations done modulo 2. With respect to bent functions, it is interesting to mention that the Reed-Muller spectrum supports the concept of the degree of bent functions. Recall that the degree of a binary bent function in n variables is at most $n/2$ [1], meaning that in the Reed-Muller functional expression of a bent function, no product with more than $n/2$ variables should appear. This observation is important, since it can be used to reduce the search space in determining bent functions [14].

Applications of this transform in different areas have received different names. In this paper, related functional expressions are called Positive Polarity Reed-Muller expressions. Mathematicians would rather speak of Zhegalkin polynomials, whereas people working in cryptography refer to these functional expressions as algebraic normal forms. (See, e.g., [10, 20].) As a closing remark, it may be mentioned that the Reed-Muller transform belongs to the family of spectral techniques for signal processing. See, e.g., [17].

2 Formalisms

Let $n = 2k$, $k \in \mathbb{N}$. The following notation will be used when explicit dimensions are needed: $\mathbf{Y}(k)$ denotes a $(2^k \times 2^k)$ matrix or a vector of length 2^k.

A simple method to generate n-place Boolean bent functions of the Maiorana-McFarland class, based on Eq. (1), is the following, adapted from [11, 12]:

$$\begin{aligned} \mathbf{F} &= vec(\mathbf{M}^{[k]} \cdot \mathbf{P}(k) \oplus [1, \ldots, 1]^T \otimes \mathbf{G}) \\ &= vec(\mathbf{M}^{[k]} \cdot \mathbf{P}(k)) \oplus vec([1, \ldots, 1]^T \otimes \mathbf{G}), \end{aligned} \tag{2}$$

where $\mathbf{F}$ denotes the value vector of an n-place Boolean function, vec is a vectorizing operation, which when applied to a matrix, concatenates the columns of that matrix to build a column vector [3]. $\mathbf{M} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$, and $\mathbf{M}^{[k]}$ indicates the k-fold tensor sum[1] of $\mathbf{M}$ with itself. Notice that the first row of $\mathbf{M}^{[k]}$ will always be a 0 row. $\mathbf{P}(k)$ stands for a $(2^k \times 2^k)$ permutation matrix, $[1, 1, \ldots, 1]$ is a row vector of length 2^k, and $\mathbf{G}$ represents the value vector of an arbitrary Boolean function on k variables. $\mathbf{G}$ will be a row vector. Unless otherwise specified, in what follows, vectors will be column vectors.

Let $\mathbf{U}(q)$ be a $(2^q \times 2^q)$ square matrix all of whose entries are 1. Then, [3, 7, 11]

$$\mathbf{M}^{[2k]} = \mathbf{M}^{[k]} \otimes \mathbf{U}(k) \oplus \mathbf{U}(k) \otimes \mathbf{M}^{[k]}. \tag{3}$$

Thus,

$$\mathbf{M}^{[2]} = \mathbf{M} \otimes \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \oplus \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \otimes \mathbf{M} \bmod 2 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}. \tag{4}$$

Proof of the bentness of $\mathbf{F}$ generated with (2):

Recall that if an n-place Boolean function is bent, then all coefficients of its Walsh spectrum have the absolute value $2^{n/2}$ [4].

Let $\mathbf{S}_F$ denote the Walsh spectrum of an n-place Boolean function f with value vector $\mathbf{F}$. Then,

$$\mathbf{S}_F(\omega) = \mathbf{W}(n) \cdot (-1)^{f(x)}, \quad \forall \omega, x \in F_2^n, \quad \text{where} \quad \mathbf{W}(n) = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}^{\otimes n} \tag{5}$$

and the superindex $\otimes n$ indicates the n-fold Kronecker product of the matrix with itself.

The Spectrum of (2) will be

$$\begin{aligned} \mathbf{S}_F &= \left(\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}^{\otimes n} \cdot (-1)^{vec(\mathbf{M}^{[k]} \cdot \mathbf{P}(k) \oplus ([1,1,\ldots,1]^T \otimes \mathbf{G}))} \right) \\ &= \left(\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}^{\otimes n} \cdot vec(-1)^{(\mathbf{M}^{[k]} \cdot \mathbf{P}(k) \oplus ([1,1,\ldots,1]^T \otimes \mathbf{G}))} \right). \end{aligned} \tag{6}$$

[1] The tensor sum of two matrices has the same structure of the Kronecker product of the matrices, except that all basic operations are sums modulo 2 [3, 7]. For this reason the tensor sum is also known as Kronecker sum.

Definition 1 Let $a_{11}, \ldots a_{rr} \in \{0, 1\}$; $(-1)^{[a_{11} \ldots a_{rr}]} := [(-1)^{a_{11}} \cdots (-1)^{a_{rr}}]$; and if

$$\mathbf{A} = \begin{bmatrix} a_{11} & \cdots & a_{1r} \\ \vdots & \ddots & \vdots \\ a_{r1} & \cdots & a_{rr} \end{bmatrix}, \quad \text{then} \quad (-1)^{\mathbf{A}} := \begin{bmatrix} (-1)^{a_{11}} & \cdots & (-1)^{a_{1r}} \\ \vdots & \ddots & \vdots \\ (-1)^{a_{r1}} & \cdots & (-1)^{a_{rr}} \end{bmatrix}. \tag{7}$$

Moreover, if $\mathbf{B}$ is a matrix of the same dimensions as $\mathbf{A}$, then

$$(-1)^{\mathbf{A} \oplus \mathbf{B}} = (-1)^{\mathbf{A}} \# (-1)^{\mathbf{B}}, \tag{8}$$

where # denotes the Hadamard product of matrices [3]. Furthermore, if $\mathbf{A}$ and $\mathbf{B}$ are square matrices of dimensions $(r \times r)$, and $\mathbf{B}$ has the particular structure that all rows are identical, then $\mathbf{A}\#\mathbf{B} = \mathbf{A} \cdot diag(b_1, b_2, \ldots, b_r)$, where $(b_1, b_2, \ldots, b_r)$ are entries of a row of $\mathbf{B}$ [12]. □

Let $Delta = diag((-1)^{g_1}, (-1)^{g_2}, \ldots, (-1)^{g_{2^k}})$. Then with (7) and (8) in (6):

$$\begin{aligned} \mathbf{S}_F &= \left(\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}^{\otimes n} \cdot vec\langle (-1)^{(\mathbf{M}^{[k]} \cdot \mathbf{P}(k))} \# (-1)^{([1,1,\ldots,1]^T \otimes \mathbf{G})} \rangle \right) \\ &= \left(\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}^{\otimes n} \cdot vec\langle (-1)^{(\mathbf{M}^{[k]} \cdot \mathbf{P}(k))} \cdot Delta \rangle \right). \end{aligned} \tag{9}$$

Notice that $(-1)^{(\mathbf{M}^{[k]} \cdot \mathbf{P}(k))} = (-1)^{\mathbf{M}^{[k]}} \cdot \mathbf{P}(k)$. Moreover, $(-1)^{\mathbf{M}}$ equals the (2×2) matrix $\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \mathbf{W}(1)$. Then, $(-1)^{\mathbf{M}^{[k]}} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}^{\otimes k} = \mathbf{W}(k)$.

Moreover, recall that $\mathbf{W}(k) \cdot \mathbf{W}(k) = 2^k \cdot \mathbf{I}(k)$, where $\mathbf{I}(k)$ denotes the identity matrix. With Lemma 4.3.1 of [3], Eq. (9) becomes:

$$\begin{aligned} \mathbf{S}_F &= vec \left(\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}^{\otimes k} \cdot \left[(-1)^{\mathbf{M}^{[k]}} \cdot \mathbf{P}(k) \cdot Delta \right] \cdot \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}^{\otimes k} \right) \\ &= vec \left(\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}^{\otimes k} \cdot \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}^{\otimes k} \cdot \mathbf{P}(k) \cdot Delta \cdot \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}^{\otimes k} \right) \\ &= vec \left\langle 2^k \cdot \mathbf{I}(k) \cdot \mathbf{P}(k) \cdot Delta \cdot \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}^{\otimes k} \right\rangle. \end{aligned} \tag{10}$$

Notice that the diagonal elements of $Delta$ belong to $\{-1, 1\}$ as well as all entries of $\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}^{\otimes k}$. The product of them and the permutation of the resulting

rows, induced by $\mathbf{P}(k)$, will preserve this structural property. All entries will have magnitude 1. Therefore, the absolute value of all the coefficients of $\mathbf{S}_F$ will be $2^k = 2^{n/2}$. The spectrum is flat, the entries have the correct magnitude, and hence the (Maiorana-McFarland) Boolean function f generated with (2) is indeed bent as it should be proven. □

If in Eq. (2) $k = 4, n = 2k = 8, 2^k = 16$, then there are 16! permutation matrices $\mathbf{P}(4)$, and there are $2^{(2^4)} = 2^{16} = 65.536$ Boolean functions $\mathbf{G}$ on 4 variables, whose value vectors have length 16. Hence, there are $16! \cdot 2^{16} \approx 1.37 \cdot 10^{18}$ Maiorana-McFarland Boolean bent functions on 8 variables (out of $\approx 99.27 \cdot 10^{30}$ Boolean bent functions on 8 variables [6] and $2^{(2^8)} = 2^{256} \approx 1.16 \cdot 10^{77}$ general Boolean functions on 8 variables).

It is simple to see that Eq. (2) provides the value vector of Maiorana-McFarland Boolean bent functions, depending on $\mathbf{P}(k)$ and $\mathbf{G}$. To obtain their functional expressions, the following Lemma of the Reed-Muller transform theory will be used:

Lemma 1 ([4, 15]) *Given the value vector* $\mathbf{F}$ *of a Boolean function, its functional Positive Polarity Reed-Muller polynomial expression* f *may be obtained as the inner product of the corresponding Reed-Muller basis and the Reed-Muller spectrum of* $\mathbf{F}$. □

If $k = 2, n = 4, 2^k = 4$, the Reed-Muller basis $\mathbf{B}(4)$, of length 2^4, is calculated as

$$\mathbf{B}(4) = \left(\left[\begin{array}{cc} 1 & x_1 \end{array}\right] \otimes \left[\begin{array}{cc} 1 & x_2 \end{array}\right] \otimes \left[\begin{array}{cc} 1 & x_3 \end{array}\right] \otimes \left[\begin{array}{cc} 1 & x_4 \end{array}\right]\right)^T.$$

After computing the Kronecker product, we get:

$$\begin{aligned}\mathbf{B}(4) = [&1, x_4, x_3, x_3x_4, x_2, x_2x_4, x_2x_3, x_2x_3x_4, x_1, x_1x_4,\\ &x_1x_3, x_1x_3x_4, x_1x_2, x_1x_2x_4, x_1x_2x_3, x_1x_2x_3x_4]^T\end{aligned}$$

or in another way of writing

$$\mathbf{B}(4) = vec\begin{bmatrix} 1 & x_2 & x_1 & x_1x_2 \\ x_4 & x_2x_4 & x_1x_4 & x_1x_2x_4 \\ x_3 & x_2x_3 & x_1x_3 & x_1x_2x_3 \\ x_3x_4 & x_2x_3x_4 & x_1x_3x_4 & x_1x_2x_3x_4 \end{bmatrix}. \tag{11}$$

Theorem 1 *Given a Maiorana-McFarland Boolean bent function generated with (2) for a fixed k, then* $\mathbf{P}(k)$ *and* $\mathbf{G}$ *make unique disjoint contributions to the RM spectrum of the function.* □

This will be illustrated with the experimental results below. The proof will be presented after the Cases.

(i) Let $k = 2, n = 4, 2^k = 4$.
Permutations will be coded as $[n_1, n_2, n_3, n_4]$, where $1 \leq i \neq j \leq 4, n_i \neq n_j$, and with $n_0 := 0$, $(n_{i-1}) \in \mathbb{Z}_4$ without repetitions. The position of n_i (i.e., i) indicates the column and the value of n_i the row where the permutation matrix has the entry 1. All other entries of the same row and column have obviously the value 0. Calculations will follow the second expression of (2) and will be done in $GF(2)$.

The required Reed-Muller transform matrices are

$$\mathbf{RM}(1) = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}, \quad \mathbf{RM}(2) = \mathbf{RM}(1) \otimes \mathbf{RM}(1) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}. \quad (12)$$

Case 1 Let $\mathbf{P}(2) = [4, 3, 2, 1]$ and $\mathbf{G}_1 = [1, 0, 0, 0]$. Then,

$$\mathbf{RM}(4) \cdot \mathbf{F}_1 = \mathbf{RM}(4)\langle vec(\mathbf{M}^{[2]} \cdot \mathbf{P}(2)) \oplus vec([1, 1, 1, 1]^T \otimes \mathbf{G}_1)\rangle \bmod 2.$$

With Lemma 4.3.1 of [3]

$$\begin{aligned} \mathbf{RM}(4) \cdot \mathbf{F}_1 &= vec\langle \mathbf{RM}(2) \cdot (\mathbf{M}^{[2]} \cdot \mathbf{P}(2)) \cdot \mathbf{RM}(2)^T \rangle \\ &\quad \oplus vec\langle \mathbf{RM}(2) \cdot ([1, 1, 1, 1]^T \otimes \mathbf{G}_1) \cdot \mathbf{RM}(2)^T \rangle \bmod 2. \\ &= vec\left\langle \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \right\rangle \\ &\quad \oplus vec\left\langle \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \right\rangle \\ &= vec\left\langle \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \oplus \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \right\rangle. \end{aligned}$$

With Lemma 1 and $\mathbf{B}(4)$

$$f(x_1, x_2, x_3, x_4) = \langle x_4 \oplus x_2x_4 \oplus x_3 \oplus x_1x_3 \rangle \oplus \langle 1 \oplus x_1 \oplus x_2 \oplus x_1x_2 \rangle.$$

Case 2 Let $\mathbf{P}(2) = [4, 3, 2, 1]$ and $\mathbf{G}_2 = [1, 0, 1, 1]$: The same permutation as in Case 1, but a new **G**-function:

$$\mathbf{RM}(4) \cdot \mathbf{F}_1 = \mathbf{RM}(4)\langle vec(\mathbf{M}^{[2]} \cdot \mathbf{P}(2)) \oplus vec([1, 1, 1, 1]^T \otimes \mathbf{G}(2)\rangle$$

$$= vec\left\langle \begin{bmatrix} 0&0&0&0 \\ 1&1&0&0 \\ 1&0&1&0 \\ 0&0&0&0 \end{bmatrix} \oplus \begin{bmatrix} 1&1&0&1 \\ 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \end{bmatrix} \right\rangle,$$

leading to

$$f(x_1, x_2, x_3, x_4) = \langle x_4 \oplus x_2x_4 \oplus x_3 \oplus x_1x_3\rangle \oplus \langle 1 \oplus x_2 \oplus x_1x_2\rangle.$$

Case 3 $\mathbf{P}(2) = [2, 1, 4, 3]$, $\mathbf{G}_1 = [1, 0, 0, 0]$: A new permutation, but the same **G**-function as in Case 1.

$$\mathbf{RM}(4) \cdot \mathbf{F}_3 = \mathbf{RM}(4)\langle vec(\mathbf{M}^{[2]} \cdot \mathbf{P}(2)) \oplus vec([1, 1, 1, 1]^T \otimes \mathbf{G}_1)\rangle$$

$$= vec\left\langle \begin{bmatrix} 0&0&0&0 \\ 1&1&0&0 \\ 0&0&1&0 \\ 0&0&0&0 \end{bmatrix} \oplus \begin{bmatrix} 1&1&1&1 \\ 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \end{bmatrix} \right\rangle,$$

i.e.,

$$f(x_1, x_2, x_3, x_4) = \langle x_4 \oplus x_2x_4 \oplus x_1x_3\rangle \oplus \langle 1 \oplus x_1 \oplus x_2 \oplus x_1x_2\rangle.$$

Case 4 $\mathbf{P}(2) = [4, 1, 3, 2]$, $G = [1, 0, 1, 0]$. A new permutation and a new **G**-function.

$$\mathbf{RM}(4) \cdot \mathbf{F}_4 = \mathbf{RM}(4)\langle vec(\mathbf{M}^{[2]} \cdot \mathbf{P}(2)) \oplus vec([1, 1, 1, 1]^T \otimes [1, 0, 1, 0])\rangle$$

$$vec\left\langle \begin{bmatrix} 0&0&0&0 \\ 1&0&1&0 \\ 0&1&1&0 \\ 0&0&0&0 \end{bmatrix} \oplus \begin{bmatrix} 1&1&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \end{bmatrix} \right\rangle,$$

from where

$$f(x_1, x_2, x_3, x_4) = \langle x_4 \oplus x_1x_4 \oplus x_2x_3 \oplus x_1x_3\rangle \oplus \langle 1 \oplus x_2\rangle.$$

The four cases illustrate that the Reed-Muller spectrum of **F** may be obtained by vectorizing two disjoint matrices. (Matrices are called "disjoint," in the case that if the Hamming weight of a row of a matrix is larger than 0, then the corresponding

row in the other matrix is a 0 row.) One of the matrices depends on the permutation $\mathbf{P}(k)$ used in (2), and the other one depends on the function $\mathbf{G}$ of (2). In all cases the first and fourth rows of the first matrix are 0 rows, whereas the second matrix is characterized by unique non-0 row in the first position.

Proof In what follows, by calculating step by step, we prove that the results of the former cases are not "biased coincidences."

$$\mathbf{RM}(4) \cdot \mathbf{F} = vec\langle\{\mathbf{RM}(2) \cdot \mathbf{M}^{[2]}\} \cdot \mathbf{P}(2) \cdot \mathbf{RM}(2)^T\rangle \\ \oplus vec\langle\mathbf{RM}(2) \cdot ([1,1,1,1]^T \otimes \mathbf{G}) \cdot \mathbf{RM}(2)^T\rangle \bmod 2.$$

First part, without including the permutation:

$$\{\mathbf{RM}(2) \cdot \mathbf{M}^{[2]}\} \bmod 2 = \begin{bmatrix} 1\,0\,0\,0 \\ 1\,1\,0\,0 \\ 1\,0\,1\,0 \\ 1\,1\,1\,1 \end{bmatrix} \cdot \begin{bmatrix} 0\,0\,0\,0 \\ 0\,1\,0\,1 \\ 0\,0\,1\,1 \\ 0\,1\,1\,0 \end{bmatrix} = \begin{bmatrix} 0\,0\,0\,0 \\ 0\,1\,0\,1 \\ 0\,0\,1\,1 \\ 0\,0\,0\,0 \end{bmatrix}. \tag{13}$$

Recall that the first row of $\mathbf{M}$ is a 0 row and the bottom row of $\mathbf{RM}$ is a constant 1 row. Furthermore, the columnwise sum of the entries of $\mathbf{M}$ is congruent with 0 modulo 2. These properties support that already $\mathbf{RM}(2) \cdot \mathbf{M}^{[2]}$ determines the position of the 0 rows of the first component of the (matrix representation of the) RM spectrum, independently of the permutation $\mathbf{P}(k)$.

Second part. The effect of $\mathbf{P}(2)$:

$$\mathbf{RM}(2) \cdot \mathbf{M}^{[2]} \cdot \mathbf{P}(2) \bmod 2 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ \odot & \odot & \odot & \odot \\ \odot & \odot & \odot & \odot \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

Obviously the first and fourth rows will remain 0 rows for any permutation, whereas the entries of the second and third row, marked with "$\odot$," will contribute a Hamming weight larger than 0 for these rows, depending on $\mathbf{P}(2)$, which will reorder the columns.

Finally,

$$\{\mathbf{RM}(2) \cdot \mathbf{M}^{[2]}\} \cdot \mathbf{P} \cdot \mathbf{RM}(2)^T = \begin{bmatrix} 0 & 0 & 0 & 0 \\ \odot & \odot & \odot & \odot \\ \odot & \odot & \odot & \odot \\ 0 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1\,1\,1\,1 \\ 0\,1\,0\,1 \\ 0\,0\,1\,1 \\ 0\,0\,0\,1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ \odot & \mathcal{Q} & \mathcal{Q} & \mathcal{Q} \\ \odot & \mathcal{Q} & \mathcal{Q} & \mathcal{Q} \\ 0 & 0 & 0 & 0 \end{bmatrix},$$

where the $\mathcal{Q}$ entries equal the sum mod 2 of 2 or 4 $\odot$ entries. The corresponding rows have a Hamming weight larger than 0. The first and fourth rows, however, remain 0 rows.

Similarly, with respect to $vec\langle \mathbf{RM}(2) \cdot ([1,1,1,1]^T \otimes \mathbf{G}) \cdot \mathbf{RM}(2)^T \rangle \bmod 2$, where $\mathbf{G} = [g_1, g_2, g_3, g_4]$.

$$\mathbf{RM}(2) \cdot ([1,1,1,1]^T \otimes \mathbf{G}) \bmod 2 = \begin{bmatrix} 1&0&0&0 \\ 1&1&0&0 \\ 1&0&1&0 \\ 1&1&1&1 \end{bmatrix} \cdot \begin{bmatrix} g_1&g_2&g_3&g_4 \\ g_1&g_2&g_3&g_4 \\ g_1&g_2&g_3&g_4 \\ g_1&g_2&g_3&g_4 \end{bmatrix}$$

$$= \begin{bmatrix} g_1&g_2&g_3&g_4 \\ 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \end{bmatrix},$$

and

$$\mathbf{RM}(2) \cdot ([1,1,1,1]^T \otimes \mathbf{G}) \cdot \mathbf{RM}(2)^T \bmod 2$$

$$= \begin{bmatrix} g_1&g_2&g_3&g_4 \\ 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \end{bmatrix} \cdot \begin{bmatrix} 1&1&1&1 \\ 0&1&0&1 \\ 0&0&1&1 \\ 0&0&0&1 \end{bmatrix} = \begin{bmatrix} g_1&g_{12}&g_{13}&g_{1234} \\ 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \end{bmatrix},$$

where $g_{12} = g_1 \oplus g_2$, $g_{13} = g_1 \oplus g_3$ and $g_{1234} = g_1 \oplus g_2 \oplus g_3 \oplus g_4$. Only the first row has entries in $\{0, 1\}$ and a Hamming weight > 0. All other rows are 0 rows.

It becomes clear that the Reed-Muller spectra of $vec(\mathbf{M}^{[2]} \cdot \mathbf{P}(2))$ and of $vec([1,1,1,1]^T \otimes \mathbf{G})$ are disjoint, since there is no overlap of nonzero coefficient rows of both matrix representations of the spectra.

Furthermore, notice that

$$\mathbf{RM}(2) \cdot \mathbf{G}^T = \begin{bmatrix} 1&0&0&0 \\ 1&1&0&0 \\ 1&0&1&0 \\ 1&1&1&1 \end{bmatrix} \cdot \begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ g_4 \end{bmatrix} = \begin{bmatrix} g_1 \\ g_1 \oplus g_2 \\ g_1 \oplus g_3 \\ g_1 \oplus g_2 \oplus g_3 \oplus g_4 \end{bmatrix},$$

which proves the following Lemma.

Lemma 2 *For a given k and $\mathbf{G}$ from (2), the transpose of the RM spectrum of $\mathbf{G}^T$ equals the first row of the (matrix representation of the) RM spectrum of $\mathbf{F}$, where $\mathbf{F}$ is obtained with (2).* □

(*ii*) Let $k = 3$, $n = 6$, $2^k = 8$.
Notice that $k = 2^r$ has no integer solution for r. This will cause an asymmetry in the calculation of $\mathbf{M}^{[k]}$. (See below.)

Permutations will be coded as $[n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8]$, where $\forall 1 \leq i \neq j \leq 8$, $n_i \neq n_j$ and with $n_0 := 0$, $(n_{i-1}) \in \mathbb{Z}_8$ without repetitions. The

position of n_i (i.e., i) indicates the column and the value of n_i the row where the permutation matrix has the entry 1. All other entries of the same row and column have the value 0:

$$\mathbf{RM}(6) = \mathbf{RM}(3) \otimes \mathbf{RM}(3),$$

as usual; however [3],

$$\mathbf{M}^{[3]} = \mathbf{M}^{[2]} \otimes \mathbf{U}(1) \oplus \mathbf{U}(2) \otimes \mathbf{M}.$$

$$\mathbf{M}^{[3]} = \begin{bmatrix} 0\,0\,0\,0 \\ 0\,1\,0\,1 \\ 0\,0\,1\,1 \\ 0\,1\,1\,0 \end{bmatrix} \otimes \begin{bmatrix} 1\,1 \\ 1\,1 \end{bmatrix} \oplus \begin{bmatrix} 1\,1\,1\,1 \\ 1\,1\,1\,1 \\ 1\,1\,1\,1 \\ 1\,1\,1\,1 \end{bmatrix} \otimes \begin{bmatrix} 0\,0 \\ 0\,1 \end{bmatrix}$$

$$= \begin{bmatrix} 0\,0\,0\,0\,0\,0\,0\,0 \\ 0\,1\,0\,1\,0\,1\,0\,1 \\ 0\,0\,1\,1\,0\,0\,1\,1 \\ 0\,1\,1\,0\,0\,1\,1\,0 \\ 0\,0\,0\,0\,1\,1\,1\,1 \\ 0\,1\,0\,1\,1\,0\,1\,0 \\ 0\,0\,1\,1\,1\,1\,0\,0 \\ 0\,1\,1\,0\,1\,0\,0\,1 \end{bmatrix}.$$

A direct calculation shows that in this case, the following also holds, where, in general, the tensor sum does not commute [3, 7, 11]:

$$\mathbf{M}^{[3]} = \mathbf{M} \otimes \mathbf{U}(2) \oplus \mathbf{U}(1) \otimes \mathbf{M}^{[2]}.$$

The Basis $\mathbf{B}(6) = \bigotimes_{i=1}^{6} [1, x_i]^T$ has a legth of $2^6 = 64$ and may be given a compact representation by vectorizing a matrix, whose first column is

$$\left[1,\ x_6,\ x_5,\ x_5x_6,\ x_4,\ x_4x_6,\ x_4x_5,\ x_4x_5x_6 \right]^T,$$

and its first row is

$$\left[1,\ x_3,\ x_2,\ x_2x_3,\ x_1,\ x_1x_3,\ x_1x_2,\ x_1x_2x_3 \right].$$

All other entries are obtained as the product of the corresponding row-column coordinates. For example, a Basis component at the position $\langle 5, 6 \rangle$ equals $x_1x_3x_4$, where the variables are ordered by increasing indices.

Case 5 Let $\mathbf{P}(3) = [6, 8, 7, 5, 2, 4, 3, 1]$, and $\mathbf{G} = [1, 0, 0, 1, 1, 1, 1, 0]$.

$$\begin{aligned}
&\mathbf{RM}(6)(vec\langle \mathbf{M}^{[3]} \cdot \mathbf{P}(3) \oplus ([1,1,1,1,1,1,1,1]^T \otimes \mathbf{G})\rangle) \\
&= vec\langle \mathbf{RM}(3) \cdot (\mathbf{M}^{[3]} \cdot \mathbf{P}(3) \oplus [1,1,1,1,1,1,1,1]^T \otimes \mathbf{G}) \cdot \mathbf{RM}(3)^T \rangle \\
&= vec(\mathbf{RM}(3) \cdot (\mathbf{M}^{[3]} \cdot \mathbf{P}(3)) \cdot \mathbf{RM}(3)^T) \\
&\qquad \oplus \quad vec(\mathbf{RM}(3) \cdot ([1,1,1,1,1,1,1,1]^T \otimes \mathbf{G}) \cdot \mathbf{RM}(3)^T).
\end{aligned}$$

Let

$$\text{Alpha} := \mathbf{RM}(3) \cdot (\mathbf{M}^{[3]} \cdot \mathbf{P}(3)) \cdot \mathbf{RM}(3)^T, \tag{14}$$

and

$$\text{Beta} := \mathbf{RM}(3) \cdot ([1,1,1,1,1,1,1,1]^T \otimes \mathbf{G}) \cdot \mathbf{RM}(3)^T.$$

Direct MatLab calculations give the following results:

$$\begin{aligned}
\text{Alpha: } &Row\ 2 = [1,0,1,0,0,0,0,0] \\
&Row\ 3 = [0,1,1,0,0,0,0,0] \\
&Row\ 5 = [1,0,0,0,1,0,0,0].
\end{aligned}$$

All other rows are 0 rows.

$$\text{Beta: } Row\ 1 = [1,1,1,0,0,1,1,1].$$

All other rows are 0 rows.

It is quite clear that Alpha and Beta are disjoint.

The 1-entries of Alpha and Beta applied on the matrix representation of $\mathbf{B}(6)$ lead to the following Positive Polarity Reed-Muller functional expression:

$$\begin{aligned}
f(x_1, x_2, x_3, x_4, x_5, x_6) &= \langle 1 \oplus x_2 \oplus x_3 \oplus x_1x_2 \oplus x_1x_3 \oplus x_1x_2x_3 \rangle \\
&\quad \oplus \langle x_2x_6 \oplus x_6 \oplus x_2x_5 \oplus x_3x_5 \oplus x_1 \oplus x_1x_4 \rangle.
\end{aligned}$$

Notice that the functional expression is of degree $3 = n/2$, which is the maximum degree allowed for f to be bent [1].

(iii) Let $k = 4, n = 8, 2^k = 16$.
Permutations will be coded as $[n_1, n_2, n_3, n_4, \ldots, n_{13}, n_{14}, n_{15}, n_{16}]$, where $1 \le i \ne j \le 16$, $n_i \ne n_j$ and with $n_0 := 0$, $(n_{i-1}) \in \mathbb{Z}_{16}$ without repetitions. The position of n_i (i.e., i) indicates the column and the value of n_i the row where the permutation matrix has the entry 1. All other entries of the same row and column have the value 0.

$$\mathbf{RM}(8) = \mathbf{RM}(4) \otimes \mathbf{RM}(4) \quad \text{and} \quad \mathbf{M}^{[4]} = \mathbf{M}^{[2]} \otimes \mathbf{U}(2) \oplus \mathbf{U}(2) \otimes \mathbf{M}^{[2]}.$$

Recall that $\mathbf{U}(2)$ denotes a (4×4) matrix all whose entries are 1.

$$\mathbf{B}(8) = [[1, x_1] \otimes [1, x_2] \otimes [1, x_3] \otimes [1, x_4] \otimes [1, x_5] \otimes [1, x_6] \otimes [1, x_7] \otimes [1, x_8]]^T$$

The matrix $\mathbf{B}(8)$, which has a length of $2^8 = 256$, may be expressed by vectorizing a $(2^4 \times 2^4)$ matrix with first column:

$$[1, x_8, x_7, x_7x_8, x_6, x_6x_8, x_6x_7, x_6x_7x_8, x_5, x_5x_8, x_5x_7, x_5x_7x_8, x_5x_6, x_5x_6x_8, x_5x_6x_7, x_5x_6x_7x_8]^T,$$

and first row

$$[1, x_4, x_3, x_3x_4, x_2, x_2x_4, x_2x_3, x_2x_3x_4, x_1, x_1x_4, x_1x_3, x_1x_3x_4, x_1x_2, x_1x_2x_4, x_1x_2x_3, x_1x_2x_3x_4].$$

An entry at the position $\langle j, k \rangle$ equals the product of j-th row entry times the k-th column entry (ordered by increasing indices). For instance, the entry at the position $\langle 4, 7 \rangle$ equals $x_2x_3x_7x_8$.

Case 6 Let $\mathbf{P}(4) = [5, 7, 3, 2, 4, 1, 6, 8, 16, 14, 9, 12, 10, 11, 15, 13]$, and $\mathbf{G} = [1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0]$.

$$\begin{aligned}
&\mathbf{RM}(8)(vec\langle \mathbf{M}^{[4]} \cdot \mathbf{P}(4) \oplus [1, 1, 1, \ldots, 1, 1]^T \cdot \mathbf{G} \rangle \\
&= vec\langle \mathbf{RM}(4) \cdot (\mathbf{M}^{[4]} \cdot \mathbf{P}(4) \oplus [1, 1, 1, \ldots, 1, 1]^T \cdot \mathbf{G}) \cdot \mathbf{RM}(4)^T \rangle \\
&= vec(\mathbf{RM}(4) \cdot (\mathbf{M}^{[4]} \cdot \mathbf{P}(4)) \cdot \mathbf{RM}(4)^T) \\
&\quad \oplus \quad vec(\mathbf{RM}(4) \cdot [1, 1, 1, \ldots, 1, 1]^T \cdot \mathbf{G} \cdot \mathbf{RM}(4)^T).
\end{aligned}$$

As in (15), let the result be expressed as $vec(\text{Alpha}) \oplus vec(\text{Beta})$. Direct MatLab calculations give the following results:

$$\begin{aligned}
\text{Alpha: } &Row\ 2 = [0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0], \\
&Row\ 3 = [0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0], \\
&Row\ 5 = [1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], \\
&Row\ 9 = [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0].
\end{aligned}$$

All other rows are 0 rows.

$$\text{Beta: } Row\ 1 = [1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0].$$

All other rows are 0 rows.

Clearly, Alpha and Beta are disjoint, since they have no overlap of nonzero rows.

The 1-entries of Alpha and Beta applied on the matrix representation of $\mathbf{B}(8)$ lead to the following Positive Polarity Reed-Muller functional expression, where to simplify the representation, x_{ijk} denotes $x_i x_j x_k$:

$$\begin{aligned} f(x_1, \ldots, x_8) = {} & \langle 1 \oplus x_3 \oplus x_4 \oplus x_{13} \oplus x_{23} \oplus x_{123} \rangle \\ & \oplus \langle x_6 \oplus x_{15} \oplus x_{17} \oplus x_{18} \oplus x_{26} \oplus x_{27} \oplus x_{28} \\ & \oplus x_{36} \oplus x_{37} \oplus x_{47} \oplus x_{128} \oplus x_{138} \oplus x_{238} \oplus x_{348} \rangle. \end{aligned}$$

Notice that the functional expression for f has degree 3, which is below the $n/2$ upper limit for a Boolean function to be bent, as stated in [1].

Test of Lemma 2

$$\begin{aligned} & \mathbf{RM}(4) \cdot [1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0]^T \\ & \quad = [1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0]^T. \end{aligned}$$

It is simple to verify that once transposed, this corresponds to the first row of Beta.

The results of the above experiments (and additional experiments with $k = 5$ and $k = 6$ not included for space reasons) show that the distribution of the 1-entries in the nonzero rows depends on the permutation used in (2). Furthermore, the results show that for any permutation in (2), Alpha comprises just k nonzero rows, starting with the second row.

Theorem 2 *The distribution of the k non-0 rows of Alpha conforms to the following relationship: The second row of Alpha is a non-0 row. If $j > 2$ denotes the position of a non-0 row, then:*

$$j_{new} = 2(j_{previous}) - 1.$$

□

See experimental results in Table 1.

Proof by induction on k.

Let $u \in N$ and let Alpha_u denote Alpha when $k = u$. Furthermore, let $\mathbf{M}^{\perp}$ denote the skew-transpose of $\mathbf{M}$.

Preliminaries

$$\mathbf{RM}(1) \cdot \mathbf{U}(1) \cdot \mathbf{RM}(1)^T = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} = \mathbf{M}^{\perp}.$$

Table 1 Observations supporting the theorem

k	Positions of the non-0 rows of Alpha
2	2, 3
3	2, 3, 5
4	2, 3, 5, 9
5	2, 3, 5, 9, 17
6	2, 3, 5, 9, 17, 33

$$\mathbf{RM}(1)\cdot\mathbf{M}(1)\cdot\mathbf{RM}(1)^T=\begin{bmatrix}1&0\\1&1\end{bmatrix}\cdot\begin{bmatrix}0&0\\0&1\end{bmatrix}\cdot\begin{bmatrix}1&1\\0&1\end{bmatrix}=\begin{bmatrix}0&0\\0&1\end{bmatrix}=\mathbf{M}.$$

Induction Basis
Assume that for $k=u$, Alpha_u satisfies the proposition. Recall that, as shown in (13), the distribution of the kind of rows is independent of $\mathbf{P}(k)$. Therefore, in what follows, $\mathbf{P}(k)=\mathbf{I}(k)$ will be considered.

Induction Step
Let $k=u+1$ and $\mathbf{P}(u+1)=\mathbf{I}(u+1)$.

$$\begin{aligned}
\text{Alpha}_{u+1} &= \mathbf{RM}(u+1)\cdot\mathbf{M}^{[u+1]}\langle\mathbf{RM}(u+1)\rangle^T\\
&= \langle\mathbf{RM}(u)\otimes\mathbf{RM}(1)\rangle\langle\mathbf{M}^{[u]}\otimes\mathbf{U}(1)\oplus\mathbf{U}(u)\otimes\mathbf{M}\rangle\langle\mathbf{RM}(u)\otimes\mathbf{RM}(1)\rangle^T\\
&= \langle\mathbf{RM}(u)\otimes\mathbf{RM}(1)\rangle\cdot\langle\mathbf{M}^{[u]}\otimes\mathbf{U}(1)\rangle\langle\mathbf{RM}(u)\otimes\mathbf{RM}(1)\rangle^T\\
&\oplus\quad \langle\mathbf{RM}(u)\otimes\mathbf{RM}(1)\rangle\cdot\langle\mathbf{U}(u)\otimes\mathbf{M}\rangle\cdot\langle\mathbf{RM}(u)\otimes\mathbf{RM}(1)\rangle^T\\
&= \langle\mathbf{RM}(u)\cdot\mathbf{M}^{[u]}\cdot\mathbf{RM}(u)^T\rangle\otimes\langle\mathbf{RM}(1)\cdot\mathbf{U}(1)\cdot\mathbf{RM}(1)^T\rangle\\
&\oplus\quad \langle\mathbf{RM}(u)\cdot\mathbf{U}(u)\cdot\mathbf{RM}(u)^T\rangle\otimes\langle\mathbf{RM}(1)\cdot\mathbf{M}\cdot\mathbf{RM}(1)^T\rangle\\
&= \text{Alpha}_u\otimes\mathbf{M}^{\perp}\oplus\langle\mathbf{M}^{\perp}\rangle^{\otimes u}\otimes\mathbf{M}.
\end{aligned}$$

Notice that $\langle\mathbf{M}^{\perp}\rangle^{\otimes u}$ is a $(2^{u+1}\times 2^{u+1})$ matrix with a single 1-entry at the left upper corner. Therefore, $\langle\mathbf{M}^{\perp}\rangle^{\otimes u}\otimes\mathbf{M}$ is a 2^{u+2} square matrix with a single 1-entry at the second position of the second row, i.e., its second row is a non-0 row.

Analysis of $\text{Alpha}_u\otimes\mathbf{M}^{\perp}$
Every row of Alpha_u will be duplicated with double length. Since the first row of Alpha, for any k is a 0 row, this row will first be duplicated, however, because of the nature of $\langle\mathbf{M}^{\perp}\rangle^{\otimes u}\otimes\mathbf{M}$, as discussed above, the second row of Alpha_{u+1} will become a non-0 row. In every other row of Alpha_{u+1}, the 1-entries of Alpha_u will be replaced by $\mathbf{M}^{\perp}$, and the 0-entries will be replaced by (2×2) 0-matrices. Therefore, Alpha_{u+1} will have $u+1$ nonzero rows, since u of them will be preserved from Alpha_u and an additional one will be provided by $\langle\mathbf{M}^{\perp}\rangle^{\otimes u}\otimes\mathbf{M}$.

From the former analysis, Alpha_{u+1} may be considered as a matrix comprising blocks of pairs of rows. If the blocks are first assigned the same position as the rows

of Alpha$_u$, the first row of a j-th block will have $j-1$ preceding blocks, i.e., $2(j-1)$ preceding rows. Its own row position will then be $2(j-1)+1 = 2j-1$.

End of the proof. □

3 The Effect of a Subset of Spectral Invariant Operations

It is known that the basic spectral invariant operations [4] preserve the bentness of Boolean functions [19]. In what follows a subset of spectral invariant operations, called additive spectral invariant operations, will be considered.

Definition 2 Additive spectral invariant operations comprise the complementation of a function as well as adding one or more (complemented or uncomplemented) arguments to the function or to its complement. □

It will be shown that additive basic spectral invariant operations preserve the Maiorana-McFarland bentness of Boolean functions and that they only affect the Beta component of the Reed-Muller spectrum of an original Maiorana-McFarland Boolean function.

Let $\mathbf{V}$ be some vector. Then, $\mathbf{1}^{\mathbf{V}}$ will represent a column vector of the same length as $\mathbf{V}$, with all components with value 1.

Given an n-place Maiorana-McFarland Boolean function f with value vector $\mathbf{F}$ of length 2^n, then, with $1 \leq i, j \leq n$:

$$\mathbf{F} \oplus \mathbf{1}^{\mathbf{F}}; \mathbf{F} \oplus \mathbf{X}_i; \mathbf{F} \oplus \mathbf{1}^{\mathbf{F}} \oplus \mathbf{X}_i; \mathbf{F} \oplus \mathbf{X}_j; \mathbf{F} \oplus \mathbf{1}^{\mathbf{F}} \oplus \mathbf{X}_j;$$
$$\mathbf{F} \oplus \mathbf{X}_i \oplus \mathbf{X}_j; \mathbf{F} \oplus \mathbf{1}^{\mathbf{F}} \oplus \mathbf{X}_i \oplus \mathbf{X}_j; \ldots; \mathbf{F} \oplus_{r=1}^{n} \mathbf{X}_r,$$

are examples of value vectors of functions obtained with additive spectral invariant operations applied to $\mathbf{F}$. $\mathbf{X}_i$ and $\mathbf{X}_j$ represent the value vectors of the variables x_i and x_j in the context of n variables, similarly for other value vectors.

Let $k = 2$. With (2), $\mathbf{F} = vec\langle \mathbf{M}^{[2]} \cdot \mathbf{P}(2) \oplus (\mathbf{1}^{\mathbf{G}} \otimes [g_1, g_2, g_3, g_4]) \rangle$.
Analysis of $\mathbf{F} \oplus \mathbf{1}^{\mathbf{F}}$ (i.e., the complement of $\mathbf{F}$).

$$\begin{aligned}
\mathbf{F} \oplus \mathbf{1}^{\mathbf{F}} &= vec(\mathbf{M}^{[2]} \cdot \mathbf{P}(2) \oplus (\mathbf{1}^{\mathbf{G}} \otimes [g_1, g_2, g_3, g_4]) \oplus \mathbf{1}^{\mathbf{F}} \\
&= vec(\mathbf{M}^{[2]} \cdot \mathbf{P}(2) \oplus (\mathbf{1}^{\mathbf{G}} \otimes [g_1, g_2, g_3, g_4]) \otimes vec(\mathbf{1}^{\mathbf{G}} \otimes (\mathbf{1}^{\mathbf{G}})^T) \\
&= vec\langle \mathbf{M}^{[2]} \cdot \mathbf{P}(2) \oplus (\mathbf{1}^{\mathbf{G}} \otimes [g_1, g_2, g_3, g_4] \oplus (\mathbf{1}^{\mathbf{G}} \otimes \mathbf{1}^{\mathbf{G}})^T \rangle \qquad (15) \\
&= vec\langle \mathbf{M}^{[2]} \cdot \mathbf{P}(2) \oplus (\mathbf{1}^{\mathbf{G}} \otimes ([g_1, g_2, g_3, g_4] \oplus (\mathbf{1}^{\mathbf{G}})^T) \rangle \\
&= vec(\mathbf{M}^{[2]} \cdot \mathbf{P}(2) \oplus (\mathbf{1}^{\mathbf{G}} \otimes [\overline{g}_1, \overline{g}_2, \overline{g}_3, \overline{g}_4])
\end{aligned}$$

It is quite clear that (15) is a particular case of (2), with $\mathbf{G} = [\overline{g}_1, \overline{g}_2, \overline{g}_3, \overline{g}_4]$. Therefore, $\mathbf{F} \oplus \mathbf{1}^{\mathbf{F}}$ is the value vector of a (new) Maiorana-McFarland Boolean bent function.

Analysis of $\mathbf{F} \oplus \mathbf{X}_2$:

$$\begin{aligned}
\mathbf{F} \oplus \mathbf{X}_2 &= vec(\mathbf{M}^{[2]} \cdot \mathbf{P}(2)) \oplus (\mathbf{1}^{\mathbf{G}} \otimes [g_1, g_2, g_3, g_4]) \oplus \mathbf{X}_2) \\
&= vec(\mathbf{M}^{[2]} \cdot \mathbf{P}(2) \oplus (\mathbf{1}^{\mathbf{G}} \otimes [g_1, g_2, g_3, g_4]) \oplus vec(\mathbf{1}^{\mathbf{G}} \otimes [0, 1, 0, 1]) \\
&= vec\langle \mathbf{M}^{[2]} \cdot \mathbf{P}(2) \oplus (\mathbf{1}^{\mathbf{G}} \otimes [g_1, g_2, g_3, g_4]) \oplus (\mathbf{1}^{\mathbf{G}} \otimes [0, 1, 0, 1]) \rangle \quad (16) \\
&= vec\langle \mathbf{M}^{[2]} \cdot \mathbf{P}(2) \oplus (\mathbf{1}^{\mathbf{G}} \otimes ([g_1, g_2, g_3, g_4] \oplus [0, 1, 0, 1]) \rangle \\
&= vec(\mathbf{M}^{[2]} \cdot \mathbf{P}(2) \oplus (\mathbf{1}^{\mathbf{G}} \otimes [g_1, \overline{g}_2, g_3, \overline{g}_4]).
\end{aligned}$$

It is simple to see that (16) is also a particular case of (2), with a different $\mathbf{G}$. Therefore, $\mathbf{F} \oplus \mathbf{X}_2$ is the value vector of another (new) Maiorana-McFarland Boolean bent function.

The analysis of the effect of additive spectral invariant operations when $k > 2$ is straightforward.

It may be seen that all other cases of functions obtained with additive spectral invariant operations may be analyzed by repetitions of the above two cases, with appropriate definitions of the $\mathbf{X}$ vectors. In all cases it will be concluded that the considered spectral invariant operations preserved the Maiorana-McFarland bentness of an original Maiorana-McFarland Boolean bent function.

Furthermore, the analyzed cases indicate that if the Reed-Muller spectrum of the generated new functions is calculated, only its Beta component—(the one dependent of $\mathbf{G}$)—will change with respect to the Reed-Muller spectrum of the originating function, and, since in the matrix representation of Beta only the first row is a nonzero row, the "new" Beta may be simply calculated with Lemma 2.

4 Closing Remarks

Boolean bent functions are frequently, if not always, associated with their Walsh spectra, since the absolute value of the spectral coefficients determines whether a function is bent or not. In this chapter, however, we have established particular structural features of the Reed-Muller spectrum of functions in the Maiorana-McFarland class of Boolean bent functions. These features are disjoint and depend separately on the parameters $\mathbf{P}$ and $\mathbf{G}$ of the generating Eq. (2). If a set of Boolean bent functions is given, the analysis of the structure of the Alpha components of their Reed-Muller spectra would allow recognizing which bent functions are Maiorana-McFarland. Furthermore, we showed that additive basic spectral operations consisting of adding a constant vector 1 or the value vector of one or more possibly complemented arguments to a reference Maiorana-McFarland Boolean bent function preserves this class and affects only the Beta component of the Reed-Muller spectrum. Tests done with random Boolean bent functions, but not Maiorana-McFarland bent functions, showed that their Reed-Muller spectra have a structure different from the one shown in Theorem 2.

Acknowledgments The authors thank the reviewers of a preliminary version of this chapter for their constructive criticism and the suggestions to improve the manuscript.

References

1. Cusick, T.W., Stănică, P.: Cryptographic Boolean Functions and Applications. Academic Press/Elsevier (2009)
2. Dillon J.F.: Elementary Hadamard Difference Sets. Ph.D Dissertation, University of Maryland, College Park, 1974
3. Horn R.A., Johnson Ch.R.: Topics in Matrix Analysis. Cambridge University Press, New York (1991)
4. Hurst S.L.: The Logical Processing of Digital Signals. Crane Rusak, New York (1978)
5. Karpovsky M.G., Stanković, R.S., Astola, J.T.: Spectral Logic and its Applications for the Design of Digital Devices. Wiley, Hoboken (2008)
6. Langevin Ph., Leader G.: Counting all bent functions in dimension 8. Des. Codes Cryptogr. **9**, 203–215 (2011)
7. Laub A.J.: Matrix Analysis for Scientists and Engineers. SIAM (2005)
8. Maiorana J.A.: A Class of Bent Functions. R41 Technical Paper, August 1970
9. McFarland R.L.: A discrete Fourier theory for binary functions. R41 Technical paper, 1971
10. Mesnager S.: Bent Functions: Fundamentals and Results. Springer, Switzerland, 2016
11. Moraga, C., Stanković, M., Stanković, R.S., Stojković, S.: Contribution to the study of multiple-valued bent functions, in Proc. 43rd Int. Symp. Multiple-valued Logic, pp. 340–345. IEEE Press (2013)
12. Moraga, C., Stanković, M., Stanković, R.S., Stojković, S.: The Maiorana Method to generate Multiple-valued Bent Functions revisited, in *Proc. 44th Int. Symp. Multiple-valued Logic*, pp. 340–345. IEEE Press (2014)
13. Muller D. E.: Application of Boolean algebra to switching circuits design and to error detection. In: IRE Trans. Electron. Comp., vol. EC-3, 6–12, 1954
14. Radmanović, M., Stanković, R. S.: Construction of subsets of bent functions satisfying restrictions in the Reed-Muller domain. Facta Universitatis, Ser. Electron. Energetics **31**(2), 217–222 (2018)
15. Reed, S.M.: A class of multiple error correcting codes and their decoding scheme. In: IRE Trans. Inf. Th., vol. PGIT-4, pp. 38–49 (1954)
16. Rothaus O.: On bent functions. J. Combin. Theory. Ser. A **21**(3), 300–305 (1976)
17. Stanković, R.S., Astola, J.T., Moraga, C.: Spectral Techniques for Boolean Problems. Origins and Applications. In: Steinbach, B. (ed.) Further Improvements in the Boolean Domain, pp. 54–63. Cambridge Scholars Publishing, Newcastle Upon Tyne (2018)
18. Stanković, R.S., Stanković, M., Moraga, C., Astola. J.: Quaternary generalized Boolean bent functions obtained through permutation of binary Boolean bent functions, in Proc. 48th Int. Symp. on Multiple-Valued Logic, Linz, Austria, May 16–18, pp. 1–6. IEEE Press (2018)
19. Stanković, M., Moraga, C., Stanković, R.S.: Some spectral invariant operations for multiple-valued functions with homogeneous disjoint products in the polynomial form. In: Proc. 47th Int. Symp. on Multiple-Valued Logic, pp. 61–66. Novi Sad, Serbia. IEEE Press (2017)
20. Tokareva, N.: Bent Functions—Results and Applications to Cryptography. Elsevier, Amsterdam (2015)
21. Zhegalkin, I.L.: O tekhnyke vychyslenyi predlozhenyi v symbolytscheskoi logykye. Math. Sb. **34**, 9–28 (1927). In Russian
22. Zhegalkin, I.L.: Aritmetizatiya symbolytscheskoi logyky. Math. Sb. **35**, 311–377 (1928). In Russian

Toward System-Level Assertions for Heterogeneous Systems

Muhammad Hassan, Thilo Vörtler, Karsten Einwich, Rolf Drechsler, and Daniel Große

1 Introduction

Driven by growth opportunities in various application domains, e.g., *Internet of Things* (IOT), many semiconductor vendors are shifting their focus toward a more integrated solution of high-performance *analog/mixed-signal* (AMS) designs. Due to this industry shift, most *System-on-Chips* (SOCs) today are AMS containing analog sensors, mixed-signal converters, and digital processors running *Software* (SW) on top, tightly integrated on a single die. One characteristic of such SOCs is that each subsystem interacts simultaneously with each other by internal connections and reacts to inputs coming from outside. Digital systems behavior usually exhibits discrete changes in time and value, whereas analog circuits usually exhibit continuous changes. While this shift has resulted in high-performance and low-area devices, it has significantly increased the efforts required to develop and verify these highly complex devices and achieving the required *Time To Market* (TTM)

M. Hassan (✉)
Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
e-mail: muhammad.hassan@dfki.de; hassan@uni-bremen.de

T. Vörtler · K. Einwich
COSEDA Technologies GmbH, Dresden, Germany
e-mail: thilo.voertler@coseda-tech.com; karsten.einwich@coseda-tech.com

R. Drechsler
Institute of Computer Science, University of Bremen and Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
e-mail: drechsle@informatik.uni-bremen.de

D. Große
Institute for Complex Systems, Johannes Kepler University, Linz, Austria

Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
e-mail: daniel.grosse@jku.at

R. Drechsler, S. Huhn (eds.), *Advanced Boolean Techniques*,
https://doi.org/10.1007/978-3-031-28916-3_5

simultaneously. Nowadays, *assertion-based verification* (ABV) in combination with coverage analysis [21, 22, 36] and constrained randomization techniques [19, 50] is widely used to perform functional verification of digital designs at *Register Transfer Level* (RTL). ABV defines temporal properties in order to verify the functional correctness of the design with respect to expected behaviors. Consequently, the bugs are found at their source. Furthermore, design observability and controllability is also improved. Applying the ABV methodology to AMS designs can bring the same benefits that the digital design community has enjoyed. However, late availability of RTL in the design process exacerbates the situation.

In this regard, the emergence of *virtual prototypes* (VPs) at the abstraction of *Electronic System Level* (ESL) has modernized the design and verification of AMS SOCs in many ways [11, 13, 14, 20, 23–25, 33, 41, 47]. Essentially, a VP is a software simulation model of the entire *Hardware* (HW) platform, created by composing models of the individual *Intellectual Property* (IP) blocks (i.e., instruction set simulators, bus and peripheral models, etc.). For this purpose, the C++-based system modeling language SystemC together with *Transaction-level Modeling* (TLM) techniques [27] and mixed-signal extension SystemC/AMS [4] is being heavily used in industrial practice [3, 11, 22, 33, 34, 41]. Overall, the adoption of VPs has led to significant improvements on the design and verification of SOCs. Because of earlier availability and significantly faster simulation speed as opposed to RTL, the VPs enable HW/SW co-design and verification very early in the development flow. Serving as reference for (early) embedded SW development and HW verification, the functional correctness of VPs is very important. Hence, a whole VP and its individual components are subjected to rigorous verification.

However, one of the main challenges is the availability of a practical assertions library for system-level design verification which enables ABV methodologies. When speaking of unavailability, we also broadly include advanced testbench concepts based on the *Universal Verification Methodology* (UVM), or in the future even more abstract based on *Portable Stimulus Specification* (PSS). Regardless of the specific solution, a system-level assertions library is missing which satisfies the following: (1) expressiveness to represent complex behaviors of heterogeneous systems, (2) compatibility to SystemC, TLM, and SystemC/AMS, (3) capture of complex analog-digital interactions, and (4) integration of complex heterogeneous characteristics like continuous time, frequency analysis, etc.

Contribution In this paper we present a system-level assertions library for heterogeneous systems, an advanced *ABV* environment for SystemC, TLM, and its mixed-signal extension SystemC/AMS. To overcome the limitations of state-of-the art libraries (see Sect. 2 for more details), the proposed SystemC assertions library provides the following features:

- New assertions specification API: An intuitive, user-friendly, and expressive *Application Programming Interface* (API) to specify complex behaviors of nontrivial heterogeneous systems has been developed.
- Compatibility: The library is compatible with SystemC and its extensions, TLM, and SystemC/AMS.

- Complex behaviors: Various complex behaviors can be captured like, (1) complex analog-to-digital, (2) digital-to-analog, (3) digital-to-digital, and (4) analog-to-analog.
- SW and TLM Support: The assertions library supports the checking of TLM interface and SW/HW interactions.
- Heterogeneous characteristics: The library integrates heterogeneous characteristics like continuous time, frequency analysis, slopes, equations, attenuations, *differential-algebraic equations* (DAE), digital signals, temporal logic, variables, and events. These characteristics are necessary for expressing complex properties.
- Improved usability: Debugging of failed assertions is supported.

Considering all these features, we develop a new system-level assertions library for bridging the gap of ABV for heterogeneous systems. The running example and experiments on a real-world model of ARM V8-based CPU using ARM fast models demonstrate the capabilities of the library to improve the system verification in a significant way.

The paper is organized as follows: Sect. 2 gives a survey of current approaches concerning heterogeneous/AMS verification. Section 3 discusses the running example along with assertions to set up the environment. Section 4 describes our contribution and the implementation and discusses the approach. This includes syntax and semantics of the system-level assertions library. In Sect. 5 we demonstrate the benefits of our methodology with experiments. Finally, we conclude and mention future work in Sect. 6.

2 Related Work

SystemC is widely used for system-level design and verification; however, it still lacks native temporal assertions support. Several approaches have been proposed for digital SystemC-based models/VPs. Besides basic work on the temporal language itself [45], these approaches can be divided into two categories, formal assertion-based verification (e.g., [7, 12, 17, 18, 29, 31, 46, 48]) and simulation-based verification (e.g., [5, 6, 9, 13, 15, 40, 44]). The formal approaches aim to fully explore the state space based on abstract representations of system-level models. However, these approaches typically run into the state space explosion problem. Furthermore, the aforementioned simulation-based methods only consider purely digital models.

In [6, 9, 40] new approaches for transaction-level assertions are introduced. However, in [40] transactions are mapped to signals, and therefore the approach is restricted only to transactions which are invoked by suspendable processes. In [6] transactions are recorded and written into a trace to do post-processing. Trace-based assertion checking however requires that everything to be recorded must be annotated in the code and the creation of simulation data bases can become very resource intensive.

Various works have also been presented for the specification and verification of analog circuits [32, 35, 39, 42, 43, 51]. Here, too, a distinction is made between formal and simulation-based methods. One focus of the work was in particular to develop suitable extensions for the specification of assertions. It should be noted, however, that the aforementioned works only target analog components and usually only address the implementation level. The overall heterogeneous systems (incl. SW) and environment considered here are not supported.

In the area of digital HW/SW co-design and verification, various formal approaches have been proposed, for example [16, 38, 49]. However, these so far assume only implementation-level descriptions for the hardware part (e.g., in Verilog or VHDL). Furthermore, due to the huge state spaces in analog domain, only small problems can be handled. Recently, abstraction techniques have been developed, and the hardware parts are abstracted to C level [26, 37]. However, these methods consider only pure digital designs.

Heterogeneous characteristics like continuous time, frequency analysis, slopes, equations, attenuations, DAE, digital signals, temporal logic, variables, and events are insufficiently integrated in all known specification languages. However, these characteristics in combination with a special time definition are necessary for expressing complex properties. Therefore, our work considers all these conditions to develop a new system-level assertions library for bridging the gap of ABV for heterogeneous designs.‘

3 Preliminaries

3.1 Assertion-Based Verification

ABV is an established technique used nowadays to verify SOCs [10]. To enable ABV, a language is required based on the general notion of *Property Specification Language* (PSL) [28], *Linear Temporal Logic* (LTL), Finite LTL (FLTL), or Computation Tree Logic (CTL) [30]. Based on the specification assertions (properties) are typically manually created and capture the design intent. The basic function of an assertion is to specify a set of behaviors that is expected to be true for a given *Design Under Verification* (DUV). Assertions are included in the DUV via monitors, and they compare the temporal behavior of the assertions against the DUV during simulation. Assertions are used in the validation environments of TLM, RTL, and gate level and offer the following advantages: (1) detect design errors at their source and increase observability, (2) actively monitor a design to ensure correct functional behavior, and (3) can be used for functional and formal verification. The widely used assertions library for RTL, *SystemVerilog Assertion* (SVA) [8], unifies simulation and formal verification semantics to drive the design for verification methodology. It takes a layered approach to define the properties of the DUV. More precisely, properties are composed of four layers: (1) the **Boolean layer** consists of propositions and Boolean connectives, (2) the **sequence layer** adds operators for

temporal reasoning to the Boolean layer. (3) the **property layer** defines operations on sequences, and (4) the **verification layer** provides indicators for the verification tools on how to apply the properties. Most often assertions use implication operators which define some specific sequence of events (known as *antecedent*) which should occur before another sequence of events (known as *consequent*) should occur.

The first three layers define the actual property (intended or error state) that relates to parts of the DUV, whereas the fourth layer is used to control the high-level behavior of the verification tools.

3.2 System-Level Running Example

For brevity, we refrain from giving a proper introduction to SystemC, TLM, and SystemC/AMS. Instead, we present here a heterogeneous system as a running example (Fig. 1) that will be used to showcase the main ideas of our approach throughout this paper. The SystemC, TLM, and SystemC/AMS constructs and semantics necessary to understand the example will be explained as needed. The running example models a temperature control system covering multiple domains, i.e., SW, digital HW, and analog behavior. The system is modeled in SystemC/AMS using different *Models of Computation* (MoC), in particular *Timed Data Flow* (TDF) and *Electrical Linear Networks* (ELNs). The overall system as shown in Fig. 1 consists of the following components:

- an ARM V8-based CPU using ARM fast models implemented as SystemC TLM [2] with Linux operating system and SW running on top,
- four ADT7420 temperature sensors implemented as SystemC/AMS TDF and discrete event model [1],
- an *Advanced Microcontroller Bus Architecture* (AMBA) bus that acts as a bridge device to connect temperature sensors and ARM processor (created in SystemC TLM)—(*COS_AMBA_DEVICE* in Fig. 1),
- an environment model (*Thermal_Network*) that builds 3 connected rooms and an ambient temperature modeled as a sinus (*SIN_SRC_TDF*), i.e., each sensor senses a different temperature (implementation as SystemC/AMS ELN and discrete event model), and finally
- a heater model implemented as SystemC/AMS ELN that can be used to increase the temperature.

The communication between SW running on the ARM8 and the connected sensors is done via registers connected to the bus of the processor. The SW configures the sensors by writing to addresses on the bus, which in turn creates TLM transactions. These TLM transactions are written into the corresponding registers of the ADT7420 sensors. The AMBA bus (*COS_AMBA_DEVICE*) also translates the AMBA-PV transactions used by ARM fast models. Additionally, I^2C transactions of the sensor model are also translated. To showcase the features of proposed system-level assertions library, the running example considers the following scenario for demonstration purposes:

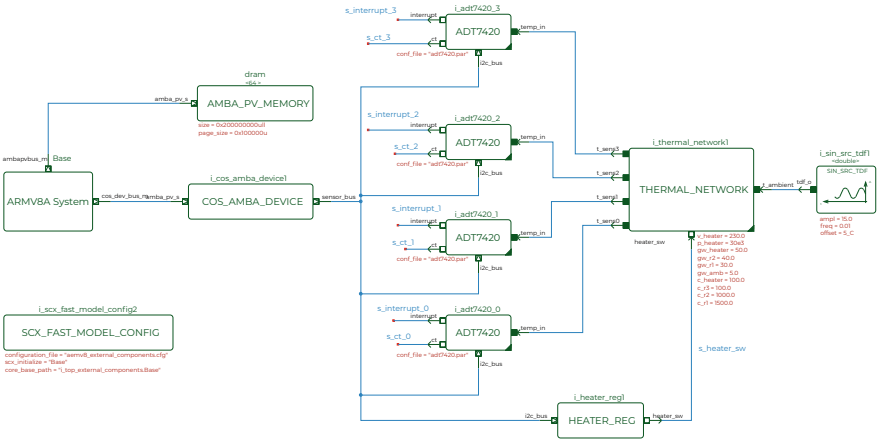


Fig. 1 Schematic of running example: temperature control system

- booting a Linux operating system on the ARM processor,
- a control SW is executed on top of Linux. The control SW continuously measures (monitors) the temperature sensor output,
- if the SW detects that the temperature value falls below a programmed threshold value, it switches the heater to *ON* state,
- otherwise, when the temperature exceeds a certain programmed threshold, the heater is switched to *OFF* state.

3.3 Assertions for System-Level Running Example

A lot of assertions can be defined for the running example introduced in Sect. 3.2. However, for the purpose of demonstrating the features of the proposed system-level assertions library, we focus on only one. The concrete assertion states that:

- When the temperature of Room 1 *t_r1* (SystemC TDF signal) is above the threshold *t_threshold* (SW-controlled TLM register value), the heater has to be switched off (*heater_sw*) within 1 ms.

How this heterogeneous assertion can be expressed in our proposed assertions library can be seen in Listing 1. Please note that we introduce all ingredients (in particular, API, layers, etc.) from the users' perspective for the proposed system-level assertions library in the next sections.

```
1  auto heater_off = (t_r1 > t_threshhold) ->* (true |
       delay(1_SC_MS) | (heater_switch==false));
2  heater_off.default_sampling(1_SC_MS);
```

Listing 1 Concrete assertion for temperature control system example

4 System-Level Assertions Library for Heterogeneous Systems

In this section, we introduce the proposed system-level assertions library and its components for bridging the gap of ABV for heterogeneous systems. First, we provide a brief overview of the library. Then, we describe the intuitive API and the layered architecture of the assertions library in detail while always providing an example.

4.1 Overview

The system-level assertions library is developed with an intuitive, user-friendly, and an expressive API. As a result, complex behaviors of heterogeneous systems can be captured easily. These behaviors are not only limited to events taking place at one point in time in one domain, rather also temporal behaviors across different domains, e.g., TLM and analog. To enable the API expressiveness, a layered architecture inline with SVA-layered architecture [8] is used, i.e., Boolean layer, sequence layer, property layer, and verification layer. At the back end, first the assertion is divided into different layers and expressions; then, multiple SystemC processes are spawned to monitor the signals and events specified in the expressions. The library uses linear time model where the assumption is that the time is linear. Each assertion is synchronized to the sampling ticks (notion of discrete time) of DUV as defined by SystemC/AMS semantics, unless specified. The assertion is evaluated at each sampling tick. If the specified expressions evaluate to *true*, the assertion is satisfied. Additionally, the complete trace of assertion evaluation is displayed to the verification engineer.

In the following sections, the components of system-level assertions library are explained in detail.

4.2 Application Programming Interface

The API of the library is designed to enable the expressiveness required for specifying cross-domain behaviors, e.g., TLM and analog. Hence, dedicated functions like *delay(...)*, *repeat(...)*, *default_sampling(...)*, etc. are defined to specify the behaviors and make the library user-friendly. Additionally, operators (e.g., pipe (|), ->*) are introduced to enable specification of sequences in SystemC.

The concrete assertion (specified in Listing 1) is interpreted in light of the proposed API as follows: an assertion property *heater_off* is created. The property joins two sequences via an overlapping implication operator (->*). The sequences are,

Table 1 Non-comprehensive list of supported Boolean expressions by system-level assertions library

Operator	Name	Data type
+= -= /= *= &= |=	Binary assignment operators	int, double
< <= > >=	Binary relational operators	int, double
+ − ∗ /	Binary arithmetic operators	int, double
&& || == !=	Binary logical operators	int, double
+ - ! ++ −	Unary operators	int, double

(1) antecedent—(t_r1 > t_threshhold), (2) consequent—*(true | delay(1_SC_MS) | (heater_switch==false))*. The sequences comprise of *four* Boolean expressions in total: (1) *(t_r1 > t_threshhold)*, (2) *true*, (3) *delay (1_SC_MS)*, (4) *(heater_switch==false)*. Furthermore, the sampling time of the assertion is written in Line. 2, i.e., *1_SC_MS*.

4.3 Boolean Layer

The Boolean layer describes the behaviors of primitive elements relative to each other at a particular point in time. The primitive elements in our proposed library are SystemC events, variables, and SystemC/AMS signals. These primitive elements are related using arithmetic, logical, or relational operators. Consequently, they form an expression, e.g., a relational expression. In Listing 1 the expression *(t_r1 > t_threshhold)* compares an analog signal *t_r1* with a digital threshold value *t_threshhold* stored in TLM register. If the relational condition is satisfied, the expression is evaluated to *true*. A non-comprehensive list of Boolean expressions is shown in Table 1.

4.4 Sequence Layer

The sequence layer builds on top of Boolean layer to specify the temporal relationship between primitive elements (Boolean expressions) over time. The sequence layer also specifies sequences as either a combination of simpler sequences using sequence operators or as basic Boolean expressions correlated by events. The proposed API introduces the pipe operator (|) to represent the continuity of a sequence. This increases readability as well as user-friendliness of the assertion property. Additionally, the API introduces *delay(...), repeat(...)* operators to specify temporal assertions. As a result, a sequence can comprise of delay operators (Sect. 4.4.1), Boolean expressions, and event expressions. To determine a match of the sequence, the Boolean expressions are evaluated at each successive sample tick, defined by a sampling event (SystemC/AMS sampling points) that gets associated

Table 2 Non-comprehensive list of supported sequence operators by system-level assertions library

Operator	Description
delay	Specifies delay from current sampling point until the next
and	Sequence *and* operation
or	Sequence *or* operation
repeat	Repetition operator

with the sequence. If all expressions of the sequence are true, then a match of the sequence occurs. For example, the assertion in Listing 1 has the expressions:

$$(true|\ delay(1\_SC\_MS)|\ (heater\_switch == false))$$

The expressions are interpreted as follows: a signal is asserted—*true*, followed by a delay operator—*delay(1_SC_MS)*, and after the delay of 1 ms, the expression *(heater_switch == false)* is evaluated. The sequence returns *true* only if all the expressions evaluate to *true*. A non-comprehensive list of supported sequence operators is shown in Table 2.

4.4.1 Delay Operator

The system-level assertions library introduces delay operator—*delay(delay_cycles)* and *delay(min_delay_cycles, max_delay_cycles)* which takes delay time as input. The function of delay operator is to create a relationship between Boolean expressions over a period of time or between the given time constraints.

4.4.2 Repeat Operator

The library also introduces repeat operator—*repeat(value)* and *repeat(min_value, max_value)*—which takes a repetition value as input for how many times the sequence should be repeated. It helps in cases when a certain set of expressions are expected to be true over multiple time points.

4.4.3 Sequence "and/or" Operators

The system-level assertions library introduces the sequence *"and/or"* operators. The sequences are evaluated in parallel. In case of *"and"* operator, if one sequence evaluates to *"false"*, the evaluation stops and the assertion fails. On the other hand, in case of *"or"* operator, the library waits for all sequences to be evaluated.

4.5 Property Layer

The property layer allows for more general behaviors to be specified, i.e., specification of properties as either a combination of simpler properties using property operators or as an implication built up from several sequences. In particular, properties allow users to invert the sense of a sequence (e.g., when the sequence should not happen), disable the sequence evaluation, or specify that a sequence be implied by some other occurrence. The properties and their respective sequences (including Boolean expressions) are evaluated on each sampling event (sampling tick) of the system's default sampling time, unless specified. In this concrete assertion (defined in Listing 1), the property sampling time is set to 1 ms (*heater_off.default_sampling (1_SC_MS)*). As a result, the assertion property in Listing 1 is evaluated every ms. The property layer supports implication operators, *"not"*, and *"and/or"* operators.

4.5.1 Implication Operator

An implication refers to a situation in which in order for a behavior to occur, a preceding sequence must have occurred. This preceding sequence in this case is known as *antecedent*. The succeeding behavior is known as *consequent*. Evaluation of an implication starts through repeated attempts to evaluate the antecedent. When the antecedent succeeds, the consequent is required to succeed for the property to hold. Thus, in other words, an antecedent sequence implies a consequent property expression, as follows:

$$antecedent -> * consequent$$

$$where -> * = overlapping\ implication\ operator$$

Nonoverlapping Implication

The *delay(...)* operator is used to implement nonoverlapping implication.

Overlapping Implication

->* In the system-level assertion library, we introduce an overlapping implication operator (->*). This means that if the antecedent sequence is evaluated to *true*, the consequent sequence is evaluated at the same tick.

As shown in Listing 1, if the expression *(t_r1 > t_threshhold)* is true, the sequence *(true |delay(1_SC_MS) |(heater_switch==false))* should be true in next sampling ticks. A non-comprehensive list of supported property operators is shown in Table 3.

Table 3 Non-comprehensive list of supported property operators by system-level assertions library

Operator	Description
Not	the evaluation of the property returns the opposite of the evaluation of the underlying property expression
and	The property evaluates to true if, and only if, both property expression 1 and property expression 2 evaluate to true.
or	The property evaluates to true if, and only if, at least one of property expression 1 and property expression 2 evaluates to true.

4.6 Verification Layer

The verification layer specifies which properties are to be asserted or covered. This layer always associates properties with corresponding verification directives. A verification directive can be parameterized by the severity level and an info string; further on it can be specified if the property should be asserted, covered or both. The proposed library supports only *assert* at the moment.

5 Experiments

This section describes the experimental evaluation on a real-world model integrating an ARM V8 CPU via ARM fast models (as described in Sect. 3.2). Fast models are accurate, flexible programmer's view models of ARM IP, allowing one to develop software such as drivers, firmware, OS, and applications prior to silicon availability. They allow full control over the simulation, including profiling, debug, and trace. As mentioned, the complete model is implemented as a mixture of a SystemC TLM model and a SystemC/AMS model.

Several assertions were created to verify the behavior of temperature control system. The behaviors to verify included but not limited to: (1) SW and TLM interactions, (2) analog and TLM interactions, (3) analog-digital, (4) digital-analog, (5) digital events, and (6) analog-analog interactions, etc. In the following, we detail the results of the concrete assertion from Listing 1.

Partial simulation results of the temperature control system SW are shown in Fig. 2. The *orange* sinus signal is the ambient temperature (*SIN_SRC_TDF*) which oscillates between 262 K and 293 K. The *green* waveform signal (*t_r1*) is the temperature of room 1. The *blue* waveform signal (*t_r2*) is the temperature of room 2. The *purple* waveform signal (*t_r3*) is the temperature of room 3. At the bottom of Fig. 2, digital signals—*heater_switch* and interrupts (irq0-irq3) from temperature sensors are displayed.

After booting the Linux OS (approx. 30s), the control SW gets started. The heater (*heater_switch*) gets turned on as the temperature in room one (*t_r1*) is below the minimum temperature of 292 K. It can be seen how the temperature slowly increases

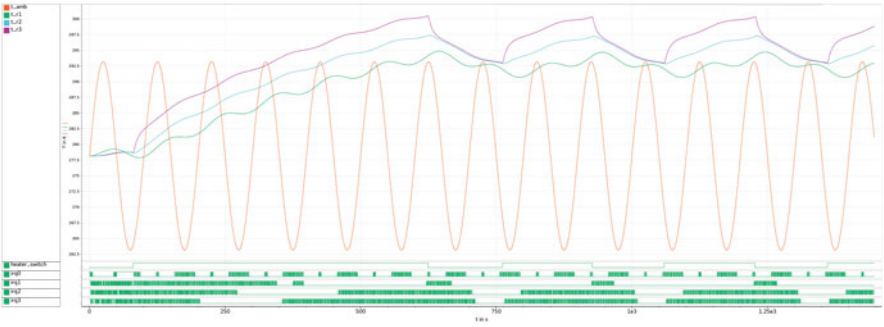


Fig. 2 Simulation results running the temperature control SW

in all rooms. When the temperature is above the maximum threshold of 294.15 K, the heater gets turned off. As a consequence, the room temperatures start to decrease. The sensors have been programmed to generate an interrupt whenever the temperature is above or below a threshold value (stored in register).

We could see the assertion was satisfied throughout the simulation. However, if we decreased the *delay(...)* from 1 ms to a smaller value, the assertion was always violated. This is expected and in accordance with the specifications. They require that the *heater_switch* should be turned off within 1 ms after the threshold temperature is crossed. The reason for 1 ms is because of the inherent delays due to reading and writing of registers in different connected devices, and can be summarized as follows:

- the temperature sensor senses the temperature,
- the sensed temperature is written into the register,
- SW reads the temperature from the ARM processor,
- SW checks whether the sensed temperature value is above the threshold value,
- and writing the heater switch control register depending on the comparison result.

Hence, using the proposed intuitive system-level assertion library, it is possible to check complex behaviors of the heterogeneous systems, e.g., digital, analog, and SW behavior.

6 Conclusion

In this paper, we presented a practical system-level assertions library for heterogeneous systems. The library comprises of an intuitive and user-friendly API and offers full compatibility with SystemC, TLM, and SystemC/AMS. As a result, the library supports specification of SW, TLM, and complex interactions, all necessary to represent complex AMS behavior. The system-level assertions library prototype

was used to verify the industrial model using ARM fast models, a temperature control system SW, environment models, temperature sensors, and assertions.

Acknowledgments This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project AUTOASSERT under contract no. 16ME0117 and by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria.

References

1. Analog Devices ADT7420 Data Sheet Rev. A (2017). https://www.analog.com/en/products/adt7420.html
2. ARM Fast Models Version 11.17 User Guide (2022). https://developer.arm.com/documentation/100965/1117/
3. Barnasconi, M., Adhikari, S.: ESL design in SystemC AMS: introducing a top-down design methodology for mixed-signal systems. In: DAC, pp. 1–5 (2017)
4. Barnasconi, M., Grimm, C., Damm, M., Einwich, K., Louërat, M., Maehne, T., Pecheux, F., Vachoux, A.: SystemC AMS extensions user's guide. Accellera Systems Initiative (2010)
5. Bombieri, N., Fummi, F., Guarnieri, V., Pravadelli, G., Stefanni, F., Ghasempouri, T., Lora, M., Auditore, G., Marcigaglia, M.N.: Reusing RTL assertion checkers for verification of systemC TLM models. J. Electron. Testing **31**(2), 167–180 (2015)
6. Chen, X., Luo, Y., Hsieh, H., Bhuyan, L., Balarin, F.: Assertion based verification and analysis of network processor architectures. Design Autom. Embed. Syst. **9**(3), 163–176 (2004)
7. Cimatti, A., Narasamdya, I., Roveri, M.: Software model checking SystemC. TCAD **32**(5), 774–787 (2013)
8. Committee, D.A.S., et al.: Ieee standard for systemverilog unified hardware design, specification, and verification language standard IEEE 1800 (2005). http://www.edastds.org/sv/
9. Ecker, W., Esen, V., Steininger, T., Velten, M., Hull, M.: Implementation of a transaction level assertion framework in SystemC. In: DATE, pp. 894–899 (2007)
10. Foster, H.D., Krolnik, A.C., Lacey, D.J.: Assertion-Based Design. Springer, Berlin (2004)
11. Grimm, C., Barnasconi, M., Vachoux, A., Einwich, K.: An introduction to modeling embedded analog/mixed-signal systems using SystemC AMS extensions. In: DAC, vol. 23 (2008)
12. Große, D., Drechsler, R.: Formal verification of LTL formulas for SystemC designs. In: ISCAS, pp. V:245–V:248 (2003)
13. Große, D., Drechsler, R.: Checkers for SystemC designs. In: MEMOCODE, pp. 171–178 (2004)
14. Große, D., Drechsler, R.: Quality-Driven SystemC Design. Springer, Berlin (2010)
15. Große, D., Groß, M., Kühne, U., Drechsler, R.: Simulation-based equivalence checking between SystemC models at different levels of abstraction. In: GLSVLSI, pp. 223–228 (2011)
16. Große, D., Kühne, U., Drechsler, R.: Hw/sw co-verification of embedded systems using bounded model checking. In: GLSVLSI, pp. 43–48 (2006)
17. Große, D., Le, H.M., Drechsler, R.: Proving transaction and system-level properties of untimed SystemC TLM designs. In: MEMOCODE, pp. 113–122 (2010)
18. Habibi, A., Tahar, S.: Assertion and model checking of SystemC. In: North American SystemC Users Group Meeting, San Diego, California, USA (2004)
19. Haedicke, F., Le, H.M., Große, D., Drechsler, R.: CRAVE: an advanced constrained random verification environment for SystemC. In: SoC, pp. 1–7 (2012)
20. Hassan, M., Große, D., Drechsler, R.: Enhanced Virtual Prototyping for Heterogeneous Systems. Springer, Berlin (2022)
21. Hassan, M., Große, D., Le, H.M., Drechsler, R.: Data flow testing for SystemC-AMS timed data flow models. In: DATE, pp. 366–371 (2019)

22. Hassan, M., Große, D., Vörtler, T., Einwich, K., Drechsler, R.: Functional coverage-driven characterization of RF amplifiers. In: FDL, pp. 1–8 (2019)
23. Herdt, V., Große, D., Drechsler, R.: Enhanced Virtual Prototyping: Featuring RISC-V Case Studies. Springer, Berlin (2020)
24. Herdt, V., Große, D., Le, H.M., Drechsler, R.: Early concolic testing of embedded binaries with virtual prototypes: a RISC-V case study. In: DAC, pp. 188:1–188:6 (2019)
25. Herdt, V., Le, H.M., Große, D., Drechsler, R.: Verifying SystemC using intermediate verification language and stateful symbolic simulation. TCAD **38**(7), 1359–1372 (2019)
26. Huang, B.Y., Ray, S., Gupta, A., Fung, J.M., Malik, S.: Formal security verification of concurrent firmware in SoCs using instruction-level abstraction for hardware. In: DAC, pp. 1–6 (2018)
27. IEEE Std. 1666: IEEE Standard SystemC LRM (2011)
28. IEEE Std. 1850: IEEE Standard for Property Specification Language (PSL) (2005)
29. Karlsson, D., Eles, P., Peng, Z.: Formal verification of SystemC designs using a petri-net based representation. In: DATE, pp. 1228–1233 (2006)
30. Kropf, T.: Introduction to Formal Hardware Verification. Springer, Berlin (1999)
31. Lämmermann, S., Ruf, J., Kropf, T., Rosenstiel, W., Viehl, A., Jesser, A., Hedrich, L.: Towards assertion-based verification of heterogeneous system designs. In: DATE, pp. 1171–1176 (2010)
32. Lämmermann, S., Weiss, R., Ruf, J., Kropf, T., Rosenstiel, W., Jesser, A., Hedrich, L.: An assertion-based verification methodology for SystemC-AMS designs. In: The 15th Workshop on Synthesis And System Integration of Mixed Information Technologies, pp. 434–439 (2009)
33. Lora, M., Vinco, S., Fraccaroli, E., Quaglia, D., Fummi, F.: Analog models manipulation for effective integration in smart system virtual platforms. TCAD **37**(2), 378–391 (2018)
34. Ma, K., Van Leuken, R., Vidojkovic, M., Romme, J., Rampu, S., Pflug, H., Huang, L., Dolmans, G.: A precise and high speed charge-pump PLL model based on systemC/systemC-AMS. Int. J. Electron. Telecommun. **58**, 225–232 (2012)
35. Maler, O., Ničković, D.: Monitoring properties of analog and mixed-signal circuits. International J. Softw. Tools Technol. Transf. **15**(3), 247–268 (2013)
36. Mehta, A.B.: System Verilog Assertions and Functional Coverage: Guide to Language, Methodology and Applications. Springer, Berlin (2019)
37. Mukherjee, R., Purandare, M., Polig, R., Kroening, D.: Formal techniques for effective co-verification of hardware/software co-designs. In: DAC, pp. 1–6 (2017)
38. Nguyen, M.D., Wedler, M., Stoffel, D., Kunz, W.: Formal hardware/software co-verification by interval property checking with abstraction. In: Proceedings of the 48th Design Automation Conference, pp. 510–515 (2011)
39. Ničković, D., Lebeltel, O., Maler, O., Ferrère, T., Ulus, D.: Amt 2.0: qualitative and quantitative trace analysis with extended signal temporal logic. Int. J. Softw. Tools Technol. Transf. **22**(6), 741–758 (2020)
40. Niemann, B., Haubelt, C., et al.: Assertion-based verification of transaction level models. In: MBMV, pp. 232–236. Citeseer (2006)
41. Pêcheux, F., Grimm, C., Maehne, T., Barnasconi, M., Einwich, K.: SystemC AMS based frameworks for virtual prototyping of heterogeneous systems. In: ISCAS, pp. 1–4 (2018)
42. Radojicic, C., Grimm, C., Schupfer, F., Rathmair, M.: Verification of mixed-signal systems with affine arithmetic assertions. VLSI Design (2013)
43. Steinhorst, S., Hedrich, L.: Model checking of analog systems using an analog specification language. In: DATE, pp. 324–329 (2008)
44. Tabakov, D., Vardi, M.: Monitoring temporal SystemC properties. In: MEMOCODE, pp. 123–132 (2010)
45. Tabakov, D., Vardi, M., Kamhi, G., Singerman, E.: A temporal language for SystemC. In: FMCAD, pp. 1–9 (2008)
46. Vardi, M.Y.: Formal techniques for SystemC verification. In: DAC, pp. 188–192 (2007)
47. Vörtler, T., Einwich, K., Hassan, M., Große, D.: Using constraints for SystemC AMS design and verification. In: DVCon Europe (2018)

48. Weiss, R.J., Ruf, J., Kropf, T., Rosenstiel, W.: Efficient and customizable integration of temporal properties into SystemC. In: Applications of Specification and Design Languages for SoCs, pp. 101–114. Springer, Berlin (2006)
49. Xie, F., Liu, H.: Unified property specification for hardware/software co-verification. In: 31st Annual International Computer Software and Applications Conference (COMPSAC 2007), vol. 1, pp. 483–490. IEEE (2007)
50. Yuan, J., Pixley, C., Aziz, A.: Constraint-Based Verification. Springer, Berlin (2006)
51. Zivkovic, C., Grimm, C., Olbrich, M., Scharf, O., Barke, E.: Hierarchical verification of AMS systems with affine arithmetic decision diagrams. TCAD **38**(10), 1785–1798 (2019)

SAT-Based Key Determination Attack for Improving the Quality Assessment of Logic Locking Mechanisms

Marcel Merten, Mohammed E. Djeridane, Sebastian Huhn, and Rolf Drechsler

1 Introduction

Nowadays, designers can benefit from access to advanced technology nodes without having the large capital expenditure of operating their own semiconductor foundries. This is thanks to the distributed manufacturing of the *integrated circuits* (ICs). However, such a distribution also yields a growing threat of compromising the integrity of once trusted IC processes by unauthorized or untrusted users [1]. During the last decade, *complementary metal-oxide-semiconductor* (CMOS)-based protection mechanisms have been the dominant technology for implementing various protection measures. However, the layout-level obfuscation by using CMOS-based camouflaging causes a significant overhead with respect to the resulting power consumption and the required area [2].

Recent research works like [1, 3, 4] have been focusing on achieving high protection while still preserving low overhead by utilizing reconfigurable silicon nanowire field-effect transistor-based polymorphic logic gates [1]. In [1], an algorithm is proposed that replaces gates impacting the original functional behavior of the circuit by reconfigurable polymorphic logic gates. Afterward, the quality of the resulting logic locking functionality is assessed by a metric based on the *Hamming distance* (HD) of the outputs over randomly applied stimuli and keys. The result is considered optimal if the HD is 50% of the maximal HD. The formal

M. Merten (✉) · M. E. Djeridane
University of Bremen, Bremen, Germany
e-mail: mar_mer@informatik.uni-bremen.de; mar_mer@uni-bremen.de; djeridam@informatik.uni-bremen.de

S. Huhn · R. Drechsler
University of Bremen, Bremen, Germany

Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
e-mail: huhn@informatik.uni-bremen.de; drechsle@informatik.uni-bremen.de

R. Drechsler, S. Huhn (eds.), *Advanced Boolean Techniques*,
https://doi.org/10.1007/978-3-031-28916-3_6

approach proposed in [5] shows the limitations of simulation-based approaches, unveiling further weaknesses in the protection mechanisms. In [5], a limited number of corrupting keys are calculated, which later be assessed. Thereby, a corrupting key is defined as a key that behaves equivalent to the correct key when considering at least one stimulus.

This work proposes a novel technique to determine corrupting keys, forming the most critical security breaches. More precisely, a framework is designed to calculate the most intimidating corrupting keys based on the concept of a SAT-based attack. In contrast to other techniques, the proposed approach calculates the corrupting keys based on *distinguishing input patterns* (DIPs), maximizing the number of equivalent behaving stimuli. This improves the quality of the assessment of potential security threats using logic locking mechanisms.

Various experiments have been conducted on the ITC'99 benchmark set. The results prove that the improved key calculation unveils weaknesses in the protection structures that remain undetected when using current approaches. The proposed technique utilizes the concept of a SAT-based attack to provide a metric for evaluating the threat of a specific corrupting key. In conclusion, the proposed approach allows a more accurate evaluation of the security of a logic locked circuit.

The remainder of this work is structured as follows: Sect. 2 briefly introduces the preliminaries of this work. Section 3 describes the proposed key determination procedure in detail. Finally, Sect. 4 presents the experimental evaluation. A conclusion and an outlook on future work are given in Sect. 5.

2 Preliminaries

Within the last decade, a lot of research has been spent on enhancing electronic systems, while the classical CMOS technology has exceeded its physical limits. Research in the field of reconfigurable technologies has gained a lot of interest since it has a great potential to realize even more complex systems. This emerging technology is a promising candidate for overcoming the constraints of Moore's law by employing polymorphic logic gates.

2.1 Reconfigurable Field-Effect Transistors

Different concepts have been proposed on realizing a device-level reconfiguration capability like RFETs. An RFET introduces a control gate that can be configured between an n-channel and p-channel behavior [2]. The reconfiguration capabilities of this new emerging technology can be used to implement new protection mechanisms, e.g., an on-chip key storage by the polymorphic logic behavior [2]. Furthermore, the RFET technology is promising to introduce effective protection mechanisms against optical reverse-engineering attacks.

A popular approach to prevent reverse engineering, even given the entire layout, is adding logic locking mechanisms to the circuit. The correct functional behavior of a circuit C is defined in Definition 1.

Definition 1 Given a circuit C, a set of appliable stimuli $\mathcal{S}$, a set of reachable states $\mathcal{F}$, and a set of possible outputs $\mathcal{P}$, the function $C : \mathcal{S} \times \mathcal{F} \rightarrow \mathcal{P}$ defines the intended functional behavior of C. In particular, $C(s, \psi)$ describes the functional behavior $\forall C, s \in \mathcal{S}, \psi \in \mathcal{F}$, with $s \in \mathcal{S}$ be a stimulus and $\psi \in \mathcal{F}$ be an internal state.

Logic locking encrypts the correct functional behavior by encrypting the circuit C using a secret key k_c. The functional behavior of a logic locked circuit is defined in Definition 2.

Definition 2 The functional behavior of a logic locked circuit C is defined given a stimulus $s \in \mathcal{S}$, an internal state $\psi \in \mathcal{F}$, and a key $k \in \mathcal{K}$. Applying the secret key k_c yields the correct functional behavior $C(s, \psi, k_c) = C(s, \psi)$.

CMOS-based approaches usually introduce XOR/XNOR key gates [6–8] or MUX gates [9–12] to obfuscate the correct functional behavior of the circuit, resulting in a huge overhead regarding the area and power consumption [1]. Figure 1 gives a basic example of an XOR gate inverting the behavior of the preceding logic when an incorrect key is applied. In the example, the locked output has the functionality of a NAND gate instead of the intended AND gate behavior.

Polymorphic logic gates like RFETs realize multiple functionalities in the same cell and, hence, are an effective way to implement a logic locking mechanism. The intended functionality is chosen by configuring a control signal. To insert key gates without the high-performance overhead of CMOS- based techniques, polymorphic logic gates can replace gates of the original circuit, meaning gates with a high impact on the primary outputs are replaced [1]. Various functionalities can be implemented by RFET-based cells like the NAND/NOR- or the XOR/XNOR-RFET. An example of the RFET visualized in Fig. 2 can be configured as an XOR or XNOR gate, depending on the control signal serving as a key bit.

2.2 Boolean Satisfiability Problem

The *Boolean satisfiability* (SAT) problem is one of the first proven NP-hard problems [13]. However, a lot of research on SAT-solving techniques has significantly

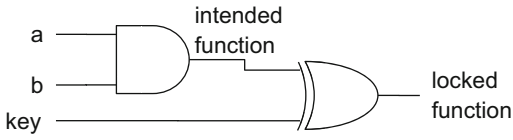


Fig. 1 Simple CMOS-based logic locking example

XOR			
a	b	key	out
0	0	0	0
0	1	0	1
1	0	0	1
1	1	0	0

XNOR			
a	b	key	out
0	0	1	1
0	1	1	0
1	0	1	0
1	1	1	1

Fig. 2 XOR/XNOR-RFET

increased the effectiveness of SAT solvers over the years. For example, *Dynamic Clause Activation* (DCA) allows to activate or deactivate a CNF Φ with an activation literal a in an extended $\Phi_a = \Phi + a$. To satisfy Φ_a, Φ only has to be satisfied if $a = 0$. Therefore, Φ can be deactivated by setting $a = 1$. The application of DCA in SAT can result in significant speedup [14].

Moreover, some modern SAT solvers are extended to solve *pseudo-Boolean* (PB) logic. PB allows defining inequalities in clauses. Furthermore, weights can be assigned to specific literals or clauses that are not required to be true in a satisfiable solution. *Pseudo-Boolean optimization* (PBO) can be used to determine a solution maximizing or minimizing the weights of a PB instance. The optimization is performed utilizing an objective function Θ. The function Θ is usually defined as a maximization or minimization of a sum of weighted literals. These PBO-based optimization techniques have been heavily orchestrated in various domains like IC testing [15].

2.3 SAT-Based Attacks

While camouflaging and logic obfuscation try to protect intellectual property from malicious misuse, attackers constantly work on techniques to remove or unlock such protection mechanisms. A popular attacking algorithm is the SAT attack first proposed in [16]. The idea of the SAT-based attack is to use SAT to unlock the circuit by determining the correct key k_c or an equivalent behaving key. First, a miter structure of two instances of the logic locked circuit is created. By solving the miter instance, a pair of keys (k_1, k_2) and a DIP D is calculated for the *primary*

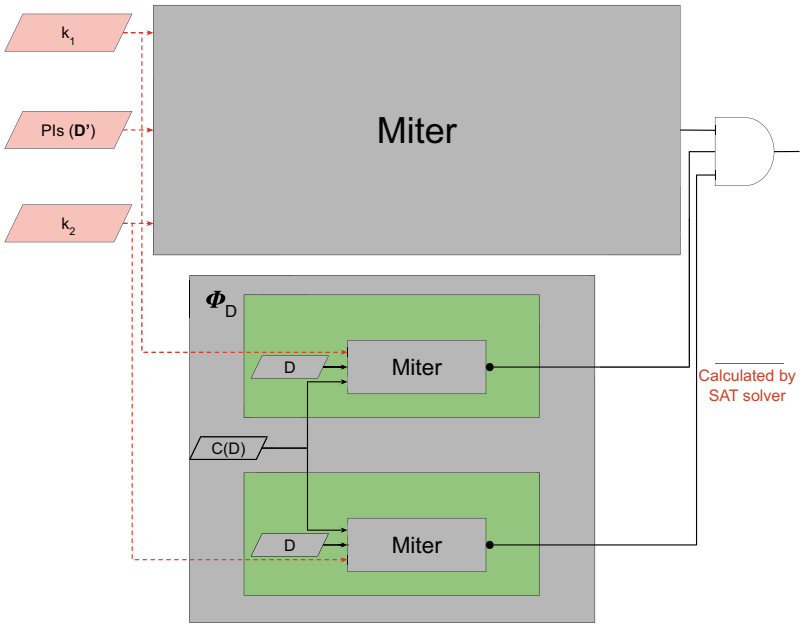


Fig. 3 Basic concept of a SAT-based attack

inputs (PIs). The DIP D is an input pattern, which results in a differing output behavior using k_1 and k_2, meaning that at least one of the output behaviors of the two compared keys is incorrect. Next, an unlocked product C of the chip is used to get the correct output behavior $C(D)$ for the D. Before the next DIP D' is calculated, the key space of k_1 and k_2 is constrained to satisfy the correct output behavior $C(D)$ for the previously calculated DIP D. This is done by adding a SAT-instance Φ_D consisting of two inverted miters. Each inverted miter forces equivalence between the logic locked circuit using $keyX$ and the oracle output $C(D)$ on the stimuli D. The basic principle of the SAT-based attack is illustrated in Fig. 3.

2.4 *Quality Assessment of RFET-Based Logic Locking Mechanisms Using Formal Methods*

This section describes the SAT-based quality assessment approach proposed in [5]. The assessment framework analyzes a *circuit under assessment* (CuA) using (RFET-based) logic locking mechanisms. First, corrupting keys—incorrect keys that result in correct functional behavior given at least one stimulus—are collected for a later assessment. A formal definition of a corrupting key k_f is given in Definition 3.

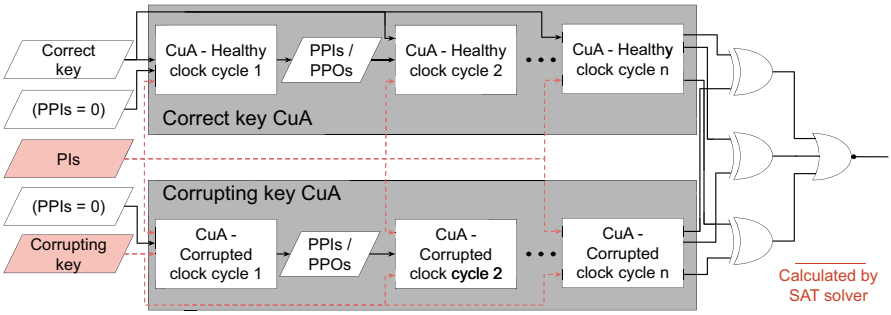


Fig. 4 Abstract model of the miter structure

Definition 3 Given a logic locked circuit C, a stimulus $s \in \mathcal{S}$, and an internal state $\psi \in \mathcal{F}$, a key $k_f \in \mathcal{K}$ is a corrupting key, iff $k_f \neq k_c$ and $\exists s, C(s, \psi, k_c) = C(s, \psi, k_f)$.

Therefore, a miter circuit is created from the *CuA* considering the correct key k_c—yielding the SAT instance Φ_{k_c}—and any incorrect key $\widehat{\mathcal{K}}$ yielding $\Phi_{\widehat{\mathcal{K}}}$. The basic principle of this construction is given in Fig. 4. The CuA is unrolled for N clock cycles to consider sequential elements. The FFs are modeled as *pseudo primary inputs* (PPIs), initialized with 0.

The entire model is stored as one SAT instance Φ_{comp} and processed by a state-of-the-art SAT solver. The inverted miter compares the unrolled Φ_{k_c} with the unrolled $\Phi_{\widehat{\mathcal{K}}}$, i.e., considering any incorrect key $k_i \neq k_c, k_i \in \widehat{\mathcal{K}}$. In particular, both states—defined by the stored FFs' values—and the primary output values can be compared for all N observed clock cycles. If a satisfiable solution is determined, i.e., a corrupting key k_f has been detected, this circumstance results in a functional equivalent of the CuA given at least one stimulus.

Next, the calculated security threat is assessed. More precisely, every determined corrupting key k_f is evaluated against possible stimuli leading to functional equivalence to the correct key k_c. More precisely, the individual corrupting key is enforced in $\Phi_{\widehat{\mathcal{K}}}$ by additional clauses. The key detection procedure—including the security threat evaluation regarding the discovered corrupting key—is repeated until Φ_{comp} is unsatisfiable or a user-defined limit has been exceeded.

3 SAT-Based Key Determination Attack

This section describes the approximative determination of the most intimidating key to improve the quality assessment of a CuA using logic locking mechanisms. The key determination procedure is divided into two parts. First, an adapted SAT-based attack is applied to collect constraints for the keys. In the second step, DCA methods are combined with the constraints to calculate a key, which forms a maximal threat to the logic locking mechanism.

To collect the constraints narrowing the key space, a miter SAT instance Φ_{DIP} is generated to calculate DIPs. Subsequently, the miter structure is constructed from the CuA while considering the a priori known *correct* key k_c yielding the SAT instance Φ_{k_c} and any incorrect key in $\widehat{\mathcal{K}}$ yielding $\Phi_{\widehat{\mathcal{K}}}$. The FFs are modeled as *pseudo primary inputs* (PPIs) in cycle $n+1$ and are connected to the corresponding *pseudo primary outputs* (PPOs) of the previous cycle n. Furthermore, similar to the SAT-based approach, the primary inputs use the same stimuli for both unrolled instances (of the CuA) and are kept constant during the unrolling. Contrary to the SAT-based assessment framework, in the attack framework, a miter is constructed to detect functional inequivalence. After the miter has been added, the key is constrained for both instances of the unrolled CuA. For Φ_{k_c}, the correct key k_c is set by adding clauses implying k_c, whereby $\Phi_{\widehat{\mathcal{K}}}$ is extended by a conflict clause excluding k_c. The entire model is stored as one SAT instance Φ_{DIP} and processed by a state-of-the-art SAT solver.

Like in the SAT-based assessment framework, the CuA is unrolled for N clock cycles since for the assessment of sequential circuits, sequential elements—meaning *flip-flops* (FFs)—have to be considered [17]. Here, the value N has to be adjusted for the CuA characteristics. The number of clock cycles required to achieve the relaxation given a stimuli of the circuit varies depending on the circuit. Similar to the approach proposed in [5], 0 is assumed as the initialization value for all FFs in cycle $n = 1$.

Next, a DIP $D \in \mathcal{D}$ is calculated, distinguishing the behavior of an arbitrary key from k_c. Similar to a SAT-based attack, a constraint is modeled as instance Φ_D of the circuit that forces the equivalence to the correct key on this specific DIP. Only one inverted miter instance is required since the correct key k_c is given. Next, a new $D' \in \mathcal{D}$ can be calculated. Like in a SAT-based attack, this procedure is repeated to narrow the search space for the keys until every remaining key results in a functional equivalent behavior (as yielded when the correct key is being applied). The algorithm terminates after the calculation of all constraints $\Phi_X, X \in \mathcal{D}$, meaning that $\Phi_D, \Phi_{D'}, \Phi_{D''}, \ldots$ constrain the corrupting key to fully unlock the circuits. The basic principle of this adapted SAT-based attack is visualized in Fig. 5.

Afterward, DCA is used to add a new activation variable $a \in \mathcal{A}$ for Φ_D, such that $\Phi_{Da} = \Phi_D + a$. Next, Φ_{Da} is added to the final key determination problem instance Φ_{KD}, so that $\Phi_{kd} = \Phi_{kd} * \Phi_{Da}$. By assuming $a = 0$, $\Phi_{Da} = \Phi_D$ and, hence equivalence to the correct key on this specific DIP is forced. Now, the next DIP D' can be calculated. In Fig. 6 a complete key determination instance $\forall D, D' \in \mathcal{D}$, and $\forall a, a' \in \mathcal{A}$ is illustrated.

Once Φ_{KD} is complete, containing all the activatable Φ_{Da}, the most intimidating key is determined. First, the weight $w(a) = -1$ is assigned for every activation signal $a \in A$. PBO-solving techniques are utilized to determine the most intimidating key. In particular, an objective function θ, defined in Eq. 1, is used to maximize the number of activated instances Φ_D. Therefore, the PBO solver increases the functional equivalence to the correct behavior on the calculated DIPs:

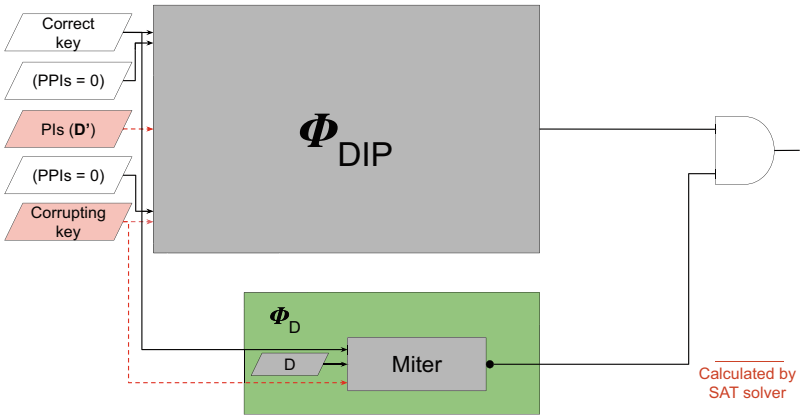


Fig. 5 Adapted SAT-based attack to collect constraints for the key space

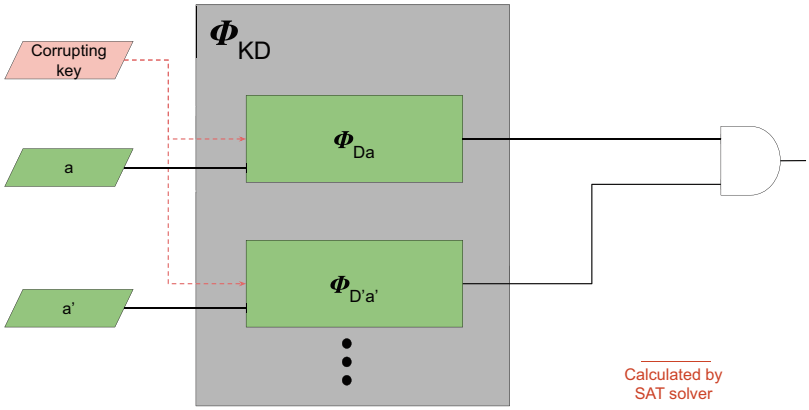


Fig. 6 Complete key determination instance including all activatable key-space constraints

$$\theta = \max\left(\sum_{a \in A}(w(a))\right) \tag{1}$$

A corrupting key is calculated by solving the problem instance $\Phi_{KD} * \theta$ that satisfies the functional equivalence to k_c on the maximum number of DIPs. The DIPs of a SAT attack are iteratively narrowing the search space of the keys to find a key that unlocks the circuit. Therefore, a key is considered a most intimidating corrupting key that satisfies the maximum number of constraints as given by the DIPs.Consequently, $\Phi_{KD} * \theta$ is solved for a predefined number of keys, which will be assessed afterward.

The assessment of the detected keys can be performed with an arbitrary assessment technique, for example, the HD approach or the approach proposed in [5].

4 Experimental Evaluation

This section describes the experimental evaluation of the proposed technique to determine corrupting keys during the quality assessment of logic locking mechanisms. The experiments compare the novel approach with the determination of corrupting keys proposed in [5], which are used as the baseline. For the assessment of the detected keys, the assessment framework defined in [5] is used for both key determination approaches.

All experiments have been executed on an AMD 4750U processor with 32 GB system memory. All implementations are solely in C++. For the evaluation, different benchmark circuits of the ITC'99 benchmark suite are considered. For each of these circuits, 15 of the *NOR*, *NAND*, *XOR*, and *XNOR* gates have been randomly replaced by RFETs, while the functional behavior is retained if the correct key is applied. Experimental evaluations have shown that 15 RFETs can be considered a sufficient number of key gates to create logic locking structures with weaknesses that are nontrivial to analyze and, hence, hard to detect. Consequently, each circuit has 15 control signals resulting in $2^{15} = 32{,}768$ possible keys. Similar to the results in [5], the 1024 most intimidating keys are assessed on both the proposed and the baseline approaches. Furthermore, up to 1024 stimuli with functionally correct behaving *primary outputs* (POs) (per corrupting key k_f) are captured.

The FFs of the CuA are initialized with 0, and the stimuli are kept constant over all five clock cycles. Each circuit has been unrolled for five clock cycles since it has been proven as an appropriate parameter to cover the functional behavior's majority of the considered benchmark circuits [18].

Table 1 shows the detailed results of the two approaches for determining the corrupting keys. It illustrates the number of detected corrupting keys, their minimum, the average and maximum number of corrupted stimuli for the novel SAT-based key determination approach, and the baseline approach proposed in [5]. Furthermore, the number of calculated DIPs for the SAT-attack-based approach is shown.

For the b05, b07, b08, b09, b12, b14, and b15, the results are equivalent regarding the number of detected corrupting keys and corrupted stimuli. However, in the case of the b05, the proposed approach shows that no DIP can be calculated, meaning all 32,768 keys are behaving equivalent. This provides additional information about the poor quality of the underlying logic locking mechanism. Considering the circuits, b06, b10, b11, and b13, the novel approach shows that there are more critical corrupting keys than those ones as detected by the baseline approach. In fact, the baseline key collection algorithm can lead to a major misjudgment of the quality of a logic locking mechanism.

Figure 7 presents the number of activated constraints for the corrupting keys when using the SAT-based key determination technique. The diagram shares further information about the actual equivalence of the corrupting keys to the correct behavior. For example, in the case of the b17, 7 corrupting keys are able to fully unlock the circuit, while 80 corrupting keys can satisfy 2 constraints and 712 corrupting keys are able to satisfy 1 constraint. In the case of the b10, all 1024

Table 1 Results—15 key bits

	SAT-attack-based approach					Baseline approach [5]			
			#stimuli				#stimuli		
Circuit	DIPs	$\#\{k_c\}$	Minimum	Average	Maximum	$\#\{k_c\}$	Minimum	Average	Maximum
b05	0	–	–	–	–	1024	2	2	2
b06	2	1024	4	4	4	1024	2	3	4
b07	1	3	2	2	2	3	2	2	2
b08	2	63	256	256	256	63	256	256	256
b09	1	1	2	2	2	1	2	2	2
b10	2	1024	1024	1024	1024	1024	256	256	512
b11	1	1024	128	128	128	1024	126	126	128
b12	1	1024	32	32	32	1024	32	32	32
b13	1	1024	1024	1024	1024	1024	512	768	1024
b14	2	1024	1024	1024	1024	1024	1024	1024	1024
b15	2	1024	1024	1024	1024	1024	1024	1024	1024
b17	3	1024	1024	1024	1024	1024	1024	1024	1024
b20	1	0	0	0	0	0	0	0	0
b21	1	0	0	0	0	0	0	0	0
b22	1	0	0	0	0	0	0	0	0

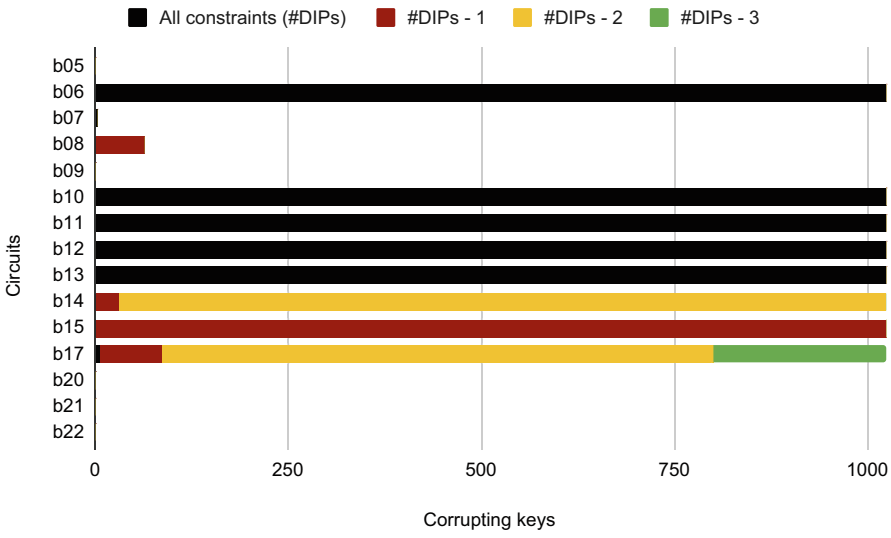


Fig. 7 Number of DIPs with correct functional behavior for corrupting keys

assessed keys are able to satisfy the equivalent behavior to the correct key on both calculated DIPs. Therefore, at least 1024 corrupting keys exist that fully unlock the circuit's functional behavior resulting in an unbearable security breach. On the other hand, the results for the b14 and b15 show that no corrupting key fully unlocking the circuit's functional behavior exists.

This clearly shows that the novel approach outperforms other approaches of determining the most intimidating corrupting keys, providing a more appropriate quality assessment of logic locking mechanisms.

5 Conclusions

This paper presented a novel method of calculating the most intimidating corrupting keys for logic locking mechanisms. In the end, the proposed technique allows determining keys, which form an enormous security threat, by adapting the conceptual structure of SAT-based attacks and enhancing the idea with PBO techniques. The resulting metric ensures the detection of potential security breaches and outperforms the existing key determination mechanisms. Future work will investigate a sophisticated weight calculation for the activation signals to prefer activating instances Φ_D, implying that the most equivalent functional behavior is achieved.

Acknowledgments This work was financially supported by the German Federal Ministry of Education and Research BMBF under the framework of VE-CirroStrato and the AI initiative of the Free Hanseatic City of Bremen. We would like to thank Verific Design Automation Inc. for providing the SystemVerilog frontend used for the implementation of our technique.

References

1. Alasad, Q., Yuan, J.-S., Bi, Y.: Logic locking using hybrid CMOS and emerging SiNW FETs. Electronics **6**(3), 69 (2017)
2. Rai, S., Srinivasa, S., Cadareanu, P., Yin, X., Hu, X.S., Gaillardon, P.-E., Narayanan, V., Kumar, A.: Emerging reconfigurable nanotechnologies: can they support future electronics? In: IEEE/ACM International Conference on CAD, pp. 1–8 (2018)
3. Alasad, Q., Yuan, J.: Logic obfuscation against IC reverse engineering attacks using PLGs. In: IEEE International Conference on Computer Design, pp. 341–344 (2017)
4. Alasad, Q., Yuan, J.-S., Subramanyan, P.: Strong logic obfuscation with low overhead against IC reverse engineering attacks. IEEE Trans. CAD Integr. Circuits Syst. **25**(4), 1–31 (2020)
5. Merten, M., Huhn, S., Drechsler, R.: Quality assessment of RFET-based logic locking protection mechanisms using formal methods. In: IEEE European Test Conference (ETS), pp. 1–2 (2022)
6. Roy, J.A., Koushanfar, F., Markov, I.L.: EPIC: ending piracy of integrated circuits. In: Design, Automation and Test in Europe, pp. 1069–1074 (2008)
7. Rajendran, J., Pino, Y., Sinanoglu, O., Karri, R.: Security analysis of logic obfuscation. In: Design Automation Conference, pp. 83–89 (2012)
8. Rajendran, J., Zhang, H., Zhang, C., Rose, G.S. Pino, Y., Sinanoglu, O., Karri, R.: Fault analysis-based logic encryption. IEEE Trans. Comput. **64**(2), 410–424 (2015)
9. Alasad, Q., Bi, Y., Yuan, J.-S.: E_2LEMI: energy-efficient logic encryption using multiplexer insertion. Electronics **6**, 1–16 (2017)
10. Wendt, J.B., Potkonjak, M.: Hardware obfuscation using PUF-based logic. In: IEEE/ACM International Conference on CAD, pp. 270–271 (2014)

11. Plaza, S.M., Markov, I.L.: Solving the third-shift problem in IC piracy with test-aware logic locking. IEEE Trans. CAD Integr. Circuits Syst. **34**(6), 961–971 (2015)
12. Lee, Y., Touba, N.: Improving logic obfuscation via logic cone analysis (2015)
13. Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of the Third Annual ACM Symposium on Theory of Computing, pp. 151–158. Association for Computing Machinery (1971)
14. Eggersgluss, S., Drechsler, R.: Increasing robustness of SAT-based delay test generation using efficient dynamic learning techniques. In: IEEE European Test Symposium, pp. 81–86 (2009)
15. Huhn, S., Eggersglüß, S., Chakrabarty, K., Drechsler, R.: Optimization of retargeting for IEEE 1149.1 TAP controllers with embedded compression. In: Design, Automation and Test in Europe Conference and Exhibition, 2017, pp. 578–583 (2017)
16. Subramanyan, P., Ray, S., Malik, S.: Evaluating the security of logic encryption algorithms. In: IEEE International Symposium on Hardware Oriented Security and Trust, pp. 137–143 (2015)
17. Arora, R., Hsiao, M.: Enhancing SAT-based bounded model checking using sequential logic implications. In: International Conference on VLSI Design, pp. 784–787 (2004)
18. Finder, A., Sülflow, A., Fey, G.: Latency analysis for sequential circuits. In: IEEE European Test Symposium, pp. 129–134 (2011)

Autosymmetric and D-reducible Functions: Theory and Application to Security

Anna Bernasconi, Valentina Ciriani, and Licia Monfrini

1 Introduction

The multiplicative complexity of a Boolean function f is defined as the minimum number of AND gates that are necessary and sufficient to represent f with a circuit, using the two-input Boolean operators AND and XOR, and the negation (NOT). The basis {AND, XOR, NOT} is widely used to represent Boolean functions in cryptographic applications [7, 8, 14–16], where the multiplicative complexity plays a crucial role. In particular, the minimization of the number of AND gates is important for high-level cryptography protocols such as zero-knowledge protocols and secure two-party computation, where processing AND gates is more expensive than processing XOR gates [1]. Moreover, the multiplicative complexity is an indicator of the degree of vulnerability of the circuits, as a small number of AND gates in an {AND, XOR, NOT} circuit indicate a high vulnerability to algebraic attacks [8, 10, 16]. However, determining the multiplicative complexity of a Boolean function f is a computationally intractable problem [8]. Therefore, the minimization of the number of AND gates, in circuits composed by the gates {AND, XOR, NOT}, is important in order to estimate the multiplicative complexity of the function. For this purpose, Boolean functions can be represented exploiting *XOR-And Graphs* (XAGs) [11, 14, 15], and the multiplicative complexity of an XAG implementation of a Boolean function can be used to provide an upper bound for its real multiplicative complexity.

A. Bernasconi
Dipartimento di Informatica, Università di Pisa, Pisa, Italy
e-mail: anna.bernasconi@unipi.it

V. Ciriani (✉) · L. Monfrini
Dipartimento di Informatica, Università degli Studi di Milano, Milano, Italy
e-mail: valentina.ciriani@unimi.it; licia.monfrini@studenti.unimi.it

R. Drechsler, S. Huhn (eds.), *Advanced Boolean Techniques*,
https://doi.org/10.1007/978-3-031-28916-3_7

The "regularities" of Boolean functions are often exploited for deriving, in shorter synthesis time, more compact circuits. In the literature, some structural regularities of Boolean functions have been studied, i.e., autosymmetry [5, 6, 13] and D-reducibility [4]. These regularities are based on the notion of affine spaces and are easily expressed using XOR gates. Thus, both these structural regularities can be exploited for decreasing the multiplicative complexity of an XAG, and to better estimate the multiplicative complexity of the function. In the literature [3] a study of the multiplicative complexity of autosymmetric functions and a study of the multiplicative complexity of D-reducible functions are proposed. Moreover, experimental results show that about the 9% of these regular functions are both autosymmetric and D-reducible.

In this paper, we further investigate on regular functions that are both autosymmetric and D-reducible. In particular, we give a formal characterization of completely specified autosymmetric and D-reducible functions. Moreover, we study the case of non-completely specified functions. Finally, we discuss the multiplicative complexity of functions that are both autosymmetric and D-reducible. The experimental results show that, for functions that are both autosymmetric and D-reducible, we get a better estimate of the multiplicative complexity in about 27% of the cases with respect to exploiting autosymmetry or D-reducibility only, with an average reduction of the number of ANDs of about 27%.

2 Preliminaries

In this section, we review the definitions and properties of autosymmetric and D-reducible functions, and we introduce our running example. Finally, at the end of the section, we give a very brief introduction to multiplicative complexity and XOR-AND Graphs (XAGs). Hereafter, we will consider Boolean functions over n variables (i.e., described in the Boolean space $\{0, 1\}^n$).

2.1 Autosymmetric Functions

In this section, we introduce a particular regularity, i.e., autosymmetry [5, 6, 13], based on affine spaces.

Intuitively, a Boolean function f over n variables is *k-autosymmetric* if it can be projected onto a smaller function f_k that depends on $n - k$ variables. The regularity of a Boolean function f is then measured computing its *autosymmetry degree* k, with $0 \leq k \leq n$, where $k = 0$ means no regularity. For $k \geq 1$ the Boolean function f is said to be *autosymmetric*, and a new function f_k depending on $n - k$ variables only, called the *restriction* of f, is identified. Moreover, an expression for f can be simply built from f_k: $f(x_1, x_2, \ldots, x_n) = f_k(y_1, y_2, \ldots, y_{n-k})$, where f_k is a Boolean function on $n - k$ variables $y_1 = \oplus(X_1), y_2 = \oplus(X_2), \ldots, y_{n-k} =$

$\oplus(X_{n-k})$ and each $\oplus(X_i)$ is a XOR whose input is a set of variables X_i with $X_i \subseteq \{x_1, x_2, \ldots, x_n\}$. Note that $\oplus(X_i)$ can be a single variable, i.e., $X_i = \{x_j\}$ and $\oplus(X_i) = x_j$. The autosymmetry test consists of finding the value of k, the restriction f_k, and each single XOR with its input variables X_i (reduction equations). Note that a degenerate function, i.e., a function that does not depend on all the variables, is autosymmetric. The computational time of the autosymmetry test is polynomial in the size of the ROBDD representation of f [5].

The restriction f_k is "equivalent" to, but smaller than f, and has $|S(f)|/2^k$ minterms only, where $S(f)$ denotes the support of f, and thus $|S(f)|$ is the number of minterms of f. Each point of f_k in $\{0, 1\}^{n-k}$ corresponds to a set of 2^k points in $\{0, 1\}^n$ where f assumes the same value. The function f can be synthesized through the synthesis of its restriction f_k. As the new $n-k$ variables are XOR combinations of some of the original ones, the reconstruction of f from f_k can be obtained with an additional logic level of XOR gates, whose inputs are the original variables, and the outputs are the new $n-k$ variables given as inputs to a circuit for f_k. In general, the restricted function f_k can be synthesized in any framework of logic minimization. In this paper we derive an XAG representation of it.

We now recall some properties of autosymmetric functions and of their restrictions, which will be useful for the analysis of their multiplicative complexity. As shown in [5, 6], any k-autosymmetric function f is associated with a k-dimensional vector space L_f, defined as the set of all minterms α s.t. $f(x) = f(x \oplus \alpha)$ for all $x \in \{0, 1\}^n$. Let L_f be sorted in increasing binary order, with the vectors indexed from 0 to $2^k - 1$. The set of vectors of L_f with indices $2^0, 2^1, \ldots, 2^{k-1}$ is called the *canonical basis* B_L of L_f. The k variables that are truly independent onto L_f are called *canonical variables*, while the other variables are called *non-canonical*. Informally, the canonical variables are the ones that assume all the possible combinations of $\{0, 1\}$ values in the vectors of the vector space L_f, meanwhile the non-canonical variables are the variables that, on L_f, have a constant value or are a linear combination of the canonical ones.

The canonical variables can be easily computed from the canonical basis $v_1, \ldots, v_k$, in the following way: for each v_i, let x be the variable corresponding to the first 1-component from left of v_i. The variable x is a *canonical variable*.

Finally, the restriction f_k corresponds to the projection of f onto the subspace $\{0, 1\}^{n-k}$ where all the canonical variables assume value 0, while the reduction equations correspond to the linear combinations that define each non-canonical variable in terms of the canonical ones (see [5, 6] for more details).

Example 1 Given an arbitrary function f, the vector space L_f provides the essential information to compute the autosymmetry degree, the restriction f_k, and the reduction equations of f. Consider, for instance, the completely specified Boolean function $f(x_1, \ldots, x_5)$ described by its minterms as follows: $f = \{00001, 00100, 00110, 01000, 01010, 01101, 10001, 10011, 10100, 11000, 11101, 11111\}$. The function f can be represented by the Karnaugh map depicted in Fig. 1. The "regularity" of the function is highlighted by the colors in the figure. The computation of the vector space L_f and of the reduction equations is not straightforward; we

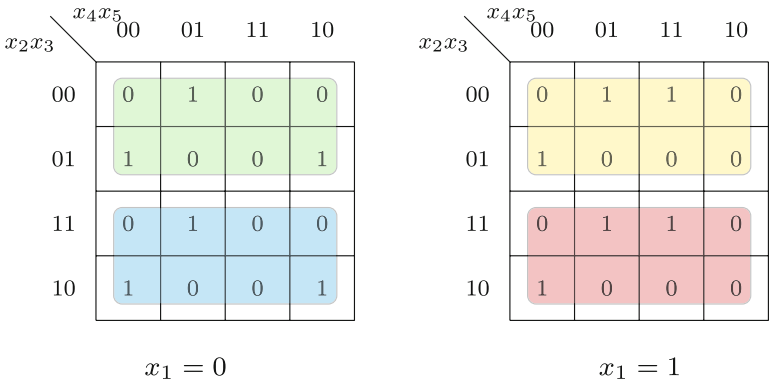


Fig. 1 Karnaugh map for the running example (function f), the colors highlight the autosymmetry regularity

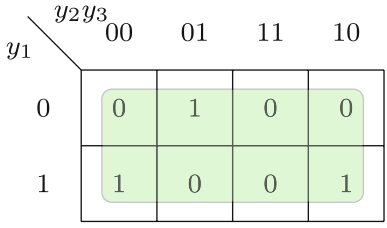


Fig. 2 Karnaugh map for the reduced function f_2 of the 2-autosymmetric function shown in Fig. 1

refer the reader to [5] for the complete algorithm. The vector space L_f associated with f is $L_f = \{00000, 01100, 10101, 11001\}$. In fact, for any element $\alpha \in L_f$, we have that $f(x) = f(x \oplus \alpha)$ for all $x \in \{0, 1\}^n$. We have that $k = \log_2 |L_f| = 2$; thus, f is 2-*autosymmetric*. The canonical basis is $B_V = \{01100, 10101\}$. The canonical variables are x_1 and x_2 (i.e., the variables that correspond to the first ones from left in the two vectors of the canonical base). The remaining variables x_3, x_4, and x_5 are non-canonical. The restriction f_2, depicted in Fig. 2, can be computed starting from the subset of minterms $\{00001, 00100, 00110\}$ of f, where all the canonical variables are equal to 0. In fact, if we project these points in the space $\{0, 1\}^3$, corresponding to the non-canonical variables x_3, x_4, and x_5, we get $f_2(y_1, y_2, y_3) = \{001, 100, 110\}$. Finally, the reduction equations for reconstructing the original function f are [5]: $y_1 = x_1 \oplus x_2 \oplus x_3$; $y_2 = x_4$; $y_3 = x_1 \oplus x_5$.

Autosymmetric functions are just a subset of all Boolean functions. Indeed, while the number of the Boolean functions of n variables is 2^{2^n}, the number of autosymmetric ones is $(2^n - 1)2^{2^{n-1}}$ [6]. Therefore, the set of autosymmetric functions is much smaller than the one containing all the Boolean functions. Nevertheless, a considerable amount of standard Boolean functions of practical interest falls in this class. Indeed, about 24% of the functions in the classical ESPRESSO benchmark suite [17] have at least one truly (i.e., nondegenerate) autosymmetric output [5, 6]. Thus, the interest on autosymmetric functions is motivated by (1) their compact (in

terms of number of AND gates) representation, which consists of an XOR layer that is the input to an XAG for the restriction, and (2) the frequency of autosymmetric functions in the set of benchmark functions.

2.2 D-reducible Functions

In this section, we summarize the definitions and the major properties of dimension-reducible Boolean functions, i.e., D-reducible functions. We recall that the Boolean space $\{0,1\}^n$ is a vector space with respect to the exclusive sum $\oplus$ and the multiplication with the scalars 0 and 1. Moreover, an affine space is a vector space or a translation of a vector space [4], more precisely: let V be vector subspace of the Boolean vector space $(\{0,1\}^n, \oplus)$ and w be a point in $\{0,1\}^n$, then the set $A = w \oplus V = \{w \oplus v \mid v \in V\}$ is an *affine space* over V with *translation point* w. The space V is called the *vector space associated* with A. Finally, a Boolean function $f : \{0,1\}^n \to \{0,1\}$ is *D-reducible* if $f \subseteq A$, where $A \subset \{0,1\}^n$ is an affine space of dimension strictly smaller than n.

The minimal affine space A containing a D-reducible function f is unique, and it is called the *associated affine space* of f. The function f can be represented as $f = \chi_A \cdot f_A$, where $f_A \subseteq \{0,1\}^{\dim A}$ is the projection of f onto A and χ_A is the characteristic function of A. Observe that the smallest affine space contains the whole on-set of a function f. Thus, this regularity is different from autosymmetry, since the numbers of minterms of the original function f and of the projected function f_A are equal to each other. Moreover, as shown in [9], an affine space can be represented by a simple expression, consisting of an AND of XORs or literals. In particular, an affine space of dimension $\dim A$ can be represented by an expression containing $(n - \dim A)$ XOR factors.

The D-reducibility of a function f can be exploited in the minimization process. The projection f_A is minimized instead of f. This approach requires two steps: first, deriving the affine space A and the projection f_A and then minimizing f_A in any logic framework (e.g., XAG). The D-reducibility test [4], which establishes whether a function f is D-reducible, and the computation of A can be performed efficiently exploiting the Gauss-Jordan elimination procedure [12], which is used to find the on-set minterms of f that are linearly independent.

Example 2 Let us consider the running example, analyzed for autosymmetry, i.e., the function f shown in Fig. 3. The minimal affine space A containing all the minterms the function f is highlighted by the color cyan in the figure. Thus, A is a four-dimension affine space. The canonical basis of the vector space V associated with A is $\{00010, 00101, 01001, 10000\}$; its canonical variables are x_1, x_2, x_3, and x_4, while x_5 is non-canonical. The representation, as an AND of XORs, of A is $x_2 \oplus x_3 \oplus x_5$. Moreover, the projection of f onto the affine space A is $f_A = \{0000, 0010, 0011, 0100, 0101, 0110, 1000, 1001, 1010, 1100, 1110, 1111\}$. The projection f_A is represented in the Karnaugh map in Fig. 4.

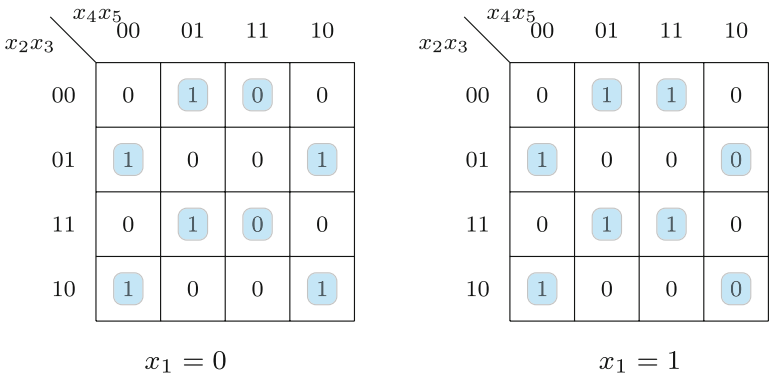
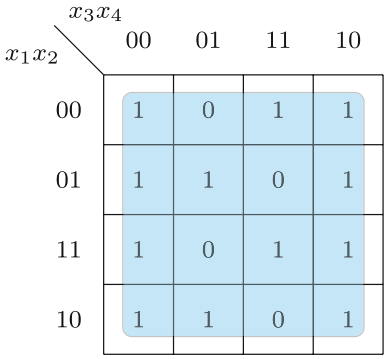


Fig. 3 Karnaugh map for the D-reducible function f. The space A of f is highlighted

Fig. 4 Karnaugh map for the projection f_A of the D-reducible function f shown in Fig. 3



2.3 Multiplicative Complexity and XOR-AND Graphs

The *multiplicative complexity* $M(f)$ *of a Boolean function* f is a complexity measure defined as the number of AND gates, with fan-in 2, which are necessary and sufficient to implement f with a circuit over the basis {AND, XOR, NOT}. Moreover, the *multiplicative complexity* $M_C(f)$ *of a circuit* C implementing a Boolean function f over the basis {AND, XOR, NOT} is the actual number of AND gates in C. Therefore, the multiplicative complexity of a circuit for f only provides an upper bound for the multiplicative complexity of f, i.e., $M(f) \leq M_C(f)$. In this work, we consider Boolean functions represented in *XOR-AND graph* (XAG) form [11, 14, 15], which are logic networks that contain only binary XOR nodes, binary AND nodes, and inverters. In particular, we refer to the XAG model described in [14], where regular and complemented edges are used to connect the gates. Complemented edges indicate the inversion of the signals and replace inverters in the network.

3 Completely Specified Autosymmetric and D-reducible Functions

A Boolean function f, which is D-reducible and autosymmetric at the same time, can be decomposed in two different ways. The first possibility is to apply the D-reducibility decomposition, and represent f as $f = \chi_A f_A$, and then to apply the autosymmetry reduction to f_A. The second possibility consists in decomposing the function f applying the autosymmetry test and deriving the restriction f_k, and then applying the D-reducibility decomposition to f_k. In this section, we prove that if f is a completely specified function, these two strategies provide the same final representation of the function f.

We first recall from [3] a theoretical result contained in the proof of a theorem, used to prove our results. For this reason, we report it as a lemma, and we recall here its proof.

Lemma 1 ([3]) *Let f be an autosymmetric function with associated linear space L_f. Let f also be a D-reducible function contained in the affine space A. Then, $L_f \subseteq V$, where V is the vector space associated with A.*

Proof First of all, we observe that the vector space L_f is a subspace of the vector space V associated with A. Let $\alpha \in L_f$, and let x be any on-set minterm of f. Then, $f(x \oplus \alpha) = f(x) = 1$, and therefore both x and $x \oplus \alpha \in A$. This in turns implies that $\alpha \in (x \oplus A)$, i.e., $\alpha \in V$, since $x \oplus A = V$ for any $x \in A$ (we refer the reader to [9] for more details on affine spaces and their properties). □

Example 3 Let us consider the function f described in Figs. 1 and 3. In the previous examples, we have shown that f is both autosymmetric and D-reducible. Example 1 shows that $L_f = \{00000, 01100, 10101, 11001\}$, and from the Fig. 3 of Example 2, we have that $A = \{00001, 00011, 00100, 00110, 01000, 01010, 01101, 01111, 10001, 10011, 10100, 10110, 11000, 11010, 11101, 11111\}$. The corresponding vector space is computed as $V = v \oplus A$ where v is any vector contained in A. Thus, if we pick $v = 00001$ and computing $V = 00001 \oplus A$, we obtain: $V = \{00000, 00010, 00101, 00111, 01001, 01011, 01100, 01110, 10000, 10010, 10101, 10111, 11001, 11011, 11100, 11110\}$. (Notice that we can use any v in A and we would obtain the same associated vector V.) We can easily verify that $L_f \subseteq V$.

Let k denote the dimension of L_f and a be the dimension of the vector space V associated with A. The dimension of an affine space A is defined as the dimension of the corresponding vector space V.

Proposition 1 *The dimension of L_f is less or equal to the dimension of A, and the canonical variables of V include all the canonical variables of L_f.*

Proof The first part of the proposition immediately follows from Lemma 1.

For the second part, observe that, since $L_f \in V$, we can construct a basis for V extending a basis for L_f. Each vector in a basis for L_f corresponds to a canonical variable of L_f, and consequently to a canonical variable of V. The remaining

$a - k$ canonical variables of V can be derived from the remaining $a - k$ linearly independent vectors in the basis of V. □

As a consequence, we have the following corollary.

Corollary 1 *The $n - k$ non-canonical variables of L_f include the $n - a$ non-canonical variables of V.*

Example 4 Let us consider the running example. Example 1 shows that the canonical variables of L_f are x_1 and x_2, and Example 2 shows that the canonical variables of the vector space V associated with A are x_1, x_2, x_3, and x_4. In this running example, we have that the function is k-autosymmetric with $k = 2$ and that $a = 4$. Moreover, the non-canonical variables of L_f are x_3, x_4, and x_5. The non-canonical variable of the vector space V associated with A is x_5. We can verify that the $n - k = 5 - 2 = 3$ non-canonical variables of L_f contain the $n - a = 5 - 4 = 1$ non-canonical variable of V.

For completeness, we recall from [3] a theorem stating that if we first apply the D-reducibility decomposition, we do not loose the autosymmetry property of the function.

Theorem 1 ([3]) *Let f be a completely specified k-autosymmetric Boolean function depending on n binary variables. If f is D-reducible with associate affine space A, then the projection f_A of f onto A is k-autosymmetric.*

In order to prove that the two decomposition strategies provide the same final representation of f, we need to prove that the restriction f_k of an autosymmetric function preserves the D-reducibility property, as shown in the following theorem.

Theorem 2 *Let f be a D-reducible completely specified Boolean function depending on n binary variables, and with associate affine space A. If f is k-autosymmetric, then the restriction f_k of f is D-reducible with respect to the same affine space A.*

Proof First of all, we notice that the reduction f_k is the result of a projection of f onto a $(n - k)$-dimensional space, where each point of f_k in $\{0, 1\}^{n-k}$ corresponds to a set of 2^k points in $\{0, 1\}^n$ where f assumes the same value (as reviewed in Sect. 2.1).

We now show that f_k is D-reducible in $\{0, 1\}^{n-k}$, where it is described by the variables y_i corresponding to the non-canonical variables of L_f, and defined by the reduction equations. Observe that the on-set minterms of f_k, and the corresponding minterms in the original space $\{0, 1\}^n$, are obviously covered by A. Moreover, recall that f_k is derived by f assigning value 0 to all the canonical variables of L_f, and renaming the non-canonical variables with $y_1, \ldots, y_{n-k}$. If we now assign value 0 to the occurrences of the k canonical variables of L_f in χ_A, and we rename the non-canonical variables of L_f as $y_1, \ldots, y_{n-k}$, we obtain the characteristic function of an $a - k$ dimensional subspace A' of A that covers f_k in $\{0, 1\}^{n-k}$. Therefore, f_k is D-reducible and can be studied in a subspace of dimension $a - k$ represented by a product of $(n - k) - (a - k) = n - a$ EXOR factors, i.e.,

$$f_k = \chi_{A'} f_{kA'} ,$$

where $f_{kA'}$ depends on $a - k$ variables.

Replacing the variables $y_1, \ldots, y_{n-k}$ in both $\chi_{A'}$ and $f_{kA'}$ with the corresponding reduction equations, we derive a representation of f as

$$f = \chi_A \, f_{kA} .$$

Observe that the affine space associated with f and f_k is the same. □

In summary, we have shown how to decompose the function f with two different strategies. If we first apply the D-reducibility decomposition, and then exploit the autosymmetry property on f_A, we obtain $f = \chi_A f_{Ak}$. If, vice versa, we first exploit the autosymmetry of f, and then we decompose the restriction f_k using the D-reducibility property, we get $f = \chi_A f_{kA}$. Observe that both functions f_{Ak} and f_{kA} depend on the same $a - k$ variables. Finally, we have the following theorem, which immediately follows from Theorems 1 and 2, and from the fact that $f = \chi_A f_{Ak} = \chi_A f_{kA}$.

Theorem 3 *The two decompositions are equivalent, i.e.,* $f_{Ak} = f_{kA}$.

The following examples show the two possible strategies implemented on the running example.

Example 5 (Autosymmetry—D-reducibility) Let us consider the running example. Now, we first apply autosymmetry and then D-reducibility to the given function f. Let us consider the function f described in Fig. 1. Example 1 shows that f is 2-autosymmetric and it computes the restriction f_2 as the set of minterms $f_2(y_1, y_2, y_3) = \{001, 100, 110\}$ in $\{0, 1\}^3$. We now compute the D-reducibility decomposition of f_2. The Karnaugh map for f_2 is shown on the left side of Fig. 5 where the affine space A, which entirely contains f_2, is highlighted in cyan. The function f_2 can be projected in A obtaining the Boolean function $f_{2A}(y_1, y_2) = \{00, 01, 11\}$ depicted in the Karnaugh map on the right side of Fig. 5. The characteristic function of A is $(y_1 \oplus y_3)$. In order to simply describe our example, we represent the function f_{2A} in SOP form (i.e., $f_{2A} = (\overline{y}_2 + y_1)$). Recall that f_{2A} can be represented in any form and that we will use the XAG representation

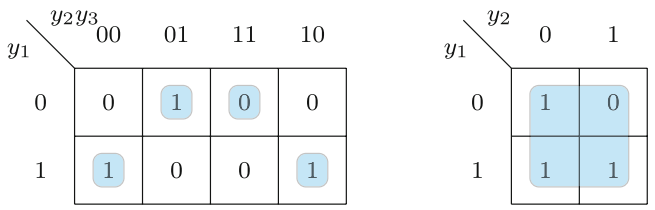


Fig. 5 Left side: Karnaugh map for $f_2(y_1, y_2, y_3)$. The space A of f is highlighted in cyan. Right side: Karnaugh map for $f_{2A}(y_1, y_2)$

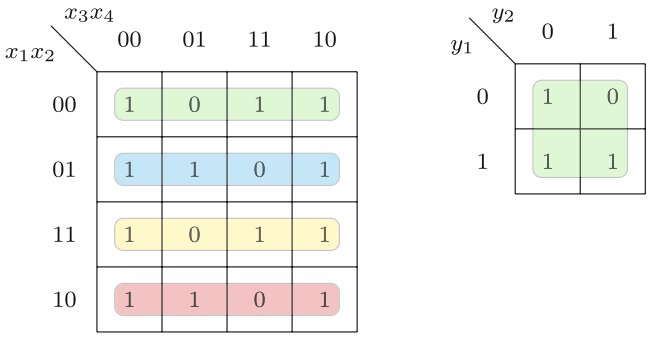


Fig. 6 Left side: Karnaugh map for the function $f_A(x_2, x_3, x_4, x_5)$. Right side: Karnaugh map for $f_{A2}(y_1, y_2)$

in the experimental section. In summary, we have that $f_2(y_1, y_2, y_3) = \chi_A \cdot f_{2A} = (y_1 \oplus y_3)(\overline{y}_2 + y_1)$. In order to reconstruct the original function f, we replace the variables y_1, y_2, and y_3 with the corresponding reduction equations computed in Example 1. We have $f(x_1, \ldots, x_5) = [(x_1 \oplus x_2 \oplus x_3) \oplus (x_1 \oplus x_5)] \cdot [\overline{x}_4 + (x_1 \oplus x_2 \oplus x_3)]$, which can be simplified. We finally obtain:

$$f(x_1, \ldots, x_5) = \chi_A \cdot f_{2A} = (x_2 \oplus x_3 \oplus x_5) \cdot [\overline{x}_4 + (x_1 \oplus x_2 \oplus x_3)] .$$

Example 6 (D-reducibility-Autosymmetry) Let us consider again the running example. In this case, we first apply D-reducibility and then autosymmetry to the given function f. Let us consider the function f described in Fig. 3. Example 2 shows that f is D-reducible, and the projection $f_A(x_2, x_3, x_4, x_5)$ is shown in Fig. 4: $f_A = \{0000, 0010, 0011, 0100, 0101, 0110, 1000, 1001, 1010, 1100, 1110, 1111\}$. We now compute the autosymmetry decomposition of f_A. The Karnaugh map for f_A is depicted on the left side of Fig. 6. The projection f_A is autosymmetric, and its associated vector space is $L_{f_A} = \{0000, 0110, 1010, 1100\}$. This space has dimension $k = \log_2 |L_{f_A}| = 2$; thus, f_A in Fig. 6 is 2-*autosymmetric*. The canonical basis is $\{0110, 1010\}$ and the canonical variables are x_1 and x_2. Thus, the non-canonical variables are x_3 and x_4. We can now compute the restriction f_{A2} using the subset $\{0000, 0010, 0011\}$ of the minterms of f_A that have the canonical variables set to 0. If we project such minterms into the Boolean space $\{0, 1\}^2$ of the variable x_3 and x_4, we obtain the function $f_{A2}(y_1, y_2) = \{00, 10, 11\}$ depicted in the Karnaugh map on the right-hand side of Fig. 6. The corresponding reduction equations are: $y_1 = x_1 \oplus x_2 \oplus x_3$; $y_2 = x_4$. A SOP form for the function f_{A2} is $SOP(f_{A2}) = \overline{y}_2 + y_1$. Applying the reduction equations, we have that $\overline{y}_2 + y_1 = \overline{x}_4 + (x_1 \oplus x_2 \oplus x_3)$. Recalling that the characteristic function of A is $\chi_A = (x_2 \oplus x_3 \oplus x_5)$, we have:

$$f(x_1, \ldots, x_5) = \chi_A \cdot f_{A2} = (x_2 \oplus x_3 \oplus x_5) \cdot [\overline{x}_4 + (x_1 \oplus x_2 \oplus x_3)] .$$

We finally notice that this decomposition is identical to the one obtained with the other strategy in the previous example.

4 Incompletely Specified Autosymmetric and D-reducible Functions

In this section, we discuss the case where an incompletely specified Boolean function f is D-reducible and autosymmetric at the same time.

The autosymmetry test of an incompletely specified Boolean function specifies the don't cares to a 0 or a 1, in order to obtain a completely specified function, whose degree of autosymmetry is maximum [2]. Therefore, after the autosymmetry test, the reduced function f_k is completely specified.

Meanwhile, the D-reducibility reduction of an incompletely specified Boolean function f has the objective to find the smallest affine space A that contains the minterms of f; the points of A that are not minterms of f can be 0 or don't cares. Thus, the projected function f_A remains an incompletely specified Boolean function. In any case, if we consider a function f that is both D-reducible and autosymmetric, the resulting decomposed functions f_{kA} and f_{Ak} are completely specified, because of the autosymmetry test.

When the initial function is incompletely specified, the properties proved in Sect. 3 do not hold. In this case, we have that the completely specified functions f_{kA} and f_{Ak} can be different. We show this through an example from the ESPRESSO benchmark suite [17].

Example 7 Consider the function f that is the first output of the *bench* benchmark defined as follows: $f^{on} = \{010001, 011010, 011110, 101001, 101110\}$, $f^{off} = \{000110, 001000, 001001, 001010, 001110, 001111, 100010, 100101, 100110\}$; all the other points are in f^{dc}. If we first apply D-reducibility and then autosymmetry, we obtain the Karnaugh maps shown in Fig. 7. On the left side

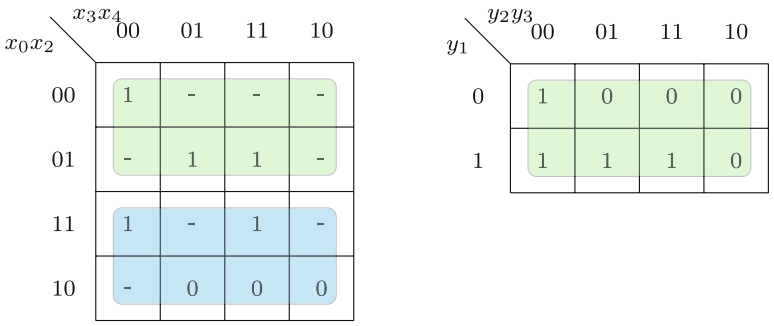


Fig. 7 Left-hand side: Karnaugh of the projection f_A for *bench_0*. Right-hand side: Karnaugh map of the restriction f_{A1} for *bench_0*

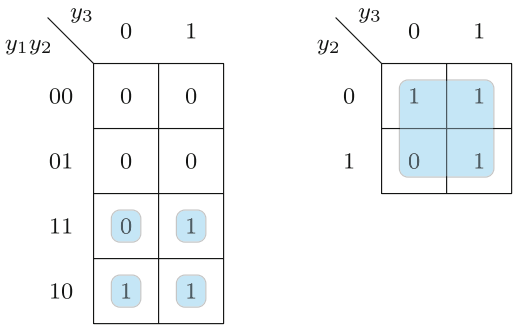


Fig. 8 Left-hand side: Karnaugh map of the restriction f_3 for *bench_0*. Right-hand side: Karnaugh map of the projection f_{3A} for *bench_0*

of the figure, we have the Karnaugh map of the projection f_A, which is a 1-autosymmetric function. Thus, on the right, we have the Karnaugh map of the restriction f_{A1}. Notice that the Karnaugh map on the left contains don't cares, since the D-reducibility test does not specify the don't care conditions. If we first apply autosymmetry and then D-reducibility, we have the Karnaugh maps shown in Fig. 8. The incompletely specified function f is 3-autosymmetric. Thus, on the left of Fig. 8, we have the Karnaugh map of the restriction f_3. On the right side, we have Karnaugh map of the projection f_{3A}. Notice that the Karnaugh map on the left does not contain don't cares, since the autosymmetry test specifies the don't care conditions in order to obtain the best degree of autosymmetry. From this example, we can observe that the in presence of don't care conditions, we can have two different final results, on changing the test ordering.

Finally, considering the results obtained in Sects. 3 and 4, we can define the following strategy:

- If the function is completely specified, we can use one of the two approaches (actually, the experiments in Sect. 6 show that performing the D-reducibility and then the autosymmetry seems to be the more efficient approach).
- If the function is incompletely specified, we should use both approaches and take the best solution (the experimental results in Sect. 6 show that the running time cost for performing both approaches is affordable).

5 Multiplicative Complexity

In this section we discuss the multiplicative complexity of a completely specified autosymmetric and D-reducible function.

Since f is autosymmetric and D-reducible, we can upper bound its multiplicative complexity by first projecting f onto A, and then by estimating the multiplicative complexity of the restriction f_{Ak} of f_A, as proved in [3], in the following way $M(f) \leq (n - \dim A) + M(f_{Ak})$.

Alternatively, we can first compute the restriction f_k and then estimate the multiplicative complexity of the projection $f_{k,A}$ of f_k on the affine space A. Indeed, we have that, since f is autosymmetric, the multiplicative complexity of f (i.e., $M(f)$) is equal to the multiplicative complexity of f_k (i.e., $M(f_k)$). In fact, f can be reconstruct from f_k just replacing the y_i with XORs of literals. Moreover, as proved in [3] we have that, if f is D-reducible, then $M(f) \leq (n - \dim A) + M(f_A)$. Recall that we have proved in Sect. 3 that, if f is both autosymmetric and D-reducible, also f_k is D-reducible. Therefore, we can say that $M(f_k) \leq (n - \dim A) + M(f_{kA})$. Since $M(f) = M(f_k)$, we finally have that $M(f) \leq (n - \dim A) + M(f_{kA})$, as expected.

6 Experimental Results

In this section, we report and discuss the experimental results reached applying both the autosymmetry test and the D-reducible decomposition to Boolean functions in the benchmarks from ESPRESSO, LGSynth'89 benchmark suite [17] and to some functions from cryptography benchmarks in the context of multiparty computation (MPC) and fully homomorphic encryption (FHE) [14, 15].

The experiments have been run on a Intel(R) Core(TM) i7-8565U 1.80 GHz processor with 8.00 GB RAM, on Windows 11 for D-reducibility, and on a virtual machine running OS Ubuntu 64-bit for autosymmetry.

Observe that autosymmetry and D-reducibility are properties of single outputs, e.g., different outputs of the same benchmark can have different autosymmetry degrees. Therefore, we perform the autosymmetry and D-reducibility tests on the single outputs of the considered benchmark suites. We considered each output as a separate Boolean function, and analyzed a total of 237 D-reducible and autosymmetric (nondegenerate) functions. The given functions and their restrictions or projections have been synthesized in XAG form using the heuristic approach proposed in [14].

We conducted four tests each composed by the following overall strategy: (1) regularity test (autosymmetry alone; or D-reducibility alone; or first autosymmetry and then D-reducibility; or first D-reducibility and then autosymmetry); (2) XAG construction on the projected/reduced function [14]; and (3) reconstruction of the original function in XAG form (adding XORs from the reduction equations and/or adding AND of XORs for the characteristic function of the affine space A).

We report in Table 1 a significant subset of functions as representative indicators of our experiments. The first column reports the name and the number of the considered output of each benchmark. The following triples of columns report the multiplicative complexity of the XAG (AND) and the number of XORs (XOR) for the case we are considering, obtained running the heuristic in [14], and the running time in seconds. These triples describe the results for the following four different strategies: autosymmetry alone, D-reducibility alone, first autosymmetry and then D-reducibility (A+D), and first D-reducibility and second autosymmetry (D+A).

Table 1 Results for functions that are both autosymmetric and D-reducible. Benchmarks with "*" are incompletely specified. The last row shows the average values obtained from all the benchmarks considered

	Autosymmetry [3]			D-reducibility [3]			A + D			D + A		
Benchmark_output	AND	XOR	Time (s)	AND	XOR	Time (s)	AND	XOR	Time (s)	AND	XOR	Time (s)
apla_4*	18	20	13.45	15	17	2.60	18	20	13.76	**9**	5	1.01
b10_4*	15	10	11.03	**14**	6	6.64	15	6	6.51	15	6	6.58
bench_0 *	**2**	0	0.01	5	15	0.33	**2**	0	0.01	3	2	0.01
cps_74	17	4	9.11	20	2	11.52	**16**	2	6.09	**16**	2	5.72
dk17_3*	5	3	0.39	12	5	3.19	**4**	9	0.05	6	0	0.01
duke2_12*	28	5	15.88	36	25	23.07	**23**	19	11.71	**23**	19	11.47
exam_4*	22	20	13.27	59	45	42.28	22	20	13.65	**10**	13	2.95
exep_6*	20	0	8.13	16	0	0.01	**15**	0	0.16	16	0	0.01
exp_11*	16	1	9.21	6	8	0.39	**5**	8	0.46	**5**	1	0.01
p1_15*	20	26	6.60	**17**	18	13.55	18	12	5.57	18	12	4.99
p3_7*	32	30	22.55	28	11	14.04	32	30	24.99	**18**	32	6.58
pdc_3*	45	17	32.50	213	65	105.92	45	17	32.70	**36**	23	20.22
pdc_5*	21	21	14.54	270	76	123.40	**19**	25	8.47	21	25	8.55
sao2_2*	**7**	0	5.83	27	9	11.49	**7**	0	1.66	**7**	0	1.56
spla_5*	**67**	40	53.53	142	55	92.78	70	28	56.70	70	28	52.70
spla_12*	64	17	43.72	95	43	56.89	**59**	24	37.86	95	43	49.25
t1_22	**5**	0	1.23	**5**	2	0.15	**5**	0	0.10	**5**	0	0.08
t4_3*	11	6	2.00	15	6	6.91	12	1	5.43	**5**	9	0.19
x1dn_2*	**16**	8	6.58	19	8	6.35	**16**	8	5.18	17	8	2.47
dec_untilsat_39	**6**	0	1.52	**6**	0	0.01	**6**	0	1.71	**6**	0	0.01
Average	10.43	6.09	5.13	22.20	9.82	9.50	**9.78**	5.89	4.19	11.81	5.46	**3.70**

The experiments show that the functions where the XAG minimization can benefit from autosymmetry and D-reducibility are about 27%, with an average reduction of the number of ANDs of about 27.4%; the number of functions where the estimates of the multiplicative complexity are the same is about 66.7%, while for the 6.3% of the functions, the method provides a worst result. The worst result could come from the fact that the approach proposed in [14], for XAG synthesis, is heuristic. Some particular benchmarks seem to highly benefit from the proposed strategies. For example, the benchmark *t4_3* can be represented using the D+A approach with the gain of 55%, in AND gates, with respect to exploiting autosymmetry alone. We finally observe that the combined methods can also provide a reduction of the number of XOR gates, due to the XOR factorization in both approaches.

In conclusion, the experiments show that:

1. Running times deeply depend on the XAG heuristic [14]. Moreover, in general, the running time for the XAG heuristic depends on the dimension of its input function. For this reason, in the cases when we perform both the testing procedures, often the total running times are reduced since the input function for the XAG heuristic is smaller. In other words, the gain in running time for constructing the XAG is higher than the running times required for testing the two regularities.
2. In case of completely specified functions (where A+D and D+A give the same results), the strategy more convenient is D+A since this strategy has better running times.
3. In case of incompletely specified functions, it is convenient to test both strategies A+D and D+A in order to find the best solution. The sum of the running times of the two approaches (A+D and D+A) is about the 50% greater than the running time of the autosymmetry approach alone (which is much more time-consuming than the D-reducibility test). Therefore, testing both strategies (A+D and D+A) is still computationally convenient.

7 Conclusion

This paper has addressed regular functions that are both autosymmetric and D-reducible. The theoretical study shows that in the case of completely specified Boolean functions, the two tests can be performed in any order, obtaining exactly the same decomposition. In the case of incompletely specified Boolean functions, this property does not hold. The experimental results validate the proposed approach. Future works can include the study of other XOR-based regularities for enhancing the computation of multiplicative complexity.

References

1. Albrecht, M.R., Rechberger, C., Schneider, T., Tiessen, T., Zohner, M.: Ciphers for MPC and FHE. In: Advances in Cryptology—EUROCRYPT Proceedings, Part I, pp. 430–454 (2015)
2. Bernasconi, A., Ciriani, V.: Autosymmetry of incompletely specified functions. In: Design Automation and Test in Europe (DATE) (2021)
3. Bernasconi, A., Cimato, S., Ciriani, V., Molteni, M.C.: Multiplicative complexity of XOR based regular functions. IEEE Transactions on Computers (Early Access) (2022)
4. Bernasconi, A., Ciriani, V.: Dimension-reducible boolean functions based on affine spaces. ACM Trans. Design Autom. Electr. Syst. **16**(2), 13:1–13:21 (2011)
5. Bernasconi, A., Ciriani, V., Luccio, F., Pagli, L.: Exploiting regularities for boolean function synthesis. Theory Comput. Syst. **39**(4), 485–501 (2006)
6. Bernasconi, A., Ciriani, V., Luccio, F., Pagli, L.: Synthesis of autosymmetric functions in a new three-level form. Theory Comput. Syst. **42**(4), 450–464 (2008)
7. Boyar, J., Peralta, R., Pochuev, D.: On the multiplicative complexity of Boolean functions over the basis ($\wedge$, $+$, 1). Theor. Comput. Sci. **235**(1), 43–57 (2000)
8. Çalik, Ç., Turan, M.S., Peralta, R.: The multiplicative complexity of 6-variable Boolean functions. Cryptogr. Commun. **11**(1), 93–107 (2019)
9. Ciriani, V.: Synthesis of SPP three-level logic networks using affine spaces. IEEE Trans. CAD Integr. Circuits Syst. **22**(10), 1310–1323 (2003)
10. Goudarzi, D., Rivain, M.: On the multiplicative complexity of Boolean functions and bitsliced higher-order masking. In: Cryptographic Hardware and Embedded Systems—CHES 2016—18th International Conference, Santa Barbara, CA, USA, Proceedings, pp. 457–478 (2016)
11. Halecek, I., Fiser, P., Schmidt, J.: Are XORs in logic synthesis really necessary? In: 20th IEEE International Symposium on Design and Diagnostics of Electronic Circuits & Systems, DDECS 2017, Dresden, Germany, April 19–21, 2017, pp. 134–139 (2017)
12. Liebler, R.: Basic Matrix Algebra with Algorithms and Applications. Chapman & Hall/CRC (2003)
13. Luccio, F., Pagli, L.: On a new Boolean function with applications. IEEE Trans. Comput. **48**(3), 296–310 (1999)
14. Testa, E., Soeken, M., Riener, H., Amaru, L., Micheli, G.D.: A logic synthesis toolbox for reducing the multiplicative complexity in logic networks. In: 2020 Design, Automation Test in Europe Conference Exhibition (DATE), pp. 568–573 (2020)
15. Testa, E., Soeken, M., Amarù, L.G., Micheli, G.D.: Reducing the multiplicative complexity in logic networks for cryptography and security applications. In: Proceedings of the 56th Annual Design Automation Conference 2019, DAC, p. 74 (2019)
16. Turan, M.S., Peralta, R.: The multiplicative complexity of boolean functions on four and five variables. In: Lightweight Cryptography for Security and Privacy—Third International Workshop, LightSec 2014, Istanbul, Turkey, pp. 21–33 (2014)
17. Yang, S.: Logic synthesis and optimization benchmarks user guide version 3.0. User guide, Microelectronic Center (1991)

Two-Operand Modular Multiplication to Small Bit Ranges

Danila Gorodecky and Leonel Sousa

1 Introduction

Residue number system (RNS) and modular arithmetic provide data parallelism by representing integers as residues given a preselected moduli set $\{p_1, p_2, \ldots, p_m\}$ of co-prime numbers, with $Z = p_1 \cdot p_2 \cdot \ldots \cdot p_m$. Each of these residues requires a significantly lower number of bits than the ordinary representation. Arithmetic operations on residues are calculated independently for each modulo of the set. A larger number of smaller moduli provide parallelism and allow fast calculations, achieving high performance computing. The range of integers represented in the RNS domain cannot exceed Z different integers.

Implementations of RNS can be found in aircraft systems [1], neural computations [2], real-time signal processing (pattern recognition) [3], cryptography [4], and radio astronomy [5]. In general, RNS is efficient for processing large amounts of data (hundreds and thousands of bits) [6] or repeatedly computed arithmetic operations, as well as in assuring the reliability of computational arithmetic [2, 7, 8].

Data processing in RNS involves the following steps, as depicted in Fig. 1:

1. input operands $A_1, A_2, \ldots, A_n$ are converted from a positional representation into a modular representation as a set of residues (block (1) on Fig. 1);
2. arithmetic operations with these residues, for each modulo p_1,p_2, $\ldots$, p_m, are computed (central blocks on Fig. 1);

D. Gorodecky (✉)
INESC-ID, Instituto Superior Tecnico, Universidade de Lisboa, Lisboa, Portugal
EHU/EPAM School of Digital Engineering, Vilnius, Lithuania

L. Sousa
INESC-ID, Instituto Superior Tecnico, Universidade de Lisboa, Lisboa, Portugal
e-mail: leonel.sousa@tecnico.ulisboa.pt

R. Drechsler, S. Huhn (eds.), *Advanced Boolean Techniques*,
https://doi.org/10.1007/978-3-031-28916-3_8

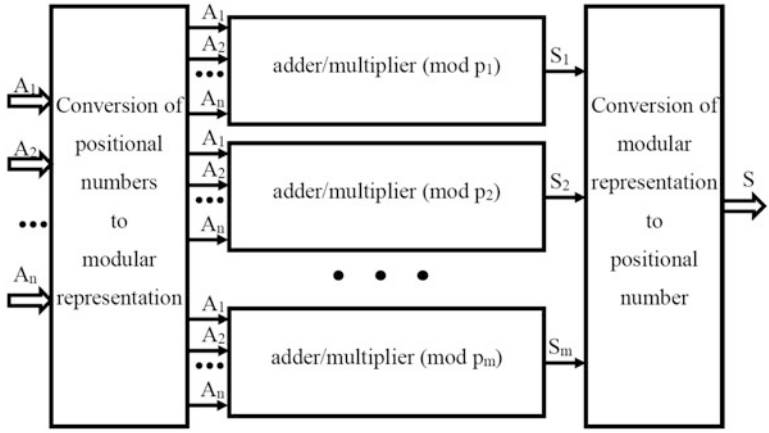


Fig. 1 RNS structure

3. the residues $S_1, S_2, \ldots, S_n$ for each modulo are converted back from the RNS domain to the positional number representation S (block (2) on Fig. 1).

Thus, the RNS structure consists of two types of components: converters and arithmetic units (adders and multipliers).

A major limitation to the application of RNS in real life is the requirements of converters, blocks (1) and (2) on Fig. 1. Forward computation of residues $(X\,(mod\,P))$ and backward recovery of positional representation are indispensable.

There are several memory and combinational hardware realizations of RNS converters [9]. Memory-free approaches are important for high performance computing, but the main severe restriction on special moduli for the set does not allow wide-range RNS implementations. Generally, these moduli sets consist of variations of $2^{\delta} \pm d$ [10], where $d = 1, 3, 5$, is a natural number, and all selected moduli must be co-prime numbers. According to this condition, in order to multiply, for example, two 50-bit integer numbers, the following moduli set can be adopted: $\{2^4 - 5, 2^4 + 1, 2^4 + 3, 2^5 - 1, 2^6 \pm 3, 2^6 - 5, 2^7 - 1, 2^7 \pm 3, 2^8 - 5, 2^8 + 1, 2^9 - 3, 2^9, 2^{10} - 5\}$.

This set consists of 15 co-prime numbers, and the speed of calculation is limited by 10-bit multiplication modulo $2^{10} - 5$. On the other hand, 50-bit multiplication can be computed in the scope of a set of 19 more general co-prime integers, such as $\{73, 71, 67, 64, 61, 59, 53, 49, 47, 43, 41, 37, 31, 29, 27, 25, 23, 19, 17\}$, where the slowest multiplication is limited by 7 bits, modulo 73.

The memory-based design of arithmetic units by modulo P (converters from RNS to binary) is organized as a pipeline and reduces the length of inputs X from k to δ in a step-by-step manner, where k is bit range of the input and P is δ-bit modulo [11]. Commonly, pipelining of memory-based approaches of modulo calculation consists of $k-\delta$ similar blocks, with each block executing multiplication, subtraction, and comparison. For instance, conversion of a 100-bit number on modulo 997 requires 90 stages of pipelining. Thus, this type of converters takes

large area costs and imposes a long calculation latency. Another type of memory-based approaches uses memory to store all possible pre-computed residues for every modulo from the moduli set [10]. This technique is based on lookup tables.

Hardware implementation of modular arithmetic is organized similarly to the conversion operations, i.e., arithmetic operations for a special moduli set are realized with non-memory techniques and calculation for an arbitrary modulo utilizes memory.

This chapter proposes a non-memory approach for computing $A \cdot B = R(mod\ P)$, for an arbitrary value for modulo P. Experimental results of the synthesis of these units on FPGA are presented.

2 Two Operand Modular Multiplication

Allowing an arbitrary modulo may reduce significantly the bit range of multiplication comparing to multiplication for a special moduli set. Considering multiplication in RNS for an arbitrary moduli set in 50-bit range, as it has noted in the introduction, the difference in slower multiplications is 3 bits (i.e., the largest modulo from the first set is $2^{10} - 5$, 10 bits are required, while the largest modulo for the second set is 73, requiring only 7 bits).

Arithmetic calculation is the main gain of using RNS. RNS splits inputs into sub-vector of smaller bit ranges; with smaller bit range of sub-vectors, the performance of calculations increases. On the other hand, RNS deals with special values of moduli due to the complexity of transformation to/from RNS for an arbitrary value of modulo.

This research is focused on modular multiplication for an arbitrary value of moduli, taking into account the idea of efficient transformation [12]. Thus, for instance, multiplication of two 3000-bit numbers might be represented as the parallel multiplication by 560 different moduli in RNS, where every modulo does not exceed 12-bit range. That is why the emphasis on multiplication by moduli up to 12-bit values.

The architecture of multiplication is based on the idea of constructing small blocks of adders and multipliers to organize multiplication for any bit range [13].

The idea of the approach consists of three steps, as pictured on Fig. 2: input splitting, bit-range reducing, and comparing and subtraction.

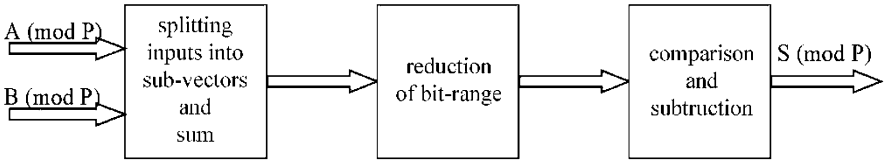


Fig. 2 Steps of the proposed approach for modular multiplication

2.1 Binary Digit Fragmentation

The first branch of the approach decomposes inputs A and B of Fig. 2 into bits multiplied by power of two constants and the results added modulo:

$$A \cdot B(mod\ P) = \sum_{i=1}^{k} \sum_{j=1}^{k} \left(a_i \cdot b_j \cdot 2^{i+j-2}\right)(mod\ P) = S, \tag{1}$$

where $A = (a_k, a_{k-1}, \ldots, a_1)$ and $B = (b_k, b_{k-1}, \ldots, b_1)$, where k bit is the most significant bit.

The sum is calculated in a recursive manner, while sum $> 2 \cdot P$, according to Algorithm 1.

Algorithm 1: Algorithm for modular multiplication

Input:

z – bit-range of $S = (s_z, s_{z-1}, \ldots, s_1)$, where S and z are defined by formula (1). Namely, it is supposed that $A = B = 2^k - 1$. In this case $a_1 = a_2 = \cdots = a_k = b_1 = b_2 = \cdots = b_k = 1$. Hence formula (1) equals $\sum_{i=1}^{k} \sum_{j=1}^{k} = 2^{i+j-2}(mod\ P)$

$f(R)$ – number of bits in R

1: $z_{new} = z$
2: $S_{temp} = S$
3: **while** $S_{temp} > 2 \cdot P$ **do**:
4: $\quad S_{temp} = \sum_{q=1}^{z_{new}} s_q \cdot \left(2^{q-1}(mod\ P)\right)$
5: $\quad z_{new} = f(S_{temp})$
6: **if** $S_{temp} > P$ **then**
7: $\quad S = S_{temp} - P$
8: **else**
9: $\quad S = S_{temp}$

Note that the output of Algorithm 1 is the result S, which does not exceed $2^{k+2} - 1$. Experimentally, it has been detected that recursive sum (the loop part of the Algorithm) is computed in 2 or 3 cycles for up to 12-bit modulo.

Consider an example for two 3-bit operands $A = (a_3, a_2, a_1)$, $B = (b_3, b_2, b_1)$ and multiplication by modulo $P = 7$. Thus, formula 1 takes the form:

$$\begin{aligned} A \cdot B(mod\ 7) &= a_1 \cdot b_1(mod\ 7) + a_1 \cdot b_2 \cdot 2(mod\ 7) + a_1 \cdot b_3 \cdot 2^2(mod\ 7) \\ &\quad + a_2 \cdot b_1 \cdot 2(mod\ 7) + a_2 \cdot b_2 \cdot 2^2(mod\ 7) + a_2 \cdot b_3 \cdot 2^4(mod\ 7) \\ &\quad + a_3 \cdot b_1 \cdot 2^2(mod\ 7) + a_3 \cdot b_2 \cdot 2^3(mod\ 7) + a_3 \cdot b_3 \cdot 2^4(mod\ 7) \\ &= a_1 \cdot b_1(mod\ 7) + a_1 \cdot b_2 \cdot 2(mod\ 7) + a_1 \cdot b_3 \cdot 4(mod\ 7) \\ &\quad + a_2 \cdot b_1 \cdot 2(mod\ 7) + a_2 \cdot b_2 \cdot 4(mod\ 7) + a_2 \cdot b_3 \cdot 2(mod\ 7) \\ &\quad + a_3 \cdot b_1 \cdot 4(mod\ 7) + a_3 \cdot b_2 \cdot 1(mod\ 7) + a_3 \cdot b_3 \cdot 2(mod\ 7) \\ &= S \leq 21, \end{aligned}$$

where $S = (s_5, s_4, s_3, s_2, s_1)$.

Note that $S = 21$ if $a_1 \cdot b_1 = a_1 \cdot b_2 = a_1 \cdot b_3 = a_2 \cdot b_1 = a_2 \cdot b_2 = a_2 \cdot b_3 = a_3 \cdot b_1 = a_3 \cdot b_2 = a_3 \cdot b_3 = 1$.

Then

$$\begin{aligned} S &= s_1 \cdot 2^0(mod\ 7) + s_2 \cdot 2^1(mod\ 7) + s_3 \cdot 2^2(mod\ 7) \\ &\quad + s_4 \cdot 2^3(mod\ 7) + s_5 \cdot 2^4(mod\ 7) \\ &= s_1 \cdot 1(mod\ 7) + s_2 \cdot 2(mod\ 7) + s_3 \cdot 4(mod\ 7) \\ &\quad + s_4 \cdot 1(mod\ 7) + s_5 \cdot 2(mod\ 7) \\ &= S_{temp} \leq 2 \cdot P. \end{aligned}$$

Actually, the maximum value of S_temp achieves 8, if $s_1 = s_2 = s_3 = s_4 = 1$ and $s_5 = 0$.

The final step of multiplication compares S_temp with P and subtracts P from S_temp if needed.

2.2 Sub-Vector Splitting

The other part of the approach is to segment factors into 2- to 6-bit sub-vectors, depending on the initial length of the factors, and perform modulo multiplication of pairs of sub-vectors by power of 2 constants. This technique proposes to consider the multiplication of sub-vectors by a constant as a system of Boolean functions. Tools like Espresso [14] and ABC [15] can be used to minimize these systems of functions.

The question of optimal splitting factors for multiplication in regular arithmetic has been already investigated in [16]. Results of experiments, for which the bit range of results is equal to the length of the factors, are shown in Table [16]. This result was obtained for two-level minimization; multi-level minimization has not been considered yet.

Table 1 Comparison of Synopsys multiplication and minimized Boolean functions in Espresso

Type of multiplication, BITs × BITs → BITs	Speed of calculations, GHz	
	Espresso multiplication	Synopsys multiplication
2 × 2 → 2	6.25	6.66
3 × 3 → 3	5.55	5
4 × 4 → 4	5.3	4
5 × 5 → 5	3.45	3.13
6 × 6 → 6	2.78	2.85
7 × 7 → 7	2.39	2.7
8 × 8 → 8	2	2.43

According to the experimental results in Table 1, 3 by 3, 4 by 4, and 5 by 5 bit multiplications are preferable. Since 3 by 3 bit multiplication provides the highest speed, it is used as the base for modular multiplication. If a bit length of an operand is not divisible by 3, it is split into 3-bit sub-vectors and one 1- or 2-bit additional sub-vector. For instance, 8-bit operand is split into two 3-bit and one 2-bit sub-vectors. Thus, binary representations of both factors are split into $\lceil k/q \rceil = v$ sub-vectors, where q is the bit range of the sub-vectors with the largest number bits. For instance, splitting 8-bit operands A and B into 3-bit sub-vectors, $q = 3$ and $v = \lceil 8/3 \rceil = 3$.

In common, k-input A and B values are split into q-bit sub-vectors. Typically, in the worst case, when $\delta = \lceil log_2 P \rceil + 1 = k$, the maximum value of S does not exceed $(P-1) \cdot v$. The most suitable bit range of q is defined experimentally depending on the hardware features. This research focuses on multiplication $A \cdot B = S(mod\ P)$, where A, B and P vary from 6 to 12 bits; then, the product of multiplication is calculated in two steps. On the first step, the sum of products of two sub-vectors is calculated with Eq. 2.

$$A \cdot B(mod\ P) = \sum_{i=1}^{v} \sum_{j=1}^{v} \left(A_i \cdot B_j \cdot 2^{(i+j-2)\cdot q}\right)(mod\ P) = S, \tag{2}$$

where $A = (A_v, A_{v-1}, \ldots, A_1)$ and $B = (B_v, B_{v-1}, \ldots, B_1)$ are v dimension sub-vectors, q is the number of bits per sub-vector, and A and B are k-bit vectors.

Equation 2 is applied while $S_{temp} \leq 2 \cdot P$, in a recursive manner, according to Algorithm 2.

Consider an example with $A \cdot B(mod\ 47)$, i.e., A and B are 6-bits numbers. Splitting operands into two, i.e., $v = 2$, 3-bit sub-vectors, formula 2 is unrolled as

$$\begin{aligned} A \cdot B(mod\ 47) &= A_1 \cdot B_1(mod\ 47) + A_1 \cdot B_2 \cdot 2^3(mod\ 47) \\ &\quad + A_2 \cdot B_1 \cdot 2^3(mod\ 47) + A_2 \cdot B_2 \cdot 2^6(mod\ 47) = \\ &= S_{temp_1}. \end{aligned}$$

Algorithm 2: Algorithm for modular multiplication

Input:

A, B;

v – number of sub-vectors in A and B, i.e. $A = (A_v, A_{v-1}, \ldots, A_1) < P$ and $B = (B_v, B_{v-1}, \ldots, B_1) < P$;

q – bit-range of i-th sub-vector of A and B;

1: $S_{temp} = max\left\{\sum_{i=1}^{v}\sum_{j=1}^{v}\left(\left(A_i \cdot B_j \cdot 2^{(i+j-2)\cdot q}\right)(mod\ P)\right)\right\}$

2: **while** $S_{temp} > 2 \cdot P$ **do**:

3: $z = \lceil \frac{f(S_{temp})}{q} \rceil$

4: $S_{temp} = (S^z_{temp}, S^{z-1}_{temp}, \cdots, S^1_{temp})$ - splitting into z sub-vectors;

5: $S_{temp} := \sum_{i=1}^{z}\left(2^{i-1} \cdot S^i_{temp}(mod\ P)\right)$

6: **if** $S_{temp} > P$ **then**

7: $S = S_{temp} - P$

8: **else**

9: $S = S_{temp}$

S_{temp_1} takes the maximum value if $A = 45$ and $B = 15$, and it equals to the 8-bit width number $158 = (10011110)$.

Decimal 45 in binary is (101101) and 15 in binary is (001111). Hence, $A_1 = A_2 = (101)$ or 5 in decimal, $B_1 = (111)$ or 7 in decimal, and $B_2 = (001)$ or 1 in decimal. Returning to formula 2, we have:

$$\begin{aligned} A \cdot B(mod\ 47) &= 5 \cdot 7(mod\ 47) + 5 \cdot 1 \cdot 2^3(mod\ 47) \\ &+5 \cdot 7 \cdot 2^3(mod\ 47) + 5 \cdot 1 \cdot 2^6(mod\ 47) \\ &= 35(mod\ 47)+40(mod\ 47)+40(mod\ 47)+45(mod\ 47) = 158. \end{aligned}$$

The idea behind the iterative implementation of Algorithm 2 is to reduce S_{temp_1} to the value which is less than $47 \cdot 2$. Let's assume that $S_{temp_1} = 158$, then

$$S_{temp_2} = 6 + 3 \cdot 2^3(mod\ 47) + 2 \cdot 2^6(mod\ 47) = 6 + 24 + 34 = 64.$$

Taking into account that $2 \cdot 47 > 64 > 47$, $S = 64 - 47 = 17$.

Note that S_{temp_1} is precalculated and is specified in the realization.

3 Boolean Representations

The central point of the proposed approach is minimization of a system of Boolean functions, which represents the multiplication of a sub-vector by a sub-vector and

two sub-vectors by a constant. Additionally these multipliers are used as structural blocks.

With the primary focus on the synthesis of modular multipliers for FPGAs, it is crucial to take into account the hardware features of the target devices.

The main circuit of an FPGA is the lookup table (LUT), a memory for implementing a system of Boolean functions. Typically, FPGA LUTs have four, five, or six inputs. Thus, we can represent modular multiplication as the superposition of functions, where the number of variables of each and every function is equal to the number of inputs of the FPGA. This is a multi-level representation of Boolean functions.

In some cases two-level representation of systems of Boolean functions might be more efficiently implemented, as it is shown in [16]. Applications for two- and multi-level minimization of systems of Boolean functions have been proposed, such as ABC [15] and FLC2 [17]. The contribution [18] describes in detail the process and the sense of multi-level minimization for FPGA implementation.

In the experiments we have used a multi-level minimization of Boolean functions for targeting on five- and six-input LUTs, and two-level minimization in disjunctive normal form. The first branch of the proposed technique is based on fragmentation of operands into bits and is characterized by $O(\delta^2)$ sums, where δ is the number of bits of the inputs. For example, 51 additions are needed for $P = 47$. The second branch of the proposed technique is concluded by summing q-bit vectors and is characterized by $O(q)$ sums. For $P = 47$, there are six sums needed.

The second branch of the technique consists in representing each operand in formula 2 as a minimized system of Boolean functions. In the example from the previous section, $A_1 \cdot B_1(mod\ 47)$, $A_1 \cdot B_2 \cdot 2^3(mod\ 47)$, $A_2 \cdot B_1 \cdot 2^3(mod\ 47)$, $A_2 \cdot B_2 \cdot 2^6(mod\ 47)$ are considered as the system of Boolean functions. For instance, the truth table of $A_2 \cdot B_2 \cdot 2^6(mod 47) = R$ realization is represented in Table 2. It consists on 64 rows and 17 columns, but *constant* columns are formal, and they are not included into the truth table during minimization, because the value of the constant is the same for all lines in the truth table.

Table 2 Truth table for $A_2 \cdot B_2 \cdot 2^6(mod\ 47)$

A_2	B_2	Constant ($2^6(mod\ 47) = 17$)	R
000	000	10001	000000
000	001	10001	000000
...	...	...	...
001	001	10001	010001
001	010	10001	100010
...	...	...	...
111	110	10001	001001
111	111	10001	100010

4 Hardware Realization of Modulo Multipliers

This section presents the results of the synthesis of modular multipliers $A \cdot B(mod\ P)$ for moduli: 47, 113, 241, 491, 887, 2001, and 4051.

All multipliers were described on Verilog. The performance is considered as the critical path in nanoseconds (ns). All approaches are non-memory; the area is measured as the number of FPGA LUTs. Vivado suite provides the option to synthesize a scheme without implementing any Block RAM (BRAM) and multipliers of FPGA. It aims to synthesize a scheme only with LUTs.

The experiments have been conducted on a Kintex-7 (xa7z010clg225-1l) in Xilinx Vivado 2019.1. Table 3 provides the number of LUTs and the critical path for the considered multipliers. The technique proposed in Sect. 2.1 *Binary bit fragmentation* of Sect. 2 is designated *per_bits*. The techniques proposed in Sect. 2.2 *Sub-Vector Splitting* of Sect. 2 are implemented in three different ways:

- two-level minimization (Espresso and FLC2) with exact minimization mode—*2lev*;
- multi-level minimization for 5-input LUTs (ABC and FLC2)—*5lt*;
- multi-level minimization for 6-input LUTs (ABC and FLC2)—*6lt*.

These four realizations are compared with the standard realization of modular multiplication by Xilinx Vivado (Standard in Table 3). The advantages of the proposal in comparison with the standard realizations are highlighted in Table 3.

We may conclude that the implementation of two- and multi-level minimization are the best to what concerns performance. The length of the critical paths varies depending on the value of the exact modulo. In fact, Vivado saves area costs with the implemented algorithm for moduli 113 and 221. Commonly, it is difficult to define a dependency between the value of modulo and the figures of merit of the realization: the critical path and the area cost. There is no gain in the *per_bits* individual approach for modular multiplication on FPGA.

5 Conclusion and Further Work

Correctness in RNS calculations demands co-prime moduli set. The known approaches of modular multiplication are oriented to special moduli. This contribution proposes a technique for efficient modular multiplication in the RNS domain without any requirement of special moduli. The idea behind the approaches is to split the inputs into smaller sub-vectors and sum the partial products. The sub-vector multiplication with reduction is optimized through minimization of systems of Boolean functions. Experimental results were provided for FPGA implementations. The experimental results cover a range of moduli represented with 6–12 bits. The main conclusion is that the proposed approach allows to achieve a trade-off between performance and area costs on FPGAs, and the resulting modular

Table 3 Synthesis Results: LUTs and Critical Path

P	#LUTs "+" (advantage) or "−" (disadvantage) in % comparing with the standard)	Critical path (ns) ("+" (advantage) or "−" (disadvantage) in % comparing with the standard)
Standard		
47	91	24.0
113	81	19.9
241	152	24.7
491	175	24.1
887	231	27.0
2011	221	31.4
4051	298	31.3
2lev		
47	54 (+ **41%**)	13.9 (+ **42%**)
113	122 (−51%)	18.1 (+ **9%**)
241	119 (+ **22%**)	19.2 (+ **22%**)
491	131 (+ **25%**)	18.2 (+ **24%**)
887	242 (−5%)	26.0 (+ **4%**)
2011	285 (−29%)	27.5 (+ **15%**)
4051	291 (+ **4%**)	27.3 (+ **23%**)
per_bits		
47	205 (−125%)	12.3 (+ **51%**)
113	260 (−220%)	70.8 (−256%)
241	462 (−203%)	98.5 (−299%)
491	249 (−42%)	115.7 (−380%)
887	360 (−56%)	44.0 (−63%)
2011	363 (−64%)	39.0 (−24%)
4051	557 (−87%)	45.1 (−44%)
5lt		
47	51 (+ **44%**)	13.7 (+ **43%**)
113	116 (− 43%)	19.3 (+ **3%**)
241	140 (+ **8%**)	17.9 (+ **27%**)
491	125 (+ **29%**)	19.3 (+ **20%**)
887	244 (−6 %)	24.3 (+ **10%**)
2011	280 (−27%)	25.2 (+ **20%**)
4051	297 (0%)	28.6 (+ **9%**)
6lt		
47	52 (+ **43%**)	14.4 (+ **40%**)
113	114 (−40%)	19.7 (+ **1%**)
241	129 (+ **15%**)	19.2 (+ **22%**)
491	129 (+ **26%**)	18.7 (+ **22%**)
887	245 (−6%)	25.3 (+ **6%**)
2011	283 (−28%)	24.5 (+ **22%**)
4051	310 (−4%)	26.7 (+ **16%**)

multipliers are more efficient than the ones implemented with FPGA automatic synthesis tools. Future research may consider to increase the number of operands up to some dozens in modular multiplication, and to extend the bit range of inputs. Another path of research is to minimize the Boolean representations of multipliers within the class of Reed-Muller polynomials.

Further research may include upgrading steps of calculations pictured on Fig. 2: choosing the most appropriate technique of minimization and bit range of input splitting (left block of the figure); improving the architectures of reducing (central block of the figure); and comparing and subtraction (right block of the figure).

References

1. Malashevich, B.M.: Unknown Modular Supercomputers. In: Proceedings of Conference for 50 years of Modular Arithmetic, pp. 50–70. Moscow, Russia (2005)
2. Cherviakov, N.I., et al.: Modular Structures of Parallel Computing Systems for Neuroprocessors. Moscow, Russia, 288 p. (2003)
3. Flatt, H., Hesselbarth, S., Flugel, S., Pirsch, P.: A modular coprocessor architecture for embedded real-time image and video signal processing. In: Embedded Computer Systems: Architectures, Modeling, and Simulation, 7th International Workshop, 2007, Samos, Greece, Proceedings, p. 241–250
4. Ozturk, E., Sunar, B., Savas, E.: Low-Power Elliptic Curve Cryptography Using Scaled Modular Arithmetic. In: Proceedings of the 6th International Workshop Cryptographic Hardware in Embedded Systems, Cambridge, MA, USA, vol. 3156, pp. 92–106 (2004)
5. Nakahara, H., Sasao, T., Nakanishi, H., Iwai, K., Nagao, T., Ogawa, N.: An FFT circuit using nested RNS in a digital spectrometer for a radio telescope. In: International Symposium on Multiple-valued Logic (ISMVL-2016), pp. 60–65 (2016)
6. Montgomery, P.L.: Modular Multiplication without Trial Division Mathematics of Computation, vol. 44 (170), pp. 519–521 (1985)
7. Sousa, L., Chaves, R.: A universal architecture for designing efficient modulo multipliers. IEEE Transactions on Circuits and Systems **52**(6), 1166–1178 (2005)
8. Zimmermann, R.: Efficient VLSI implementation of modulo addition and multiplication. In: 14th IEEE Symposium on Computer Arithmetic, Adelaide, Australia, pp. 158–167 (1999)
9. Mohan, P.V.A.: Residue Number Systems: Teory and Applications. Birkhauser, Basel (2016)
10. Omondi, A.R., Premkumar, B.: Residue Number System: Theory and Implementation. Imperial College Press, New York (2007)
11. Butler, J.T., Sasao, T.: Fast hardware computation of x mod z. In: 25th IEEE International Parallel and Distributed Processing Symposium Anchorage, Ak, USA, pp. 289–292 (2011)
12. Gorodecky, D., Villa, T.: Efficient implementation of modular division by input bit splitting. In: Proceedings of the 26th IEEE Symposium on Computer Arithmetic, June 10–12, 2019, Kyoto, Japan, pp. 54–60 (2019)
13. Schonhage, A., Strassen, V.: Schnelle Multiplikation groser Zahlen. Computing **7**, 281–292 (1971)
14. https://embedded.eecs.berkeley.edu/pubs/downloads/espresso/index.htm
15. http://people.eecs.berkeley.edu/~alanmi/abc/
16. Gorodecky, D.: Multipliers design technique based disjunctive normal form minimizationa and Fourier transformation. In: Proceedings of the 12th International Workshop on Boolean Problems, Freiberg, Germany, Sept. 22–23, 2016. ed. by Steinbach, B.: Freiberg University of Mining and Technology, pp. 145–150

17. Bibilo, P., Lankevich, Yu.: Minimizing the multilevel representations of systems of Boolean functions based on Shannon decomposition. Informatika **2**, 45–57 (2017)
18. Gorodecky, D., Bibilo, P.: Constant Multiplication Based on Boolean Minimization. In: Proceedings of the 14th International Workshop on Boolean Problems, Bremen, Germany (2020)

Low-Latency Real-Time Inference for Multilayer Perceptrons on FPGAs

Ahmad Al-Zoubi and Goerschwin Fey

1 Introduction

In areas like process control systems, e.g., in particle accelerators, autonomous driving, and critical infrastructures, control and monitoring systems are increasingly adapting to the use of neural networks, due to their high accuracy and tolerance to faulty data [4, 5, 11]. Given the sheer volume of sensor and image data on one hand, and the critical timing of output inference on the other hand, these systems are required to process this data with high throughput and low latency. Here, throughput measures the number of data samples processed within a given time, while latency measures the time between receiving a data sample and inferring the related result, e.g., a category that corresponds to a control decision or new control parameters. While several studies and commercial processing units have focused on the throughput aspect of the neural networks inference performance, latency-optimized processors were falling behind. Nevertheless, very low latency is required for real-time neural network processors in certain applications.

The multilayer perceptron (MLP) is a type of feed-forward neural network that uses the back-propagation technique for its training. An MLP has an input layer which acts as a receiver, one or more hidden layers for data computation, and an output layer which predicts the output. The ability of learning complex nonlinear relationships, generality, and imposing no restrictions on the input data distribution resulted in a wide use of this type of neural network. Along with its

A. Al-Zoubi (✉)
Hamburg University of Technology, Hamburg, Germany
Center for Data and Computing in Natural Sciences, Hamburg, Germany
e-mail: ahmad.al.zoubi@tuhh.de

G. Fey
Hamburg University of Technology, Hamburg, Germany
e-mail: goerschwin.fey@tuhh.de

R. Drechsler, S. Huhn (eds.), *Advanced Boolean Techniques*,
https://doi.org/10.1007/978-3-031-28916-3_9

parallel computation model, the MLP presents a worthwhile subject for hardware acceleration, specifically in real-time settings.

In this work, we propose a low-latency optimized architecture for MLPs. Mainly focusing on improving the latency of an inference, our architecture uses early evaluation by splitting activation functions in piecewise linear segments, specifically the ***PLAN*** approximation of the sigmoid proposed in [8]. Developed using high-level synthesis (HLS), the architecture's performance in terms of accuracy, throughput, latency, and power consumption is then analyzed in comparison with the state-of-the-art implementation. Specifically, we compare our architecture to another latency-driven design [3] and to highly tuned commercial IPs provided by the FPGA vendor [9]. In both cases, our proposed architecture has a significantly lower latency.

The rest of the paper is organized as follows: Sect. 2 presents and discusses the related work, Sect. 3 describes in detail the proposed MLP architecture, Sect. 4 displays the experimental results, while conclusions and future work are discussed in Sect. 5.

2 Related Work

Although developed for different applications, several studies have proposed accelerated implementations of the MLP; we discuss a representative set of them. In [10], the authors proposed an FPGA-based implementation of the MLP for gas classification. The proposed architecture uses a set of parallel lookup table (LUT)-based processing elements (PE), to handle the multiplications and the additions, while the activation functions were realized using read-only memory (ROM). The evaluation has proven a significant speedup in comparison with the software version. However, the use of memory to store support points to approximate the activation function limits the speedup as each neuron has to look up for the appropriate value in an address of a large number of elements, in this specific case 1001 support points. In [3], the authors proposed an architecture suitable for requirements given by an activity classification task. Similar to the aforementioned study, the multiplication-addition were carried in a parallel fashion. However, the ROM implementation of the activation function was replaced with a set of linear segments. The proposed implementation did show a 2x speedup as a result. Still, the architecture retained the sequential order of multiply-add-activate, which leaves room for further enhancements in the execution flow. In [1], the authors developed a 32-3-4 MLP neural network (32 input neurons, 1 hidden layer of 3 neurons and an output layer of 4 neurons) for a multispectral classification of satellite images. Using VHDL as the prototyping language, the implementation was 8 bits and was capable of achieving 670 ns of inference speed. Nonetheless, similar to the work in [10], the implementation of the activation function was memory based and following the same sequential order of neuron execution, not fully exploiting the parallel nature of both the MLP and FPGA. Finally, in [2], the authors used the Xilinx System generator to implement a classifier for blue whale calls. Despite not reaching a true

real-time performance, the implementation showed how flexible it is to use a higher level of abstraction to prototype the MLP model. The examples while showing varying techniques in implementing the activation function of an MLP on FPGA all share the sequential execution model of the single perceptron. The novelty of the architecture proposed here is in the parallel evaluation of the activation values, when segmented into a set of linear functions, providing significant decrease in design latency.

3 Implementation

In this section, we discuss the proposed architecture broken into the neuron model and MLP model, where we show the parallelization of the activation function. In addition, we discuss the optimizing pragmas inferred in the C/C++ implementation for HLS, to achieve the final desired performance.

3.1 Neuron Model

The neuron is considered the fundamental block in the structure of the MLP. Each of the neurons in a given layer receives a set of inputs, where each has to be multiplied by a set of weights tuned in the training process, before the accumulated summation of these products undergoes a given activation function. The mathematical representation of a single neuron is defined as follows:

$$y_j = f\left(\left(\sum_{i}^{n} w_{ji} x_i\right) + b_j\right) \tag{1}$$

where y_j is the output of the jth neuron, x_i is the ith input, of the neuron, w_{ji} is the ith element of the trained weight set of the jth neuron, while b_j is the bias of the neuron and f is the activation function. In this work, to have a fair comparison to the state of the art and to show the gains of our parallel architecture, we will focus on the sigmoid as the nonlinear activation function, forming the most computationally intensive part in our neuron model, which is given in the following formula:

$$S(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

Based on the function segmentation technique of nonlinear formulas, we adopt the ***PLAN*** function, as this is one of the best approximations of the sigmoid activation [8]. Table 1 shows the details of the ***PLAN*** function, while the proposed architectural model of the neuron is illustrated in Fig. 1. As shown in the **neuron**

Table 1 PLAN sigmoid approximation function

Sel	X	Y = F(x)
0	$0 \leq \|x\| < 1$	$0.25 \times \|x\| + 0.5$
1	$1 \leq \|x\| < 2.375$	$0.0125 \times \|x\| + 0.625$
2	$2.375 \leq \|x\| < 5$	$0.03125 \times \|x\| + 0.84375$
3	$5 \leq \|x\|$	1

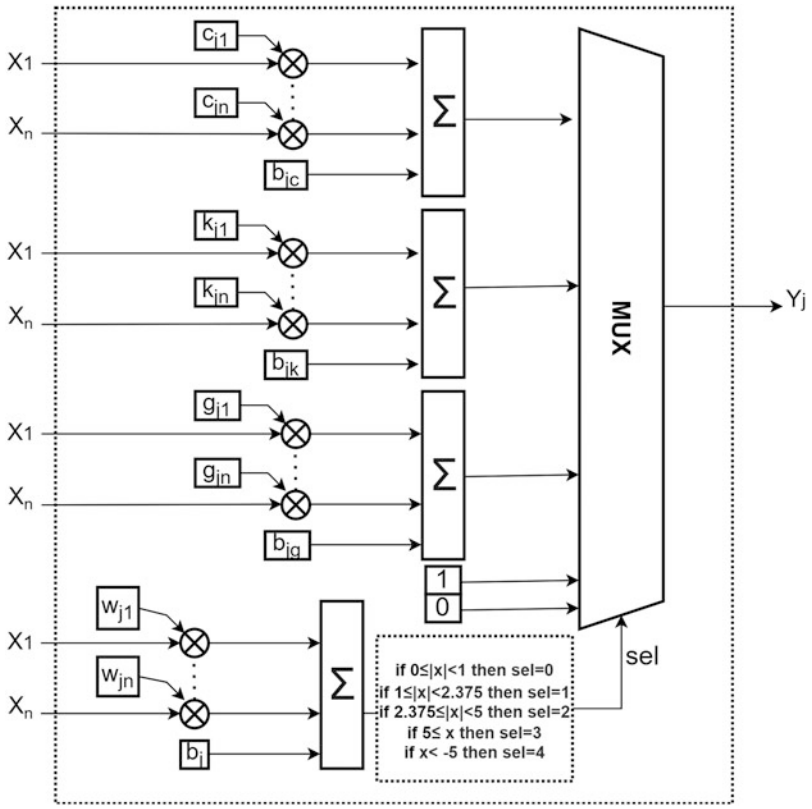


Fig. 1 Neuron model

model, the input-weight's multiplication and addition are parallelized for all activation possibilities at the same level of the activation selection, i.e., we evaluate all causes of the linear function segments while computing the selective lines.

In order to clarify further the calculations behind this architecture, let us take the first linear segment from Table 1 as an example. The linear function takes the sum of products $|x|$ and adds the bias of the neuron. Knowing the absolute value, the linear function therefore is evaluated if x is in the range $0 \leq x < 1$ or $0 \geq x > -1$; hence $-1 \leq x < 1$. The substitution at this stage is as follows:

$$F(x) = 0.25 \times \left(\left(\sum_{i}^{n} w_{ji} x_i \right) + b_j \right) + 0.5 \tag{3}$$

Given that w_{ji} and b_j are constants known a priori, the formula can be further reduced to

$$F(x) = \left(\left(\sum_{i}^{n} c_{ji} x_i \right) + b_{jc} \right) \tag{4}$$

where c_{ji} is the result of multiplying the w_{ji} with 0.25 and b_{jc} is the neuron b_j bias multiplied by 0.25 and added to 0.5. This applies analogously for the second and third formulas, while for values higher than 5 and or lower than -5, the output is either 1 or zero, respectively. This implementation allows for pre-computing the set of coefficients and biases that are needed to be multiplied and added to the neuron input, saving computational costs on the FPGA, and completely parallelizing the neuron computations with coefficients stored directly on the FPGA, shortening the length of the data path. This directly reduces the latency for inferring the output for given input samples.

3.2 MLP Model

In order to capitalize on the parallel architecture proposed for neuron computations, the MLP structure is also designed to process all neurons in a single layer all at the same time, as shown in Fig. 2. In addition, each input neuron has to be normalized, and a normal maximum has been added after the output layer, so the MLP can be used as a multi-class classifier. Given the nonlinear form of the soft-max, it naturally requires more computing resources and, more important, an expensive penalty to the inference latency. In compliance with the architectural optimization target of resource and computational complexity savings, the normal maximum has been chosen over the soft-max.

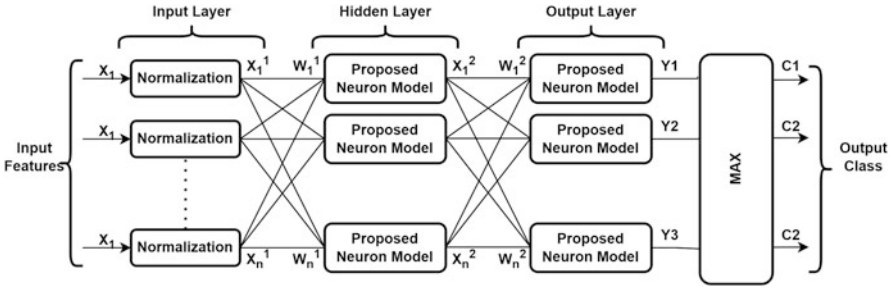


Fig. 2 MLP model

3.3 HLS Pragmas

Xilinx Vitis HLS [6] allows prototyping FPGA designs in C/C++ and provides a wide selection of compiler pragmas that can be used to enhance the performance of the design under development. In the proposed architecture, the following pragmas have been applied:

- ***Loop unrolling***: The loop unrolling pragma allows for the replication of operations found in the body of the loop to be executed in parallel, and therefore enhances the loop's performance. However, this pragma is considered one of the most resource consuming, and for large, high bounded loops, a partial loop unrolling option is recommended over full unrolling. In our architecture we adopt a relatively small MLP implementation, and therefore all loops on both neuron and MLP models have been fully unrolled.
- ***Pipelining***: The pipelining pragma allows for the concurrent execution of the operations in the selected region, and therefore increases utilization efficiency of the resources, either on loop, function, or top-level module levels. Similar to the case of loop unrolling, we adopt a top level pipelining style.
- ***Array Partitioning***: As weights, biases, and coefficients are all stored on the FPGA side for maximal performance, the implementation of this data was realized through the complete partitioning option of the pragma, i.e., using multiple registers instead of the Block-RAM (BRAM) memory.

Note, while prototyping in HLS, restrictions on the target processing elements for certain operations exist as well. In certain implementations, like in [3], digital signal processing (DSP) slices were the main processing elements for multiplications and additions. However, although a DSP is considered faster than an equivalent functionality in programmable logic (PL) for the same operations, routing the design can grow problematic, limiting the operational frequency. Therefore, we adopt a more relaxed approach in realizing the desired architecture, to achieve the fastest possible implementation with successful timing closure, by allowing the synthesis tool to alternate its operation target according to the requested clock time.

4 Experimental Results

In this section, we describe the development environment; discuss the performance of the proposed MLP accelerator in terms of accuracy, latency, resources, and power consumption; and finally compare the performance of our IP to the state of the art.

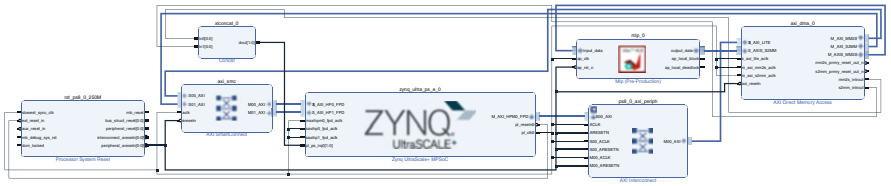


Fig. 3 System block design

4.1 Experimental Setup

This work addresses the human activity recognition application, classifying the activity based on accelerometer sensor data. The network topology is 7-9-9-9-5, a neural network of 7 inputs, 3 hidden layers of 9 neurons, and 5 outputs for classification. The aforementioned topology was the result of the MLP training on the UCI transition-aware human activity recognition dataset [7], using the TensorFlow framework. In this experiment we rely on seven features (body acceleration standard deviation of the three axes, the signal magnitude area of body acceleration, and the gravity acceleration mean of the three axes) in the classification target of five classes (sitting, walking, standing, activity transition, and laying).

The proposed MLP accelerator has been developed using Vitis HLS, Vivado, and Vitis Unified software 2021.2 using C/C++. In addition to the latency-optimized custom design in [3], our comparison includes the deep-processing unit (DPU), Xilinx commercial IP for neural networks processing. A single-cored DPU-512 implementation was used, and the neural network quantization and compilation were done using Vitis AI 2.0. The Zynq UltraScale+ ZCU104 evaluation board has been used to implement the proposed architecture and DPU designs. In Fig. 3, a system-level view of the proposed architecture can be seen. Given that an MLP usually falls within an extended processing pipeline, the MLP IP has been provided with AXI stream interfaces. The clock frequency adopted in our implementation was 300 MHz.

4.2 Model Accuracy

In Table 2, an exploration of the number of neurons and required number of hidden layers is presented. Each consecutive layer is preceded by a layer of the number of neurons that yield the highest accuracy. The maximum accuracy was achieved with a topology of three hidden layers, consisting of nine neurons each. Any additional increase on the number of the layers did not improve on the accuracy or even decreased it.

Table 2 Model's accuracy

	Layers				
Neurons	1	2	3	4	5
1	84.35	90.69	92.48	93.98	**95.2**
2	87.21	93.7	**94.59**	94.5	94.12
3	89.61	**95.3**	94.64	95.11	93.46
4	90.88	95.44	96.66	**97.41**	95.2
5	93.65	94.78	**98.16**	97.69	96.24
6	92.81	94.31	**98.77**	95.67	96.56
7	89.00	95.67	**98.40**	97.18	97.32
8	90.13	96.47	97.6	**98.59**	95.62
9	95.06	97.08	**98.87**	98.87	96.87
10	94.54	95.58	**97.55**	97.18	95.91

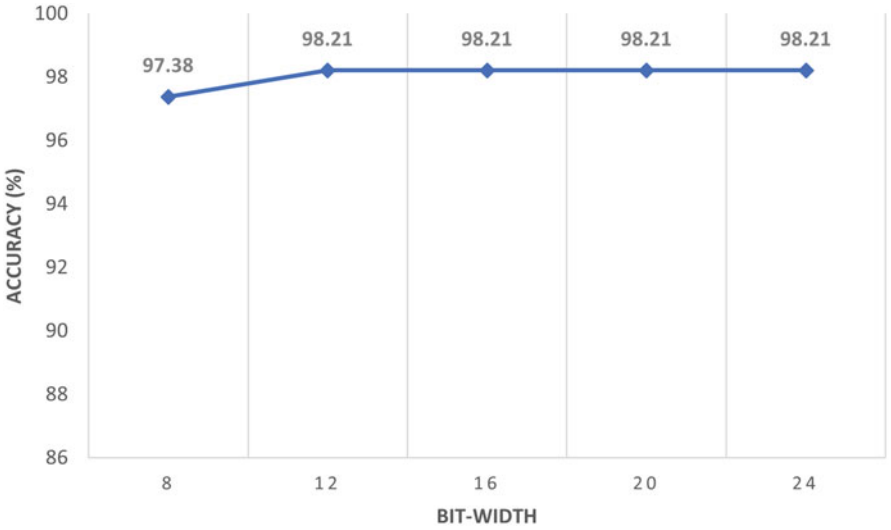


Fig. 4 Bit width of the weights vs accuracy

Next, we assess the impact of the ***PLAN***, the quantization of the model parameters (weights and activations), and the inputs/outputs on the accuracy of the model. When the sigmoid function is replaced with the ***PLAN***, the reduction in the model's accuracy was minimal with a drop of 0.67%. In Fig. 4, the accuracy of the MLP is presented when the weights and activations are quantized to 8, 12, 16, 20, and 24 bits. Note that the accuracy increase when increasing the bit width; however, a representation of 16 bits is sufficient, as any further increase will yield the same accuracy level with a higher cost in hardware. Finally, an assessment of the bit width of the inputs/outputs versus the model's accuracy is presented in Fig. 5. For the best accuracy level with minimal cost of hardware, a representation of 12 bits is sufficient.

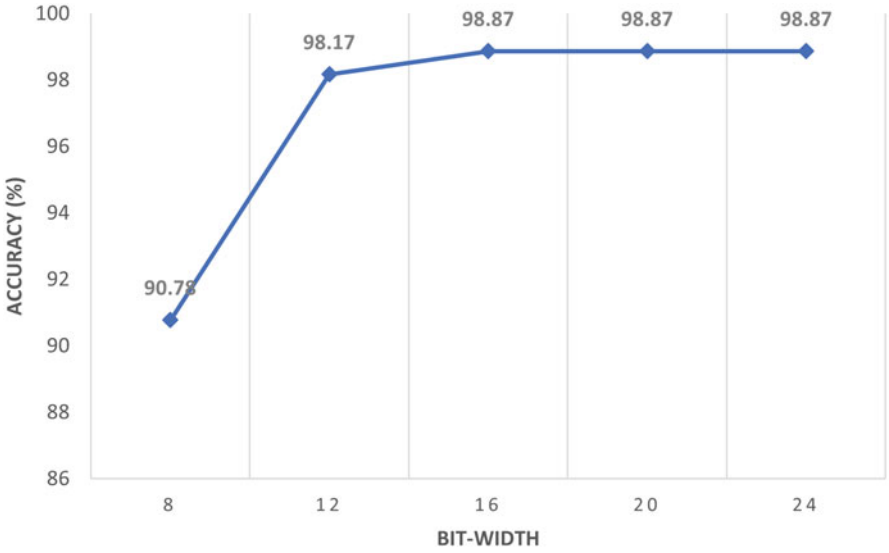


Fig. 5 Bit-width of the inputs/outputs vs accuracy

Table 3 Implementation details

Attributes					
Latency (ns)	LUT	FF	DSP	BRAM	Power (W): PL/PS
–	916	413	3	2	–
86.58	7476	9492	3	2	3.731: 0.976/2.755

In light of the previous discussion on the model's accuracy, a topology of 7-9-9-9-5, with model parameters and inputs/outputs of 16 bits, is the one selected for implementation. Table 3 lists the design latency, resources, and power consumption. The design resources for both the MLP itself and the entire design are listed as well. Note that the increase in hardware resources is due to the use of the AXI-DMA and AXI-Interconnect.

4.3 Comparison to the State of the Art

In order to fairly assess the performance gains of the proposed architecture, we compare between our IP and the implementation mentioned in [3] and the official DPU processor from Xilinx. Table 4 highlights the key features of each implementation. The 16-bit variation of the proposed architecture, with a single hidden layer of 6 neurons, is the one used in this comparative study, in line with the selection made in our state-of-the-art review in Sect. 2. Additionally, the clock frequency has been reduced to 100 MHz to match the implementation we are comparing to.

Table 4 Comparative analysis

	MLP Processor		
Attributes	Proposed	State-of-the-art [3]	DPUCZDX8G
Latency (ns)	130	270	43266
Power (W): PL/PS	0.719/2.743	0.241/-	2.006/2.768
Bits	16	16	8
Frequency (MHz)	100	100	300
DSP	3	81	110
LUT	6982	3466	27088
FF	9299	569	36052
BRAM	2	0	28
URAM	0	0	12

First, the proposed design is 2.1x and almost 332.81x times faster than the state of the art and a conventional neural accelerator like the Xilinx DPU. Second, the power consumption in the proposed accelerator is second to the design in [3]. One reason for this increase is the utilization of the Zynq processing system (PS), while in [3], the MicroBlaze soft processor is used. However, the use of the PS offers far more capable processing power that fits the domain requirements. A second reason is the increase in PL power due to the total parallelization of the activation function, which increased the resource utilization as a result. Third, it is noticeable that our proposed MLP accelerator has the lowest usage of DSP slices. Instead, PL implementations of mathematical operations were used to enable the synthesis tool achieving timing closure, by avoiding the routing complications to an extended number of DSPs, which could limit the design frequency.

5 Conclusion

In this work, a latency-optimized MLP design has been developed. The design architecture is based on the full parallelization of the network neurons, down to the internal computations of each single neuron. A topology of 7-9-9-9-5 has been proposed after an analysis of the model accuracy with respect to both parameters and input/output bit representations. Empirical results show that our latency-optimized MLP design outperforms a throughput-tuned state-of-the-art IP core tuned by the FPGA vendor by 332.81x orders of magnitude. Even another latency-optimized design has a significantly higher latency, for which our implementation is 2.1x faster. Our architecture performs inference for a small MLP architecture in only 130 ns enabling real-time inference for high-speed control.

Future work is aimed toward the inclusion of more segmented activation functions, and an online training hardware. We applied our technique to an MLP architecture using sigmoid activations. Nonetheless, various other architectures

can directly reuse this concept, most specifically the rectified linear unit (ReLU), hyperbolic tangent (Tanh), and soft-max activation functions, which are suitable for immediate parallelization.

Acknowledgments This work was supported in part by TUHH and HamburgX grant LFF-HHX-03 to the Center for Data and Computing in Natural Sciences (CDCS) from the Hamburg Ministry of Science, Research, Equalities and Districts.

References

1. Alilat, F., Yahiaoui, R.: Mlp on fpga: Optimal coding of data and activation function. In: 2019 10th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), vol. 1, pp. 525–529. IEEE, New York (2019)
2. Bahoura, M.: FPGA implementation of blue whale calls classifier using high-level programming tool. Electronics **5**(1), 8 (2016)
3. Gaikwad, N.B., Tiwari, V., Keskar, A., Shivaprakash, N.: Efficient FPGA implementation of multilayer perceptron for real-time human activity classification. IEEE Access **7**, 26696–26706 (2019)
4. Gupta, L.: Securing critical infrastructure through innovative use of merged hierarchical deep neural networks. In: 2021 18th International Conference on Privacy, Security and Trust (PST), pp. 1–8. IEEE, New York (2021)
5. Kocić, J., Jovičić, N., Drndarević, V.: An end-to-end deep neural network for autonomous driving designed for embedded automotive platforms. Sensors **19**(9), 2064 (2019)
6. Mousouliotis, P., Zogas, S., Christakos, P., Keramidas, G., Petrellis, N., Antonopoulos, C., Voros, N.: Exploiting vitis framework for accelerating sobel algorithm. In: 2021 10th Mediterranean Conference on Embedded Computing (MECO), pp. 1–5. IEEE, New York (2021)
7. Reyes-Ortiz, J., Anguita, D., Oneto, L., Parra, X.: UCI Machine Learning Repository: Smartphone-based Recognition of Human Activities and Postural Transitions Data Set
8. Tisan, A., Chin, J.: An end-user platform for FPGA-based design and rapid prototyping of feedforward artificial neural networks with on-chip backpropagation learning. IEEE Trans. Industr. Inform. **12**(3), 1124–1133 (2016)
9. Verucchi, M., Brilli, G., Sapienza, D., Verasani, M., Arena, M., Gatti, F., Capotondi, A., Cavicchioli, R., Bertogna, M., Solieri, M.: A systematic assessment of embedded neural networks for object detection. In: 2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), vol. 1, pp. 937–944. IEEE, New York (2020)
10. Zhai, X., Ali, A.A.S., Amira, A., Bensaali, F.: MLP neural network based gas classification system on Zynq SoC. IEEE Access **4**, 8138–8146 (2016)
11. Zhu, J., Chen, Y., Brinker, F., Decking, W., Tomin, S., Schlarb, H.: High-fidelity prediction of megapixel longitudinal phase-space images of electron beams using encoder-decoder neural networks. Phys. Rev. Appl. **16**(2), 024005 (2021)

Thirty-Six Officers of Euler-New Insights Computed Using XBOOLE

Bernd Steinbach and Christian Posthoff

1 Introduction

Leonhard Euler (1707–1783) is well recognized as one of the most productive scientists in both the amount and the significance of his contributions. He is especially known as a mathematician, but he worked also successfully in other scientific areas like Mechanics, Movement of the Planets, Fluid Dynamics, Optics, and others. The key for this success was the use of the analytical methods of Mathematics. His main contributions to Mathematics are related to Analysis and Number Theory.

A Latin square is a square of $n \times n$ fields where one of n symbols has been assigned to each field. The main rule of a Latin square is that each symbol occurs only once in each row and each column. The name *Latin square* goes back to Euler who used Latin characters as symbols; however, other symbols or numbers can alternatively form a Latin square. The number n is the *order* of a Latin square.

The well-known number puzzle Sudoku is a Latin square of order 9 with the additional condition that all symbols occur exactly once within nine sub-squares of the size 3×3.

Two Latin squares can be combined into a single square that contains in each field the concatenation of the symbols of the associated fields of the Latin squares given. Two such Latin squares are called orthogonal to each other if all n^2 combined

B. Steinbach (✉)
Institute of Computer Science, Freiberg University of Mining and Technology, Freiberg, Germany
e-mail: steinb@informatik.tu-freiberg.de

C. Posthoff
Department of Computing and Information Technology, The University of the West Indies, St. Augustine Campus, Trinidad & Tobago
e-mail: christian@posthoff.de

R. Drechsler, S. Huhn (eds.), *Advanced Boolean Techniques*,
https://doi.org/10.1007/978-3-031-28916-3_10

pairs of symbols are different. A combined square with this property is referred to as *Graeco-Latin square* or *Euler square*.

Leonhard Euler studied intensively such squares. He developed methods to construct Graeco-Latin squares of odd orders and orders of $n = 4k, k = 1, 2, \ldots$. In 1779, he explored the problem of the 36 officers with the aim to construct a Graeco-Latin square of order $n = 6$, but he did not find a solution. Euler was then of the opinion that Graeco-Latin squares of order $n = 4 \cdot k + 2, k = 0, 1, \ldots$ will not exist because he knew that there are no Graeco-Latin squares of order $n = 2$. R. C. Bose and S. S. Shrikhande rebutted this assumption in 1959 by finding counterexamples [2]. The same authors published together with E. T. Parker in 1960 a paper [3] containing the proof that Graeco-Latin squares of order $n = 4k + 2$ for $k = 2, 3, \ldots$ exist.

Recently a group of scientists from India and Poland mapped this problem into the quantum domain and proved that *absolutely maximally entangled (AME)* quantum states (4, 6) exist where $N = 4$ is the number of parties and $d = 6$ the local dimension [6]. However, we explore in this chapter a detailed quantified version of the original problem of the 36 officers specified by Euler.

2 Preliminaries

We are going to solve the problem of the 36 officers defined by Euler using the XBOOLE-monitor XBM 2. This software can be downloaded by everyone free of charge from the web page:

https://tu-freiberg.de/en/fakult1/inf/xboole/download.

A comprehensive help system supports the user of the XBOOLE-monitor XBM 2. Many examples to solve tasks from several areas by means of the XBOOLE-monitor XBM 2 are provided in [7].

The main data structure used in XBOOLE is the ternary vector list (TVL). We use TVLs in this paper to express DC-clauses of CDC-SAT equations (see Sect. 6.1) and (partial) solution sets of such equations. The TVL

$$\mathrm{ODA}(f) = \begin{array}{|ccccccccc|}\hline x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 & x_9 \\ \hline 1 & 0 & 0 & 0 & - & - & 0 & - & - \\ 0 & 1 & 0 & - & 0 & - & - & 0 & - \\ 0 & 0 & 1 & - & - & 0 & - & - & 0 \\ \hline \end{array}$$

has the form predicate ODA (orthogonal disjunctive or antivalence form) and can therefore be used to express function f in disjunctive (D) or antivalence form (A):

$$\begin{aligned} f &= x_1\overline{x}_2\overline{x}_3\overline{x}_4\overline{x}_7 \vee \overline{x}_1x_2\overline{x}_3\overline{x}_5\overline{x}_8 \vee \overline{x}_1\overline{x}_2x_3\overline{x}_6\overline{x}_9 \\ &= x_1\overline{x}_2\overline{x}_3\overline{x}_4\overline{x}_7 \oplus \overline{x}_1x_2\overline{x}_3\overline{x}_5\overline{x}_8 \oplus \overline{x}_1\overline{x}_2x_3\overline{x}_6\overline{x}_9 \end{aligned}$$

because the condition of orthogonality is satisfied by these three conjunctions. This TVL also expresses the solution set of the logic equation $f = 1$. Each of these three ternary vectors contains four dashes (−). Each dash represents a combination of the two values 0 and 1. The number of binary vector represented by each of these three ternary vectors is equal to $2^4 = 16$. Due to the orthogonality of ODA(f), the solution set of the equation $f = 1$ consists of 48 binary vectors. Only 3 ternary vectors express these 48 binary vectors. The use of dashes exponentially decreases the number of required ternary vectors.

XBOOLE provides more than 100 operations, which are very efficiently implemented. The most important operation of XBOOLE used in this paper is the intersection of two TVLs.

3 The Problem to Solve and Its Complexity

Leonhard Euler specified the problem as follows:

> "Six different regiments have six officers, each one belonging to different ranks. Can these 36 officers be arranged in a square formation so that each row and column contains one officer of each rank and one of each regiment?" [4]

The answer to this question can be YES or NO. That means that Euler specified a satisfiability problem (SAT) [1]. Looking for convenient Boolean variables to determine a possible SAT formula, we noticed that four words are used to characterize each of these officers: *row*, *column*, *regiment*, and *rank*. Unfortunately, three of these words begin with the letter "*r*." Using the military unit *brigade* instead of regiment and *grade* instead of rank, we get four words with different first letters. We define $36 \times 36 = 1296$ logic variables as follows:

$$x_{rcbg} = \begin{cases} 1 & \text{if the officer on the field determined by row } r \text{ and} \\ & \text{column } c \text{ belongs to brigade } b \text{ and has grade } g\ , \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

where each of the index variables $r, c, b, g = 1, \ldots, 6$. The encoding by variables of these four index values provides a direct specification of the row and column of the square as well as the brigade and grade of the 36 officers.

Using these variables, a requirement clause can be defined for each of the 36 fields. Each of these clauses consists of 36 variables connected by $\vee$-operations. We get, for example, the clause:

$$\left(\bigvee_{b=1}^{6} \bigvee_{g=1}^{6} x_{11bg} \right)$$

for field (1,1). This is a disjunction of 36 variables.

Equal index values (r, c) of two variables determine restrictions of the type $x_{1111} \wedge x_{1112} = 0$, which ensure one-hot encoding. That means that only one single officer can be assigned to each field. This restrictive equation can be transformed into a characteristic equation $\overline{x}_{1111} \vee \overline{x}_{1112} = 1$. The left-hand side of this equation is a clause. The number of clauses to ensure the one-hot encoding on all 36 fields is equal to

$$36 \cdot \frac{(35+1) \cdot 35}{2} = 22{,}680\,. \tag{2}$$

In order to exclude that the brigade of an officer does not occur on two fields of the same row, all six possible grades must be taken into account for each of the selected fields. For fields (1,1) and (1,2), we get the restriction

$$\bigvee_{b=1}^{6} \left(\bigvee_{g=1}^{6} x_{11bg} \wedge \bigvee_{g=1}^{6} x_{12bg} \right) = 0\,,$$

which can be transformed into a characteristic equation of $6 \cdot 36 = 216$ clauses

$$\bigwedge_{b=1}^{6} \left(\bigwedge_{g=1}^{6} \overline{x}_{11bg} \vee \bigwedge_{g=1}^{6} \overline{x}_{12bg} \right) = 1\,.$$

The number of clauses of these restrictions for all pairs of fields of each of the 6 rows is equal to

$$6 \cdot \frac{(5+1) \cdot 5}{2} \cdot 216 = 19{,}440\,. \tag{3}$$

Similarly, it can be excluded that the grade of an officer does not occur on two fields of the same row. For fields (1,1) and (1,2), we get the restriction

$$\bigvee_{g=1}^{6} \left(\bigvee_{b=1}^{6} x_{11bg} \wedge \bigvee_{b=1}^{6} x_{12bg} \right) = 0\,,$$

which can be transformed to a characteristic equation of $6 \cdot 36 = 216$ clauses

$$\bigwedge_{g=1}^{6} \left(\bigwedge_{b=1}^{6} \overline{x}_{11bg} \vee \bigwedge_{b=1}^{6} \overline{x}_{12bg} \right) = 1\,.$$

The number of clauses of these restrictions for all pairs of fields of each of the 6 rows is again equal to

$$6 \cdot \frac{(5+1) \cdot 5}{2} \cdot 216 = 19{,}440 \,. \tag{4}$$

The explored conditions must also be satisfied for all six columns. Exchanging the index r and c, we get the formulas needed to ensure that in each column each brigade b and each grade g occur only once. The number of clauses related to the columns is again two times 19,440.

The third rule is that an officer of both the same brigade b and the same grade g does not occur on two different fields. For fields (1,1) and (1,2) and chosen values $b = g = 1$, we get the restriction:

$$x_{1111} \wedge x_{1211} = 0 \,,$$

which can be transformed into equation

$$\overline{x}_{1111} \vee \overline{x}_{1211} = 1 \,,$$

where the left-hand side is a clause. The number of clauses of these restrictions for all pairs of fields and all 36 combinations of b and g is equal to

$$\frac{(35+1) \cdot 35}{2} \cdot 36 = 22{,}680 \,. \tag{5}$$

The number of clauses of a SAT formula to solve the problem of the 36 officers is therefore equal to

$$n_c = 36 + 2 \cdot 22{,}680 + 4 \cdot 19{,}440 = 123{,}156 \,. \tag{6}$$

It may be that some restrictive clauses satisfy more than one condition and are therefore counted twice. However, it is time-consuming to determine and exclude such doubled clauses.

It seems that the straightforward approach to solve the 36 officer problems using a SAT equation with 1296 variables and 123,156 clauses is not a convenient method to solve this problem. The search space of the 1296 logic variables would be $2^{1296} \approx 1.3 \cdot 10^{390}$.

There are several possibilities of a more compact encoding of this problem. The use of three Boolean variables to encode either the brigade b or the grade g of each field leads to $36 \cdot (3+3) = 216$ variables so that the search space is reduced to $2^{216} \approx 10^{65}$. However, this encoding increases the number of required variables of about 100,000 restrictive clauses from 2 to 6. The 22,680 clauses regarding the one-hot encoding are not needed in this approach, but the number of variables in the required clauses triples. Hence, this modified approach is also not favorable.

The number of Latin squares reduces the search space furthermore. Unfortunately, there is no formula to easily compute the number of Latin squares. However, it is known from the literature [8] that 812,851,200 Latin squares of order 6 exist.

Hence, the number of Graeco-Latin squares is equal to

$$812{,}851{,}200 \cdot 812{,}851{,}200 = 660{,}727{,}073{,}341{,}440{,}000 \approx 6.6 \cdot 10^{17} \tag{7}$$

and this number is in the range

$$2^{59} < 6.6 \cdot 10^{17} < 2^{60} .$$

Such a reduction to 60 Boolean variables also does not help us to solve the problem of the 36 officers because the constraints regarding repeated officers cannot be expressed based on these variables.

4 A Quantitative Specification of the Problem to Solve

Using a SAT solver we get either YES or NO as the answer to the problem of the 36 officers. The answer "there is no solution" of a SAT solver gives us no information how many different officers could be assigned to the fields of a Graeco-Latin square. We know from (7) that more than $6.6 \cdot 10^{17}$ Graeco-Latin squares exist and the two digits in each field of the square determine the brigade b and the grade g of the officer assigned to this field. To get a more significant answer to the problem of the 36 officers, we declare two additional quantitative questions:

1. How many officers of different brigades b and different grades g can satisfy the rules of a Graeco-Latin square of order 6?
2. How may such maximal assignments of officers of both a different brigade b and a different grade g exist on a Graeco-Latin square of order 6?

5 Approaches to Simplify the Problem to Solve

The one-hot encoding requires 1296 logic variables to specify all conditions of the problem of the 36 officers in a simple manner. A smaller number of variables, which satisfy the welcome property of the one-hot encoding, would reduce the effort to solve this problem. We start with $36 \times 36 = 1296$ Boolean variables where 36 variables characterize all 36 possible pairs of values $\langle b, g \rangle$ on each of the 6×6 fields. Figure 1a depicts this initial state.

Permuting pairs of rows or pairs of columns transforms a given Graeco-Latin square into another Graeco-Latin square. Hence, $6! \cdot 6! = 720 \cdot 720 = 518{,}400$ Graeco-Latin squares of order 6 belong to an equivalence class. It is enough to explore one representative of each of these classes. Knowing one solution for the representative, we can generate the other 518,399 Graeco-Latin squares of such a class by permutations of rows and columns.

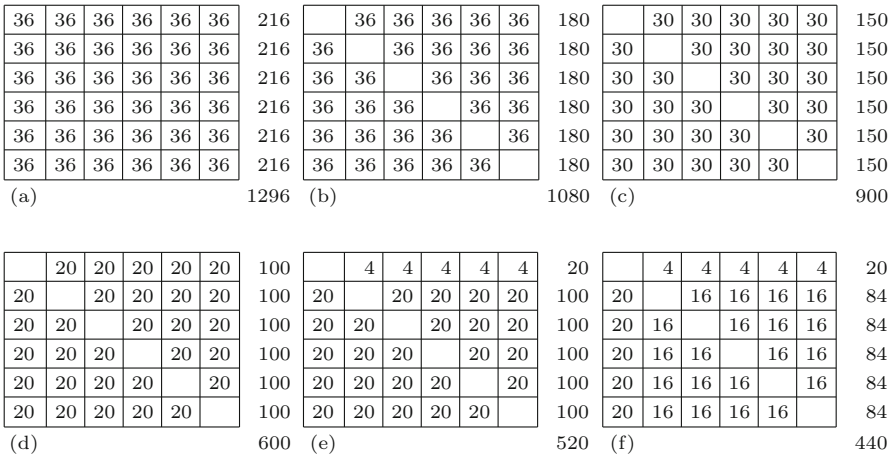


Fig. 1 Steps to exclude variables not required to specify the problem of the 36 officers

This approach of equivalence classes requires the definition of a representative. Such a representative can be constructed by permutations of rows and columns such that $b = 1$ and $g = r = c$ occur in the fields of the main diagonal.

The restriction to this representative allows us to reduce the number of Boolean variables required to solve the problem of the 36 officers. As a result of the fixed values in the main diagonal, we can exclude $6 \cdot 36 = 216$ variables, which express in the basic approach the values on the main diagonal. There remain $1296 - 216 = 1080$ variables as shown in Fig. 1b.

The fixed value $b = 1$ in the main diagonal prohibits six pairs $\langle b = 1, g \rangle$, $g = 1, \ldots, 6$, in all other fields. This knowledge allows us to exclude additionally $6 \cdot 5 \cdot 6 = 180$ variables leaving us with $1080 - 180 = 900$ variables as shown in Fig. 1c.

The representative of the equivalence class regarding $b = 1$ determines the values of g in the fields of the main diagonal as $g = r = c$. This knowledge allows us to exclude additionally $5 \cdot 5 \cdot 6 = 150$ variables x_{rcbg} with $r = g$ and also $5 \cdot 5 \cdot 6 = 150$ variables x_{rcbg} with $c = g$. That means that the number of required variables can be reduced from 30 to 20 on each field. Applying these two rules, $900 - 300 = 600$ variables remain as shown in Fig. 1d.

All six values of a brigade b must occur in each row. The value $b = 1$ is already used in the first row on field (1,1). There are $5! = 120$ permutations of the values b in the first row, which determine one more equivalence class. We use the rule $b = c$ for row $r = 1$ as representative of this equivalence class. The two equivalence classes are orthogonal to each other and can therefore be combined so that each common equivalence class represents $120 \cdot 518{,}400 = 62{,}208{,}000$ Graeco-Latin squares. As a result of fixed values $b = c$ for row $r = 1$ and columns $c = 2, \ldots, 6$, we can additionally exclude $5 \cdot 4 \cdot 4 = 80$ variables. There remain $600 - 80 = 520$ variables as shown in Fig. 1e.

The fixed values of b for row $r = 1$ and columns $c = 2, \ldots, 6$ allow us to exclude variables x_{rcbg} with $b = c$ in rows $r = 2, \ldots, 6$ and columns $c = 2, \ldots, 6$ which are additionally $5 \cdot 4 \cdot 4 = 80$. This number takes into account that variables x_{rcbg} with $r = g$ have been already excluded. Finally, $520 - 80 = 440$ variables remain as shown in Fig. 1f.

We are going to answer the two questions of Sect. 4 for the representative of the equivalence class specified in this section. We already know that 62,208,000 equivalent solutions can be generated for each found solution.

6 Method to Solve the Problem

6.1 CDC-SAT Model

We take field (1,2) as an example. The requirement clause (disjunction of variables) of the explored representative is

$$(x_{1223} \vee x_{1224} \vee x_{1225} \vee x_{1226}) \tag{8}$$

because the value b is fixed to 2. The variable $x_{rcbg} = x_{1221}$ is excluded due to $g = r = 1$, and the variable $x_{rcbg} = x_{1222}$ is excluded due to $g = c = 2$.

Six clauses ensure the one-hot encoding of these four variables:

$$\begin{aligned}(\overline{x}_{1223} \vee \overline{x}_{1224})(\overline{x}_{1223} \vee \overline{x}_{1225})(\overline{x}_{1223} \vee \overline{x}_{1226})\\ \wedge(\overline{x}_{1224} \vee \overline{x}_{1225})(\overline{x}_{1224} \vee \overline{x}_{1226})\\ \wedge(\overline{x}_{1225} \vee \overline{x}_{1226})\ .\end{aligned} \tag{9}$$

The conjunction of (8) and (9) results in

$$\begin{aligned}(x_{1223}\overline{x}_{1224}\overline{x}_{1225}\overline{x}_{1226} \vee \overline{x}_{1223}x_{1224}\overline{x}_{1225}\overline{x}_{1226}\\ \vee \overline{x}_{1223}\overline{x}_{1224}x_{1225}\overline{x}_{1226} \vee \overline{x}_{1223}\overline{x}_{1224}\overline{x}_{1225}x_{1226})\ .\end{aligned} \tag{10}$$

Such a disjunction of conjunctions (DC) is called a DC-clause.

The value $g = 3$ of $x_{rcbg} = x_{1223}$ prohibits the same grade in the other fields of the first row. Three clauses

$$(\overline{x}_{1223} \vee \overline{x}_{1443})(\overline{x}_{1223} \vee \overline{x}_{1553})(\overline{x}_{1223} \vee \overline{x}_{1663}) \tag{11}$$

ensure this condition because x_{1333} has been excluded due to $g = c = 3$.

Computing the conjunction of (10) and (11) extends the first conjunction of (10) by the second negated variables of the three disjunctions of (11) but does not change the other three conjunctions of (10) due to the absorption rule:

$$(x_{1223}\overline{x}_{1224}\overline{x}_{1225}\overline{x}_{1226}\overline{x}_{1443}\overline{x}_{1553}\overline{x}_{1663}\vee\overline{x}_{1223}x_{1224}\overline{x}_{1225}\overline{x}_{1226} \vee \overline{x}_{1223}\overline{x}_{1224}x_{1225}\overline{x}_{1226}\vee\overline{x}_{1223}\overline{x}_{1224}\overline{x}_{1225}x_{1226}) \ . \tag{12}$$

All other restrictions of the problem of the 36 officers have the same structure as shown in (11). Hence, the principle of extension demonstrated for the values of g in the first row can be applied for other all types of rules simply by adding negated variables to the associated conjunction of a DC-clause. The types of rules are:

1. one-hot encoding of the variables belonging to one DC-clause;
2. exclusion of value g of the non-negated variable in all other fields of the same row;
3. exclusion of value g of the non-negated variable in all other fields of the same column;
4. exclusion of value b of the non-negated variable in all other fields of the same row;
5. exclusion of value b of the non-negated variable in all other fields of the same column;
6. exclusion of the pair of values $\langle b, g\rangle$ of the non-negated variable in all other fields.

Rule 1 is required to model the problem as a conjunction of disjunctions of conjunctions SAT (CDC-SAT) equation. Rules 2 and 3 ensure that only Latin squares for grades g belong to the solution. Analogously, rules 4 and 5 ensure that only Latin squares for brigades b belong to the solution. Rule 6 prohibits that an officer of the same brigade b and the same grade g occurs in two or more fields.

All four conjunctions belonging to the DC-clause of field (1,2) consist of one non-negated and 31 negated variables. We generate each DC-clause directly as a ternary vector list (TVL). Each row of such a list represents one conjunction of a DC-clause. The non-negated variable is indicated by the value 1 in the ternary vector, and all negated variables appear as values 0 in the appropriate columns of the TVL. Figure 2 shows the generated conjunction of the non-negated variable $x_{rcbg} = x_{1223}$ and highlights the rules used to add the required variables to the conjunction.

	x_{1223}	x_{1224}	x_{1225}	x_{1226}	x_{1443}	x_{1553}	x_{1663}	x_{2123}	x_{2423}	x_{2523}	x_{2623}	x_{4123}	x_{4233}	x_{4243}	x_{4253}	x_{4263}	x_{4523}	x_{4623}	x_{5123}	x_{5233}	x_{5243}	x_{5253}	x_{5263}	x_{5423}	x_{5623}	x_{6123}	x_{6233}	x_{6243}	x_{6253}	x_{6263}	x_{6423}	x_{6523}
	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
rule 1: one-hot	•	•	•																													
rule 2: g row					•	•	•																									
rule 3: g column													•	•	•	•				•	•	•	•				•	•	•			
rule 4: b row																																
rule 5: b column																																
rule 6: b, g fields								•	•	•	•	•					•	•	•					•	•					•	•

Fig. 2 Rules that determine conjunction 1 of the DC-clause of field (1,2)

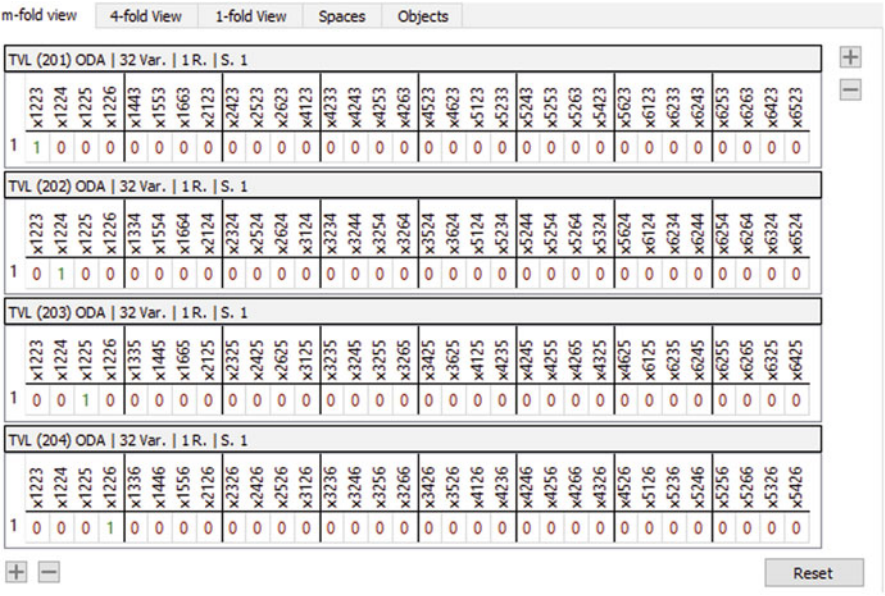


Fig. 3 Binary vectors of four conjunctions belonging to the DC-clause of field (1,2)

The application of rules 1 and 2 has already been explained above. Figure 2 shows that the conjunction with the selected value $g = 3$ in the non-negated variable does not contain any variable belonging to row 3 because $g = r = 3$ has been fixed for field (3,3) of the main diagonal and all variables with $g = 3$ have been excluded for this row. Brigade 2 is fixed as representative of the equivalence class in field (1,2). Therefore, only values 3, 4, 5, and 6 of b must be excluded in column 2. The associated 12 negated variables are indicated by bullets of rule 3. For the same reason, variables with $b = 2$ have been excluded so that no variables for rules 4 and 5 occur. Negated variables with the pair $\langle b, g\rangle = \langle 2, 3\rangle$ must be added to the conjunction of $x_{rcbg} = x_{1223}$ for all fields except row 1 due to $x_{rcbg} = x_{1223}$, row 3 due to $g = 3$ in field (3,3), column 2 due to $x_{rcbg} = x_{1223}$, column 3 due to $g = 3$ in field (3,3), and all fields of the main diagonal. There remain 13 negated variables indicated by bullets in the last line of Fig. 2.

The other three conjunctions of the DC-clause belonging to field (1,2) can be generated analogously. Figure 3 shows the generated four binary vectors of these conjunctions as separate TVLs in the m-fold view of the XBOOLE-monitor XBM 2.

The DC-clause of field (1,2) can be represented by a TVL of 4 ternary vectors and 116 columns. Variables x_{1223}, x_{1224}, x_{1225}, and x_{1225} are commonly used by all four vectors of Fig. 3. All other variables of these vectors are different. Therefore, we get $4 + (32 - 4) \cdot 4 = 116$ columns which correspond to the used variables of the DC-clause. The four ternary vectors are disjoint so that this TVL can take the form predicate ODA (orthogonal disjunctive or antivalence form). Each of these

	116	116	116	116	116
324		278	278	278	278
324	278		278	278	278
324	278	278		278	278
324	278	278	278		278
324	278	278	278	278	

Fig. 4 Numbers of columns of the TVLs that represent the 30 DC-clauses of the problem of the 36 officers

four ternary vectors contains $116 - 32$ dash elements $(-)$. The number of binary vectors represented by this TVL is equal to $4 \cdot 2^{116-32} \approx 7.737 \cdot 10^{25}$.

The TVLs of all DC-clauses can be generated in the same manner. The number of generated ternary vectors is equal to the number of variables shown in Fig. 1 (f). Larger numbers of variables belonging to the fields in rows 2–6 cause more negated variables in the conjunctions of the DC-clauses and also more variables in the associated TVL. Figure 4 shows the numbers of columns (variables) the TVLs of the generated 30 DC-clauses depend on.

The CDC-SAT equation (conjunction of disjunctions of conjunctions) consists on the left-hand side of 30 DC-clauses which are connected by AND and on the right-hand side the constant value 1. The solution of this equation requires the computation of 29 intersections of TVLs, which represent these DC-clauses. The number of partial solutions after a certain number of intersections of TVLs belonging to DC-clauses of certain fields determines the number of solutions for the used subset of fields. The maximal number of used TVLs of selected DC-clauses for which the number of solutions is greater than 0 answers the first question regarding the 36 officers specified in Sect. 4. The last computed number of solutions greater than 0 is the answer to the second question.

6.2 Practical Implementation Using the XBOOLE-Monitor XBM 2

The XBOOLE-monitor XBM 2 can be used to create TVLs and to compute intersections of them. Elementary tasks are specified in the XBOOLE-monitor XBM 2 by means of commands. These commands can be combined into problem programs (PRPs), which can be executed command by command, in an uninterrupted fashion until a break point is reached, or completely until the end. All operations are executed within several Boolean spaces. The user must specify the maximal number of variables for each of these spaces. We use a single Boolean space of 440 variables to solve the problem of the 36 officers.

The utilization of two equivalence classes allows us to reduce the number of required Boolean variables from 1296 to 440. We reduced the effort required to specify the problem of the 36 officers from more than 100,000 clauses of a classical SAT equation to only 30 DC-clauses of a CDC-SAT equation. Nevertheless, we are faced with a problem hard to solve.

The specification of all conditions of the problem regarding the 36 officers leads to a PRP of the XBOOLE-monitor XBM 2 that consists of several thousand commands. Therefore, we split the procedure to solve the problem into four subtasks:

1. defining a Boolean space of 440 variables, generating the 440 variables regarding the representative of the combined 2 equivalence classes, and creating equivalently structured TVLs that indicate the requirement rules in the remaining 30 fields of the square;
2. creating the TVLs of the DC-clauses of the first row of the square and computing the partial solution of this row;
3. creating the TVLs of the DC-clauses of the remaining five rows of the square;
4. computing the solution by intersections using the results of subtasks 2 and 3.

The PRPs of the first three subtasks require more than a thousand commands each. Therefore, we implemented simple but very fast (<1 s) programs using the programming language C++ that generate the required PRPs which are thereafter executed within the XBOOLE-monitor XBM 2.

Subtask 1 The C++ program of subtask 1 consists of 98 lines of code and generates a PRP with 1815 lines of commands. The rows of the generated TVLs are associated with the 36 pairs of values $\langle b, g\rangle$. The first six of these pairs $\langle 1, g\rangle$ are avoided because pairs with $b = 1$ determine the representative of the first equivalence class and have therefore been excluded from the computation. The rows of these TVLs store values 0 for variables x_{rcbg} using the indices $((b-2)\cdot 6)+g$. We store values 0 in these TVLs because zeros occur most often in the conjunctions of the DC-clauses. Values 1 of these variables can easily be computed. Figure 5 shows the TVL in D form (disjunctive form) for field (2,1). The index of these TVLs can be computed by $6 * (r - 1) + c$.

Subtask 2 The DC-clauses of row 1 are simpler than the DC-clauses of the remaining rows because fixed values b are used due to the second equivalence class. The C++ program of subtask 2 consists of 106 lines of code and generates a PRP with 1353 lines of commands. The generated TVLs of the five DC-clauses of row 1 are so simple that additionally the partial solution of row 1 is computed at the end of this PRP.

Subtask 3 The creation of the 25 TVLs of the DC-clauses of rows 2–6 is the most complex task. The use of a C++ program to generate the required PRP is very helpful for this subtask. Only 147 lines of C++ code suffice to generate the 42,867 lines of command for the PRP for this subtask.

Subtask 4 This subtask is computationally expensive but very easy to express. This PRP uses TVL 57 computed as a result of row 1 in subtask 2, TVLs 67–95 of the DC-clauses of rows 2–6, and computes 25 intersections to solve the problem of the 36 officers. The PRP consists of 17 lines and solves this task using 2 nested `for`-loops, which iterate over the rows and columns. Figure 6 shows the PRP that

$$
\mathrm{D}(f_7) =
\begin{array}{c|cccccccccccccccccccccccccccccccccccc}
 & x_{2123} & x_{2124} & x_{2125} & x_{2126} & x_{2133} & x_{2134} & x_{2135} & x_{2136} & x_{2143} & x_{2144} & x_{2145} & x_{2146} & x_{2153} & x_{2154} & x_{2155} & x_{2156} & x_{2163} & x_{2164} & x_{2165} & x_{2166} \\
\hline
 & - \\
 & 0 & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - \\
 & - & 0 & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - \\
 & - & - & 0 & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - \\
 & - & - & - & 0 & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - \\
 & - \\
 & - & - & - & - & 0 & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - \\
 & - & - & - & - & - & 0 & - & - & - & - & - & - & - & - & - & - & - & - & - & - \\
 & - & - & - & - & - & - & 0 & - & - & - & - & - & - & - & - & - & - & - & - & - \\
 & - & - & - & - & - & - & - & 0 & - & - & - & - & - & - & - & - & - & - & - & - \\
 & - \\
 & - & - & - & - & - & - & - & - & 0 & - & - & - & - & - & - & - & - & - & - & - \\
 & - & - & - & - & - & - & - & - & - & 0 & - & - & - & - & - & - & - & - & - & - \\
 & - & - & - & - & - & - & - & - & - & - & 0 & - & - & - & - & - & - & - & - & - \\
 & - & - & - & - & - & - & - & - & - & - & - & 0 & - & - & - & - & - & - & - & - \\
 & - \\
 & - & - & - & - & - & - & - & - & - & - & - & - & 0 & - & - & - & - & - & - & - \\
 & - & - & - & - & - & - & - & - & - & - & - & - & - & 0 & - & - & - & - & - & - \\
 & - & - & - & - & - & - & - & - & - & - & - & - & - & - & 0 & - & - & - & - & - \\
 & - \\
 & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & 0 & - & - & - & - \\
 & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & 0 & - & - & - \\
 & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & 0 & - & - \\
 & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & 0 & - \\
 & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & - & 0
\end{array}
$$

Fig. 5 One result of subtask 1: TVL 7 representing the requirement of field (2,1) expressed by values 0

computes the intersections between the partial solution of the first row and the TVLs of the DC-clauses belonging to the fields of rows 2–6.

7 Solutions for the First Row

Values b of the brigades are fixed in the first row due to the chosen representative of the equivalence classes and for the same reason value g of field (1,1) is constant 1. Different combinations of values g of grades can be used in fields (1,2), ..., (1,6) of the first row. There are $5! = 120$ permutations of the numbers 2, ..., 6 which can be used in a Latin square.

The representative used restricts the possible combinations of values g in the first row because the number of permitted values g is restricted to 4 due to $g = 1$ in field (1,1) and $g = r = c$ in the fields of the main diagonal. Using this restriction, we get a tighter upper bound on the number of permutations so that instead of $5! = 120$ only $4 \cdot 4! = 96$ permutations are required.

```
lds p6666_2
set $lastpss 57
for $r 2 6
(
for $c 1 6
(
if (ne $r $c)
(
set $f (add (mul 6 (sub $r 1)) $c)
set $rule (add $f 60)
set $newpss (add 100 $f)
isc $lastpss $rule $newpss ; main operation
set $lastpss $newpss
)
)
)
sts p6666_3_t_d
```

Fig. 6 PRP of subtask 4: computation of 25 intersections of DC-clauses

Table 1 All 44 permitted combinations of values g in the first row

$g(1,2)=3$	$g(1,2)=4$	$g(1,2)=5$	$g(1,2)=6$
⟨1, 3, 2, 5, 6, 4⟩	⟨1, 4, 2, 3, 6, 5⟩	⟨1, 5, 2, 3, 6, 4⟩	⟨1, 6, 2, 3, 4, 5⟩
⟨1, 3, 2, 6, 4, 5⟩	⟨1, 4, 2, 5, 6, 3⟩	⟨1, 5, 2, 6, 3, 4⟩	⟨1, 6, 2, 5, 3, 4⟩
⟨1, 3, 4, 2, 6, 5⟩	⟨1, 4, 2, 6, 3, 5⟩	⟨1, 5, 2, 6, 4, 3⟩	⟨1, 6, 2, 5, 4, 3⟩
⟨1, 3, 4, 5, 6, 2⟩	⟨1, 4, 5, 2, 6, 3⟩	⟨1, 5, 4, 2, 6, 3⟩	⟨1, 6, 4, 2, 3, 5⟩
⟨1, 3, 4, 6, 2, 5⟩	⟨1, 4, 5, 3, 6, 2⟩	⟨1, 5, 4, 3, 6, 2⟩	⟨1, 6, 4, 3, 2, 5⟩
⟨1, 3, 5, 2, 6, 4⟩	⟨1, 4, 5, 6, 2, 3⟩	⟨1, 5, 4, 6, 2, 3⟩	⟨1, 6, 4, 5, 2, 3⟩
⟨1, 3, 5, 6, 4, 2⟩	⟨1, 4, 5, 6, 3, 2⟩	⟨1, 5, 4, 6, 3, 2⟩	⟨1, 6, 4, 5, 3, 2⟩
⟨1, 3, 5, 6, 2, 4⟩	⟨1, 4, 6, 2, 3, 5⟩	⟨1, 5, 6, 2, 3, 4⟩	⟨1, 6, 5, 2, 3, 4⟩
⟨1, 3, 6, 2, 4, 5⟩	⟨1, 4, 6, 3, 2, 5⟩	⟨1, 5, 6, 2, 4, 3⟩	⟨1, 6, 5, 2, 4, 3⟩
⟨1, 3, 6, 5, 2, 4⟩	⟨1, 4, 6, 5, 2, 3⟩	⟨1, 5, 6, 3, 2, 4⟩	⟨1, 6, 5, 3, 2, 4⟩
⟨1, 3, 6, 5, 4, 2⟩	⟨1, 4, 6, 5, 3, 2⟩	⟨1, 5, 6, 3, 4, 2⟩	⟨1, 6, 5, 3, 4, 2⟩

The PRP of subtask 2 generates the DC-clauses of fields (1,2), ..., (1,6) and computes their intersections. The computed TVL 57 consists of 44 rows and 340 columns. Each row in this TVL contains a single value 1, 144 values 0, and 195 dashes. That means that this TVL expresses $44 \cdot 2^{195} \approx 2.2 \cdot 10^{60}$ binary vectors. This large number of binary vectors is again a proof of the efficiency of TVLs used in XBOOLE.

Each of the 44 rows determines exactly one possible combination of values g in 5 fields to the right of the first row. This number is smaller as the precomputed upper bound. Does a mistake in the PRP used cause this smaller number? A detailed exploration confirms that the computed result is correct. The number of permitted combinations of values g is really only 44. The reason for this restriction is that one of the numbers 2,..., 6 is excluded in each of the five fields to the right of the first row, and the excluded number differs from field to field. Table 1 enumerates

all 44 tuples of values g of the first row of the square of the 36 officers. It can be seen that for each of the 4 different values g of field (1,2) exactly 11 different combinations exist.

8 Analysis of the Computational Effort

The computation of the intersections in subtask 4 is most expensive due to the large number of variables involved. Figure 7 shows the numbers of ternary vectors after the use of n DC-clauses using a logarithmic scale.

This computation has been done using the DC-clauses of the fields row by row top down and from the left to the right within the rows. It can be seen that the number of rows of a TVL representing a partial solution grows until the maximum of 5,095,920 rows after the use of 18 DC-clauses. Thereafter this number decreases and reaches the value of 480 after the use of the DC-clause of field (6,3). The solutions for fields (6,4) and (6,5) are equal to 0. The time to compute all these 29 intersections using the XBOOLE-monitor XBM 2 on a PC with a processor Intel(R) Core(TM) i7-5960X CPU @ 3.00 GHz processor is 3.282 s.

A detailed study of Fig. 7 shows that the number of ternary vectors decreases when the DC-clause of the last field of a row has been included into the sequence of intersections. This welcome effect results from the restriction that the value of both b and g of the last fields of a row is already determined by these values in the other fields of the same row. The exploration of the numbers of rows after each intersection shows also that the computational effort depends on the order in which the TVLs of the DC-clauses are used. Figures 1f and 4 show that the largest TVLs of DC-clauses occur in the first column of the square. An approach that minimizes the effort therefore:

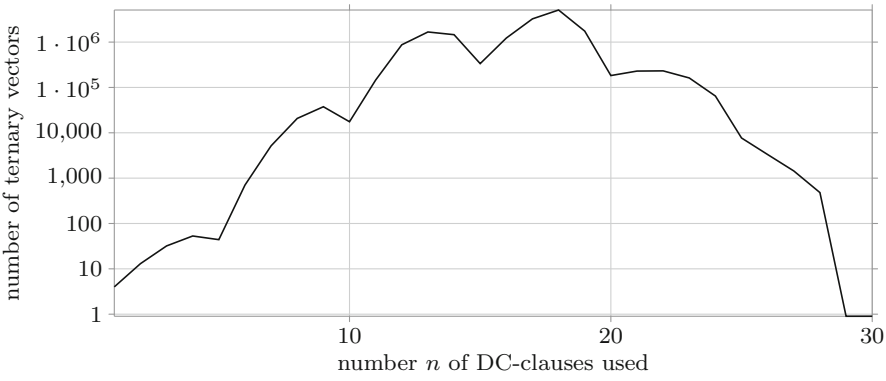


Fig. 7 Number of rows after each intersection using the DC-clauses of the fields from the left to the right in the rows and top down

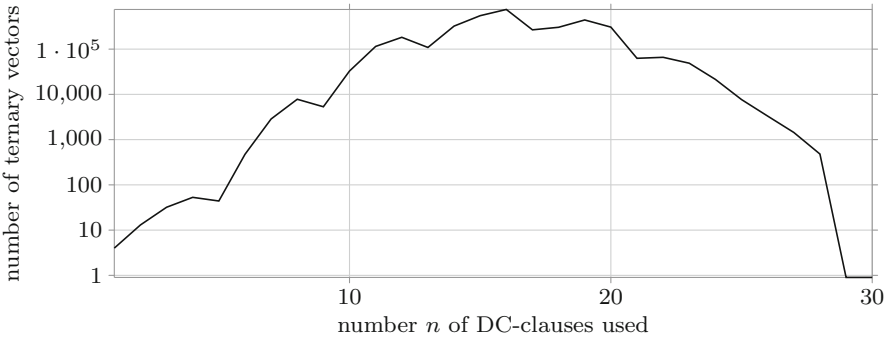


Fig. 8 Number of rows after each intersection using the DC-clauses of the fields in rows 2–6 column by column from the right to the left

1. computes the intersections of the DC-clauses of the first row from the left to the right;
2. computes subsequently the intersections of the DC-clauses in the order of columns from the right to the left and within each column from row 2 down to row 6.

The computation of the intersections in subtask 4 has additionally been executed in this modified order from the right to the left. Figure 8 shows the numbers of ternary vectors after the use of n DC-clauses using a logarithmic scale.

The maximal number of ternary vectors as the result of an intersection in this changed order is equal to 752,564 reached after the intersection of 16 DC-clauses. This is less than 15 percent of the previous approach. The solutions for fields (5,1) and (6,1) are equal to 0. The time to compute all 29 intersections using the XBOOLE-monitor XBM 2 has been reduced to 1.131 s for this changed order. The last not empty TVL also consists of 480 ternary vectors, and the results of the intersections for the fields (5,1) and (6,1) are equal to 0 so that again two free fields remain.

9 Detailed Evaluation of One Partial Solution

We evaluate the first vector of the partial solution where DC-clauses of the fields (6,4) and (6,6) are excluded. Using this vector, we fill the pairs of values $\langle b, g \rangle$ for which the function value 1 occur in this vector into the fields of a 6×6 square. The field to assign such a pair of values is determined by value r of the index that selects the row and value c of the index that selects the column. As a result of this procedure, we get 34 of 36 fields filled in the square as shown in Fig. 9a.

1,1	2,3	3,2	4,5	5,6	6,4
2,4	1,2	6,6	5,1	4,3	3,5
4,6	5,5	1,3	6,2	3,4	2,1
6,5	3,6	4,1	1,4	2,2	5,3
3,3	6,1	5,4	2,6	1,5	4,2
5,2	4,4	2,5			1,6

(a)

1,1	2,3	3,2	4,5	5,6	6,4
2,4	1,2	6,6	5,1	4,3	3,5
4,6	5,5	1,3	6,2	3,4	2,1
6,5	3,6	4,1	1,4	2,2	5,3
3,3	6,1	5,4	2,6	1,5	4,2
5,2	4,4	2,5	3,3	6,1	1,6

(b)

3,1 6,3

(c)

Fig. 9 Pairs $\langle b, g \rangle$ of the maximal solution of the problem regarding the 36 officers determined by the first of 480 ternary vectors computed for field (6,3): (**a**) computed partial solution, (**b**) extension of fields (6,4) and (6,5) regarding the rules of a Graeco-Latin square with highlighted repeated pairs of values $\langle b, g \rangle$, (**c**) missing pair of values $\langle b, g \rangle$

The evaluation of Fig. 9a confirms that this partially filled square:

1. satisfies rules 2 and 3 specified in Sect. 6.1 as a condition of a Latin square of values g;
2. satisfies rules 4 and 5 specified in Sect. 6.1 as a condition of a Latin square of values b; and
3. satisfies rule 6 specified in Sect. 6.1 as a condition to exclude the assignment of any pair of values $\langle b, g \rangle$ to more than one field.

Knowing the five pairs of values $\langle b, g \rangle$ in columns 4 and 5, it is easy to complete the assigned pairs of values to a Graeco-Latin square. Figure 9b shows this extension. The highlighted fields show that two pairs of values (officers) are assigned to two different fields each. However, this assignment violates rule 6.

The missing 2 pairs of values can be found by evaluation of the 34 different pairs of values of Fig. 9a. Figure 9c shows the two missing pairs of values $\langle b, g \rangle$.

The comparison of the two pairs of values $\langle 3, 3 \rangle$ and $\langle 6, 1 \rangle$ that extend the 34 assignments of Fig. 9a to a Graeco-Latin square as shown in fields (6,4) and (6,5) of Fig. 9b with the two missing pairs of different values shown Fig. 9c reveals an interesting property:

- exchanging values b in pairs $\langle 3, 3 \rangle$ and $\langle 6, 1 \rangle$ that complete the Graeco-Latin square results in pairs $\langle 6, 3 \rangle$ and $\langle 3, 1 \rangle$ which are exactly the missing pairs shown in Fig. 9c; and
- exchanging values g in pairs $\langle 3, 3 \rangle$ and $\langle 6, 1 \rangle$ that complete the Graeco-Latin square results in pairs $\langle 3, 1 \rangle$ and $\langle 6, 3 \rangle$ which are also exactly the missing pairs shown in Fig. 9c.

In summary, we can state that not permitting the exchange of either the values of grade g or brigade b in 2 fields of the 6×6 square prohibits the solution of the problem of the 36 officers specified by Euler more than 200 years ago.

The evaluated partial solution shown in Fig. 9a is a computed single representative of an equivalence class. All 62,208,000 equivalent results of this class can be

generated by means of permutations of rows, columns, and values $b = 2, \ldots, 6$. Due to the 480 computed representatives of equivalence class in the case of the excluded fields (6,4) and (6,5), we have already $480 \cdot 62{,}208{,}000 = 29{,}859{,}840{,}000$ partial solutions for this case.

10 Complete Evaluation

The model of Fig. 1f determines the number of ternary vectors of the TVLs for the 30 DC-clauses. We know from Sect. 8 so far only the partial solutions for two different excluded pairs of fields: $\langle(6, 4), (6, 5)\rangle$ and $\langle(5, 1), (6, 1)\rangle$. However, there are $\binom{30}{2} = \frac{(29+1)\cdot 29}{2} = 435$ different pairs of fields for which the DC-clauses can be excluded from the CDC-SAT formula. We computed all representatives of equivalence classes for all these 435 cases. Table 2 summarizes the computed results.

It is a property of the model being used that the brigades b of the first row are uniquely determined. Even if the DC-clause of one field of the first row is excluded from the CDC-SAT formula, the brigade b of this field remains implicitly specified. Therefore, we must distinguish in Table 2 the general cases where different values of b are possible in the two excluded fields (upper part of this table) and the special cases where the value of b is implicitly specified at least for one of the excluded fields (lower part of this table).

Table 2 shows that the number of representatives of equivalence classes of partial solutions of the problem of the 36 officers depends on the positions of the 2 excluded fields.

Table 2 Evaluation for all cases of pairs of unused fields f_1 and f_2

Condition	Number of cases	Number of representatives
	No fixed value of the brigade b	
$\langle f_1, f_2\rangle$ are in the same row	50	480
$\langle f_1, f_2\rangle$ in the same column	40	480
$(r(f_1) == c(f_2)) \wedge (c(f_1) == r(f_2))$	10	528
$(r(f_1) == c(f_2)) \oplus (c(f_1) == r(f_2))$	80	276
Otherwise	120	708
Sum of these cases	300	
	At least one fixed value of the brigade b	
$\langle f_1, f_2\rangle$ are in the same row	10	480
$\langle f_1, f_2\rangle$ in the same column	20	480
$(r(f_1) == c(f_2)) \wedge (c(f_1) == r(f_2))$	5	480
$(r(f_1) == c(f_2)) \oplus (c(f_1) == r(f_2))$	40	144
Otherwise	60	256
Sum of these cases	135	
Sum of all cases	435	

The first special relation of the two excluded fields is that these fields belong to the same row of the explored square. There are ten such cases in each of the six rows of the square. Mirroring of these cases on the main diagonal leads to a second special relation where the two excluded fields belong to the same column of the explored square. We computed 480 representatives for all these cases.

The third special relation of the two excluded fields is that these fields are located symmetric to the main diagonal. That means the index of the row of field 1 $r(f_1)$ is equal to the index of the column of field 2 $c(f_2)$ and the index of the column of field 1 $c(f_1)$ is also equal to the index of the row of field 2 $r(f_2)$. We computed 528 representatives for such pairs of excluded fields where the value of the brigade b is not fixed in one of these fields and 480 representatives otherwise.

The fourth special relation of the two excluded fields is a partial symmetry; either $r(f_1)$ is equal to $c(f_2)$ or $c(f_1)$ is equal to $r(f_2)$. We computed 276 representatives for such pairs of excluded fields where the value of the brigade b is not fixed in one of these fields and 144 representatives otherwise.

In the remaining cases, we computed 708 representatives for such pairs of excluded fields where the value of the brigade b is not fixed in one of these fields and 256 representatives otherwise.

All computed representatives are orthogonal to each other due to the condition that all officers must be different in both the brigade b and the grade g. The number of all representatives n_{ar} is therefore the sum of the products of the numbers of cases and the numbers of representatives listed in Table 2: $n_{ar} = 193{,}440$. This number of representatives already includes permutations of rows and/or columns for which the two excluded fields remain unchanged. However, the number n_{ar} has been computed under the conditions:

- the brigade $b = 1$ in the main diagonal, where at all 6 values are possible;
- the grade $g = r = c$ in the main diagonal, where at all 6!=720 assignments are possible; and
- the brigade $b = c$ in the first row, where at all 5!=120 assignments are possible.

Hence, the number of all partial solutions n_{aps} consisting of 34 of officers satisfying the condition of Euler and using all fields of the main diagonal is equal to the product of all representatives n_{ar} and the possible alternatives of the utilized equivalence classes $n_{ues} = 6 \cdot 6! \cdot 5! = 518{,}400$:

$$n_{aps} = n_{ar} \cdot n_{ues} = 193{,}440 \cdot 518{,}400 = 100{,}279{,}296{,}000 \approx 10^{11} .$$

11 Conclusion

We confirmed the assumption of Euler that there is no possibility to place 36 officers of 6 different brigades and 6 different grades on a 6×6 matrix so that they form a Graeco-Latin square. A straightforward description of all related conditions as usual

SAT formula would require 1296 Boolean variables and more than 100,000 clauses in the case that a one-hot encoding is used.

Utilizing equivalence classes of more than 60 million elements and the CDC-SAT approach, we modeled this problem using only 440 Boolean variables and 30 DC-clauses. Only one second was needed to compute the required 29 intersections using the XBOOLE-monitor XBM 2 in the case of an optimized order of the DC-clauses. Based on the complete evaluation for all pairs of excluded fields located out of the main diagonal and the utilized equivalence classes, we found that more than 10^{11} maximal assignments of 34 different officers partially satisfy the explored problem of the 36 officers.

The computed quantitative solution of a very hard Boolean problem confirms once more the power of both the XBOOLE-library and the XBOOLE-monitor XBM 2. A deep analysis of the problem and the utilization of the detected properties were important preconditions for this success.

References

1. Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T.: Handbook of Satisfiability, vol. 185. IOS Press, Amsterdam (2009). ISBN 978-1-58603-929-5
2. Bose, R.C., Shrikhande, S.S.: On the Falsity of Euler's Conjecture About the Non-Existence of two Orthogonal Latin Squares of Order 4t+2. In: Proceedings of the National Academy of Sciences of the United States of America, vol. 45(5), pp. 734–741 (1959)
3. Bose, R.C., Shrikhande, S.S., Parker, E.T.: Further results on the construction of mutually orthogonal latin squares and the falsity of Euler's Conjecture. Can. J. Math. **12**, 189–203 (1960). https://doi.org/10.4153/CJM-1960-016-5
4. Euler, L.: Recherches sur une nouvelle espéce de quarres magiques (1782). https://scholarlycommons.pacific.edu/euler-works/530/, visited on 29/05/2022
5. Posthoff, Ch., Steinbach, B.: Logic Functions and Equations—Binary Models for Computer Science, 2nd edn. Springer Nature, Cham (2019). ISBN 978-3-030-02419-2
6. Rather, S.A., Burchardt, A., Bruzda, W., Rajchel-Mieldzioc, G., Lakshminarayan, A., Zyczkowski, K.: Thirty-six entangled officers of Euler: Quantum solution to a classically impossible problem. arXiv:2104.05122 [quant-ph]. https://doi.org/10.48550/arXiv.2104.05122
7. Steinbach, B., Posthoff, Ch.: Logic Functions and Equations—Fundamentals and Applications using the XBOOLE-Monitor, 3 edn. Springer Nature, Switzerland (2022). https://doi.org/10.1007/978-3-030-88945-6
8. Wikipedia: Latin Square. visited on 2022-08-23. https://en.wikipedia.org/wiki/Latin_square

Start Small But Dream Big: On Choosing a Static Variable Order for Multiplier BDDs

Khushboo Qayyum, Alireza Mahzoon, and Rolf Drechsler

1 Introduction

With the size of *Integrated Circuits* (ICs) getting smaller and their functionality getting more complex, the task to assert the correctness of an IC becomes crucial. It is imperative that functionality of chips is thoroughly verified before silicon to prevent bugs from escaping into the final product. These bugs not only cause malfunctions but are a threat to the security of systems and a cause of monetary losses [1]. In this regard, formal verification techniques allow reliable verification of ICs using mathematical proof. Among formal verification methods, *Binary Decision Diagram* (BDD)-based methods are widely used to prove the correctness of ICs. A BDD is a canonical representation of a Boolean function as a *Directed Acyclic Graph* (DAG) [2]. Therefore, if two Boolean functions perform the same task, their BDDs will be the same if the input variables of both functions are in the same order regardless of how the Boolean function is defined. This attribute of canonicity allows two circuits to be easily compared and verified. State-of-the-art tools can perform this verification by a simple root pointer comparison of the BDDs of two Boolean functions [1].

However, one of the main challenges of using BDDs is to find a good order of the input variables. The size of BDDs is very sensitive to the input variable order; therefore, a good order of input variables may produce a size of BDD within polynomial order, but a bad order can cause the size of a BDD to be of exponential

K. Qayyum (✉) · A. Mahzoon
University of Bremen, Bremen, Germany
e-mail: khushboo@uni-bremen.de; mahzoon@uni-bremen.de

R. Drechsler
Institute of Computer Science, University of Bremen and Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
e-mail: drechsle@uni-bremen.de

R. Drechsler, S. Huhn (eds.), *Advanced Boolean Techniques*,
https://doi.org/10.1007/978-3-031-28916-3_11

order. This attribute of a BDD calls for the choice of input variable order to be perceptive. A number of different heuristics have been developed in the past that try to associate the arrangement of input variables with different aspects of circuits' architecture or by using searching and sorting techniques [3–7]. These heuristics can be divided into two main categories based on when the input order is applied to the BDD construction. In *static variable ordering* heuristics, the input variable order is arranged and decided before the construction of BDDs, and in *dynamic variable ordering* heuristics, the input variable order is applied during the construction of BDD.[1]

The erratic behavior of the BDD size is not just limited to the ordering of the input variables, but the size is also sensitive to the structure of underlying function. This behavior is particularly evident in BDDs of complex arithmetic circuits. Arithmetic circuits make integral part of ICs; therefore, their correct functionality is essential. Bugs like Pentium FDIV can render a chip useless if they are not identified before silicon. Within the category of arithmetic circuits, *multipliers* have piqued the interest of researchers for a long time as the BDDs of multiplier circuits tend to explode in size even at substantially small multipliers [2]. Due to this explosion, constructing the BDDs for multipliers requires tremendous hardware resources [8]. Furthermore, using systems with insufficient resource requirements lead to prolonged runtimes only to result in failure at the end. Keeping the concern for the size of BDDs for multipliers in mind, the choice of an appropriate input variable ordering for a multiplier becomes paramount as a good input variable ordering can help in reducing this size explosion. Additionally, an early estimation of memory requirement can facilitate an appropriate selection of resources and thus save time and effort.

Contribution In this paper, we present a methodology to choose an optimal static variable ordering heuristic for larger multipliers with an early estimation of the *endsize*, *peaksize*, and *memory* required for constructing the BDD nodes. Our proposed methodology allows a fast and resource-efficient optimal static variable ordering heuristic selection. The estimation of the size and memory requirements of the large multiplier allows a more insightful selection of resources for the BDD construction. In our methodology, we first obtain the smaller version of the target multiplier and perform analysis using various static variable ordering heuristics. Our results show that a static variable ordering heuristic that is optimal for a scaled-down circuit is also optimal for the larger circuit while requiring only a fraction of time and memory resources. For the endsize estimation, we reuse chosen heuristics and incrementally increase the size of the circuit to estimate the endsize and the peaksize of the larger circuits. Using the estimated peaksize of the BDD, a conservative estimation of the memory requirements for the BDD nodes can be determined with high accuracy. We perform extensive experiments on multipliers obtained using the GenMul [9] multiplier generator.

[1] In this work, we overlook dynamic variable ordering heuristics as these heuristics can be counterintuitively slow and thus prohibitive for complex circuits like multipliers.

2 Related Work

Multipliers have been a subject of interest for a long time, and multiple strategies have been developed specifically to address the size complexity of the multiplier BDDs. In [10], the authors address the complexity of multipliers BDDs through the introduction of input variables although the complexity is reduced but not solved completely. Recently, in [11] the authors also present a method to decrease the complexity of verification of the multiplier BDDs; however, an optimal variable ordering is not addressed. Multiple different static variable ordering heuristics have been developed in the past. In [5–7] authors exploit circuit architecture to come up with a suitable input variable ordering, whereas in [3, 4], searching algorithms are applied to the circuit to find a good input variable ordering. Most of these consider only the endsize of the BDD and only for a single output.

The memory usage issue of BDDs is addressed by several works [12–15] using different approaches in constructing and manipulation of the BDDs. The estimation of size of BDDs has been attempted using timed automata by [16, 17] for generic circuits.

Our work differs in how the problem is approached with only multipliers as our target circuits. We focus mainly on choosing an optimal static variable ordering heuristic from the already developed heuristics. For calculations, we consider all the outputs for endsize and also consider the peaksize of the BDDs. Additionally, we focus on memory estimation instead of memory management.

3 Preliminaries

3.1 Binary Decision Diagrams

BDDs are a tree-like representation of a Boolean function created using Shannon expansion. Once ordered and reduced, these *Reduced Ordered BDDs* (ROBDDs) form a canonical DAG for the given Boolean function. In our work, we refer to the ROBDDs as BDDs. The canonicity of the BDDs is indifferent to the architecture of the underlying function. That is, given two Boolean functions, their BDDs will be the same if :

- both the functions perform the same tasks regardless of the underlying architecture and
- both the graphs are made with input variables arranged in the same order.

This makes the comparison of two circuits a trivial task. For this reason, the BDDs are favored in the area of formal verification. However, the size of BDDs and their sensitivity to certain circuit architectures sometimes undermines its performance and ease. The ordering of input variables of a Boolean function heavily influences the size of its BDD. Thus, a good choice of the input variable

order plays a decisive role in finding an optimally sized BDD. Although finding an optimal input variable order for a BDD is NP-hard [2], various heuristics have been developed to address this problem. Using these heuristics, the input variables can be arranged during the construction of BDD using dynamic variable ordering heuristics or before the construction of BDD using static variable ordering heuristic like the ones given below.

- **Initial Order:** Input variable order as they are defined in the circuit description.
- **Reverse Order:** Reverse of the initial order.
- **Dependency Order:** The variables influencing more outputs of the circuit get precedence [7].
- **Depth-First Search Order:** Depth-First Search (DFS) is used to determine input variable order [3].
- **Fanin Order:** The inputs that are deeper in the circuit get precedence [6].
- **Fanout Order:** The inputs with more fanouts in the circuit get precedence [4].
- **Random Order:** The input variable order is generated randomly.
- **Breadth-First Search Order** Breadth-First Search is used to determine the ordering for the given circuit [3].

Although dynamic variable ordering heuristics are capable of producing better outcomes, their excessive runtimes make them counterproductive. For this reason, in this work, we only focus on static variable ordering heuristics.

3.2 Multipliers

Multipliers are essential components in modern ICs. Many pivotal applications like encryption, digital signal processing, etc. require multipliers. Different types of multiplier architectures have been developed over time to satisfy demands in aspects like power, speed, area, and accuracy. When individually compared, these architectures are apparently different, but based on their internal functions, a multiplier can be broadly represented as a three-stage structure as represented in Fig. 1. Each of these stages performs the following task:

- *Partial Product Generator* (PPG): generates the partial product from the multiplier and multiplicand.
- *Partial Product Accumulator* (PPA): reduces and aggregates the partial products.
- *Final Stage Adder* (FSA): sums up all the result of the PPA to generate the final product.

The PPGs for multipliers can be implemented using simple AND gates and Booth encoding algorithm. Likewise, some of the examples of PPAs are array, Wallace tree, counter-based Wallace, and Dadda tree algorithms. For the FSA stage, architectures like Brent-Kung, ripple carry, carry look-ahead, Lander-Fischer, Kogge-Stone, and Carry-skip can be used.

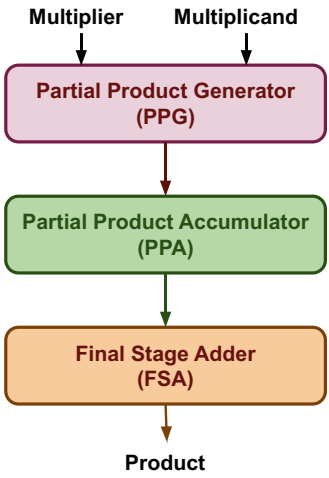


Fig. 1 Multiplier structure

4 Methodology

In this section, we explain our proposed methodology. First, we present an overview and later we explain each step of our methodology.

4.1 Overview

The overview of our proposed methodology is illustrated in Fig. 2. Our proposed methodology is comprised of two steps. In the first step, shown in Fig. 2 by the red solid line, a smaller version of the target multiplier circuit is obtained. The behavior of BDDs for a number of different static variable ordering heuristics is observed for the smaller scaled-down circuit, and the most optimal heuristic is identified. Once the optimal heuristic is selected, the first step of the methodology is concluded and the second step begins. In the second step, shown as a blue dashed line in Fig. 2, we build a set of circuits which are also scaled-down versions of the target multiplier. The circuits in this set have the same structure but incrementally increase in size. The optimal static variable ordering heuristic is applied to this set of circuits, and different parameters related to the size of BDDs of each circuit are collected. Using these parameters, the endsize, peaksize, and memory requirement of the target multiplier are estimated.

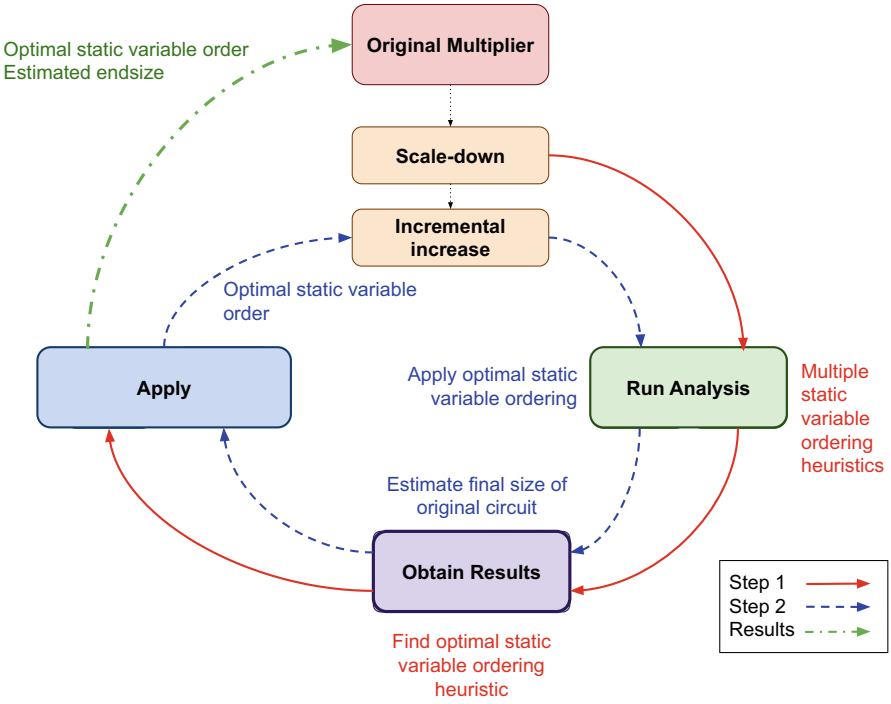


Fig. 2 Proposed two-step methodology

4.2 Optimal Static Variable Order Selection

The first step of our proposed methodology aims to find an optimal static variable ordering heuristic for the target multiplier. This process is represented in Fig. 2 by the red solid line. The idea is to work on smaller version of the target circuit to find the optimal static variable ordering heuristic and then use it for target multiplier. This helps to reduce the time and resources required for constructing the BDDs that are otherwise substantial when the larger multiplier is used. Consider a target circuit that is a 64 $\times$ 64 bit signed multiplier with a Array PPA and a Brent-Kung FSA. The scaled-down version is a 8 $\times$ 8 bit signed multiplier with the same Array PPA and a Brent-Kung FSA. We use GenMul [9] to obtain multiplier circuits in various structures and sizes. Our proposed methodology is agnostic to the underlying tools, thus, any BDD construction tool can be adopted that provides information about the endsize and the peaksize of the BDD. In order to find the optimal variable ordering heuristic, we use an in-house framework with CUDD [18] at its heart. This framework finds input variable orders for a given circuit using different heuristics and then uses these input variable orders to construct BDDs for the circuits. In addition to the BDD construction, the framework also monitors different parameters of BDDs like peaksize during construction and endsize. Since the GenMul tool only

Table 1 Results of static variable ordering heuristics for the same multiplier in different sizes

Static variable ordering heuristic	Circuit size		
	8×8	10×10	12×12
Initial order	6480	50,085	391,891
Reverse order	**6386**	**48,876**	**374,537**
Dependent order	12,916	129,321	1,276,275
Fanin order	6969	51610	390512
Fanout order	6480	50085	391891
Random order	16710	230951	2524622
BFS appending	12618	123378	1178732
Initial order interleaved	12916	129321	1276275

provides circuits in Verilog, we convert the circuits into bench format that can be processed by our framework. This can be done using the Yosys tool [19].

The framework generates input variable orderings for the scaled-down multiplier using different heuristics. Using the generated input variable orders, it constructs the BDDs for the circuit and records their endsizes and peaksizes in a database. The details of these static variable ordering heuristics are given in Sect. 3.1. Once the results of all the static variable ordering heuristics are available, the optimal heuristic is chosen. The choice of the optimal heuristic can be based on different factors, e.g., smallest endsize and resource usage. With the decision of the optimal static variable ordering heuristic, the first step of the methodology concludes, and the second step for the endsize and peaksize estimation for the target multiplier begins.

Revisiting the earlier example, Table 1 shows the endsizes of the Array PPA and Brent-Kung FSA multiplier for three different sizes. The bold values show the smallest values. It can be seen that heuristic that performs well for 8×8 also performs well for the larger 10×10 and 12×12 multiplier thus in line with our claim that the heuristic that performs well for smaller circuits also performs well for larger circuits.

4.3 BDD Endsize and Peaksize Estimation

In the second step, we estimate the endsize and the peaksize of the target multiplier. Since the BDDs of multipliers usually explode in size, an estimation of the peaksize and endsize can help in projecting the required resources for the construction of their BDDs. This process is shown by the blue dashed line in Fig. 2. In this step, we use the optimal static variable ordering heuristic that is previously selected in the first step of the methodology. We obtain two or more circuits from the GenMul tool with the same structure to perform the experiments. These circuits are slightly larger than the scaled-down version of the target multiplier. In our set of circuits, we obtain multipliers with the same architecture with 8×8, 9×9, and 10×10 bit-size. Our framework constructs the BDDs of these circuits and extracts vital information such

as the endsize and peaksize of the BDDs of each circuit using the selected static variable order heuristic. Once this information is available, trends are observed with respect to the growth of BDD. Using these trends, the growth factors are calculated, and these growth factors are used to estimate the endsize and peaksize of the target multiplier using the following equations:

$$\hat{e_y} = {d_e}^{y-x} \times e_x \tag{1}$$

$$\hat{p_y} = {d_p}^{y-x} \times p_x \tag{2}$$

where $\hat{e_y}$ and $\hat{p_y}$ are the estimated endsizes and peaksizes of the target multiplier and e_x and p_x are the endsize and peaksize of the scaled-down multiplier. x is the bit-size of the inputs of the scaled-down multiplier and y is the bit-size of the input of the target multiplier. d_e and d_p are the growth factors per bit of the endsize and peaksize, respectively, for the given multiplier structure. The peaksize of a BDD shows the maximum number of nodes that were created throughout the construction of the BDD. Therefore, the peaksize dictates the memory consumption during the construction of a BDD. Using the peaksize and the memory required by a single node, the estimation of the memory required by the target BDD is calculated as follows:

$$memory\_required = \hat{p_y} \times size\_per\_node \tag{3}$$

However, the memory estimation is conservative as they do not include the auxiliary memory that maybe required for processing. Regardless, they can allow for a more insightful resource allocation and thus produce practical runtimes for BDD construction.

5 Experiments

In this section, we present the experimental results of the proposed methodology to select the optimal static variable ordering heuristic and estimation of peaksize and endsize. In our work, we obtained different multiplier structure combinations using the GenMul tool. We applied our methodology to a wide range of multiplier structures, but for brevity, we present results for only a few static variable ordering heuristics and multiplier structures.

5.1 Selection of Optimal Heuristic

Figure 4 shows the endsize and peaksize of different static variable ordering heuristics for four different 8×8 multiplier structures. The naming of the multipliers

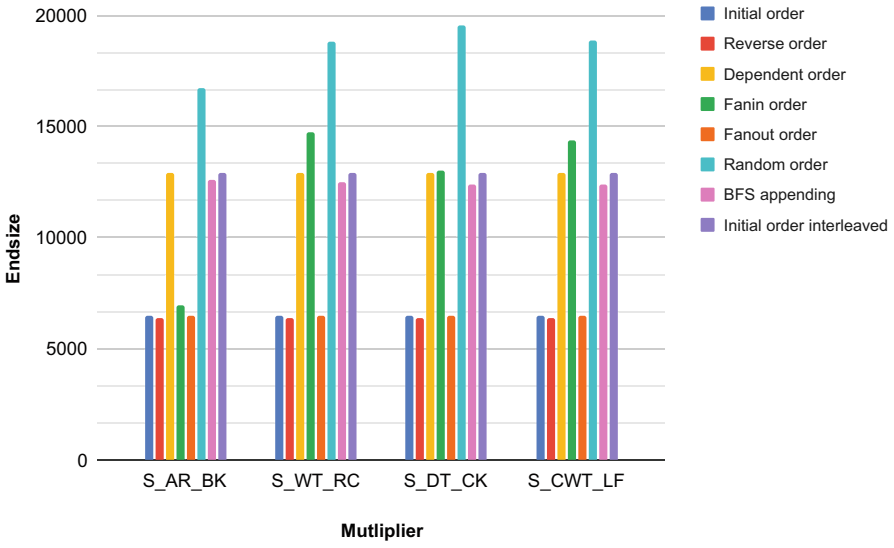
in the figures is in the X_A_B format where X is the PPG (S = signed simple, U = unsigned simple), A is the PPA type (AR = Array, WT = Wallace tree, DT = Dadda tree, CWT = counter-based Wallace tree), and B represents its FSA type (BK = Brent-Kung, RC = ripple carry, CK = carry skip, LF = Ladner-Fischer). The x-axis represents the multiplier structure and the y-axis shows the number of nodes. When choosing the optimal static variable ordering heuristic based on the endsize, the lowest value would be considered. From Fig. 3a which shows the endsize for signed multipliers, it can be seen that the *reverse* order performs well for all the multiplier structures, but in Fig. 4a which shows the endsize of unsigned multipliers, *initial* and *fanout* ordering heuristics outperforms the other heuristics. The initial and reverse ordering heuristics are less intuitive, but it is interesting to see that there are other heuristics like the fanout ordering that produce similar results as that of initial order. And consequently its performance would match the reverse order. While some heuristics look oblivious to the structures, heuristics like *fanin order* seem to be affected by the structure of the multiplier. The random order performs the worst for all the selected structures which reinforces that the selection of input variable order should be rational.

Figures 3b and 4b show the peaksizes using different static variable ordering heuristics for signed and unsigned multipliers, respectively. When selecting the static variable ordering heuristic based on the peaksize, the difference seems to be less obvious. It seems so because on average the peaksize is $10x$ larger than the endsize; therefore, the difference between orders is less evident. Within our selected signed multiplier structures (Fig. 3b), the reverse order performs well but this is not universal. For the unsigned multiplier of Array and Brent-Kung combination (U_AR_BK) as evident in Fig. 4b, the BFS produces a much smaller peaksize and thus would consume fewer resources and therefore would be the choice for optimal heuristic when the peaksize is considered.

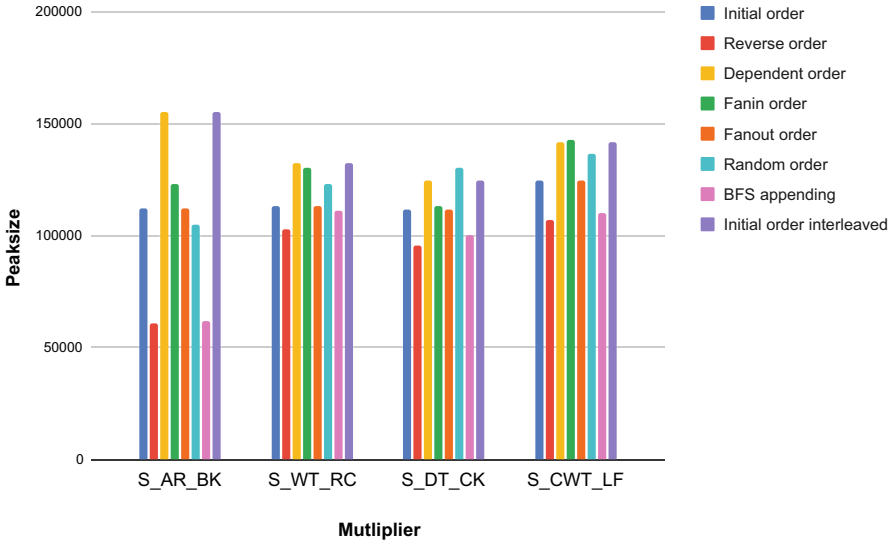
5.2 Estimation of Endsize and Peaksize of the BDDs

From the results of step 1, as evident from Fig. 3a, the reverse order was selected as the optimal static variable ordering heuristic for the multiplier structure S_AR_BK. Since our scaled-down version of multiplier was 8×8, the circuits in this set for estimation are of size 9×9, 10×10, and 11×11. Although a set of three increments would suffice, more incremental circuits would result in a better estimation. We obtained a set of circuits of all the multiplier structures with incremental increase in size. However, due to space constraints, we only show the result for one of the multiplier structure, i.e., signed multiplier with Array PPA and Brent-Kung FSA (S_AR_BK). As expected in our results, the growth factor of the peaksize was slightly greater than the endsize (2.8 and 2.6, respectively).

Figure 5a, b shows the estimated endsize and peaksize using Equations 1 and 2 for our selected multiplier structure. The x-axis represents the size of the multiplier in bits, and the primary y-axis shows the number of nodes, and the secondary y-axis

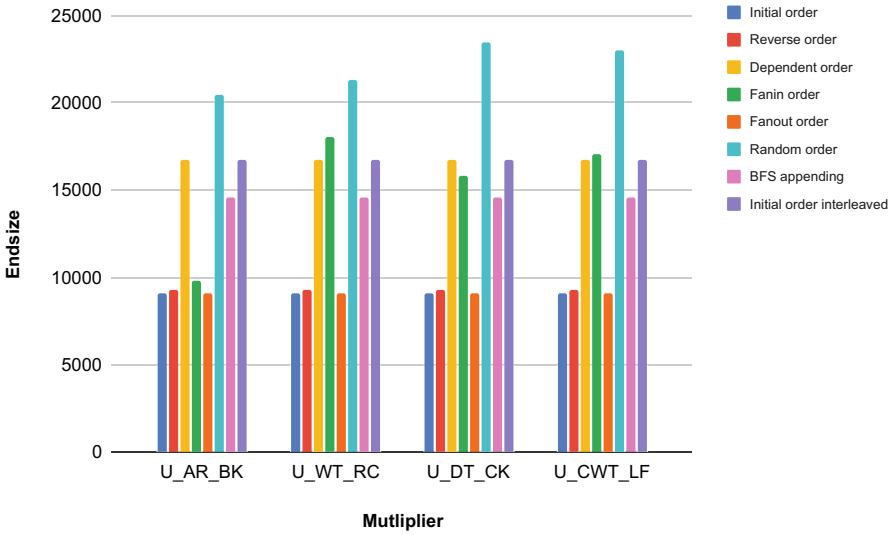


(a)

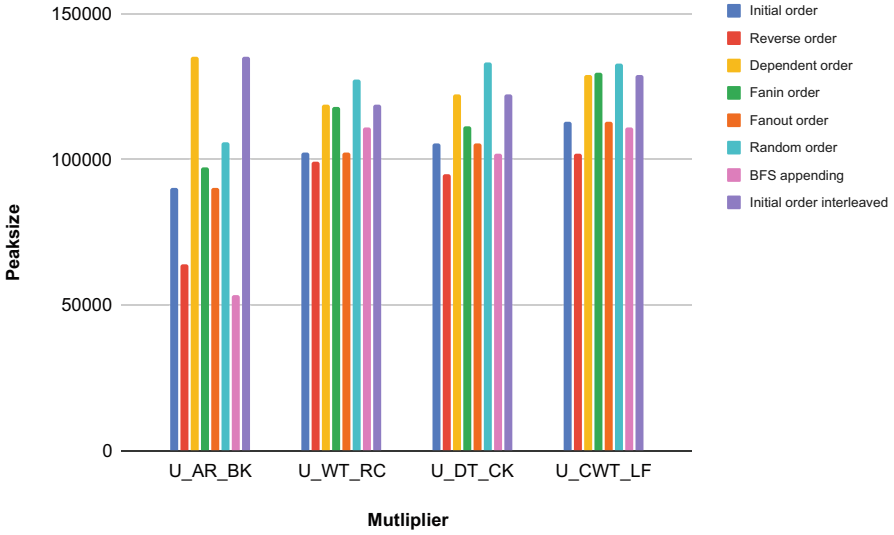


(b)

Fig. 3 Endsize and peaksize of four 8×8 signed multiplier structures for different static variable ordering heuristics. (**a**) Endsize for signed multipliers. (**b**) Peaksize for signed multipliers

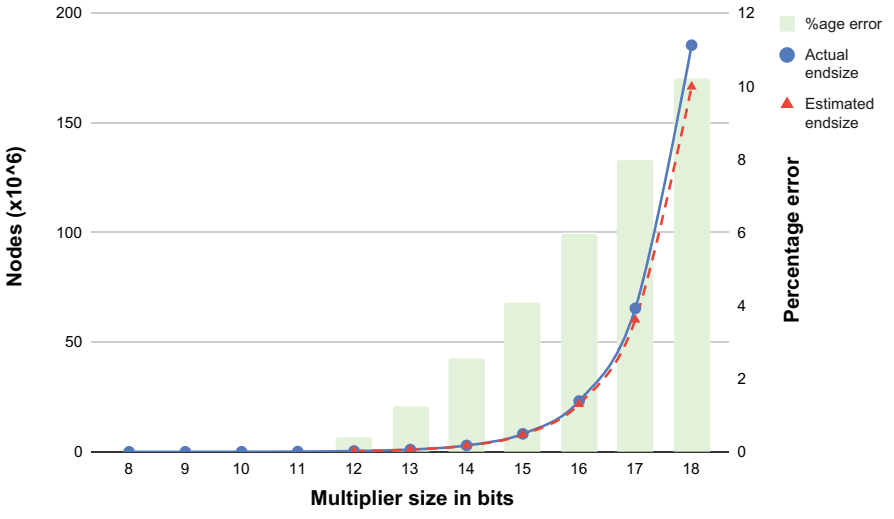


(a)

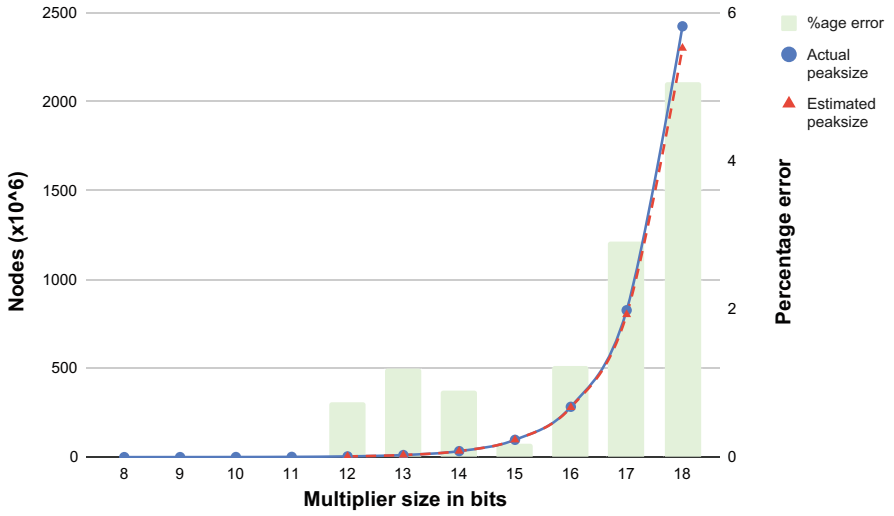


(b)

Fig. 4 Endsize and peaksize of four 8×8 unsigned multiplier structures for different static variable ordering heuristics. (**a**) Endsize for unsigned multipliers. (**b**) Peaksize for unsigned multipliers



(a)



(b)

Fig. 5 Estimated endsize and peaksize for the signed multiplier with Array PPA and Brent-Kung FSA (S_AR_BK). (**a**) Endsize. (**b**) Peaksize

gives the percentage error between the actual and the estimated values. The solid blue lines show the actual endsize and peaksize, and the red dashed lines show the estimated endsize and peaksize for these circuits. The percentage error is shown by the bar graph (green) in the background of the respective graphs.

As visible from Fig. 5a, b, the estimated values follow the same trends as the actual values. Consider the 16×16 bit multiplier in Fig. 5, the estimated endsize is 21, 967, 242 and the estimated peaksize is 279, 660, 485. To calculate the percentage error, we constructed the BDD for the circuit sizes that were achievable. For the 16×16 bit multiplier, the percentage error in endsize is $\approx 5.9\%$ and for peaksize it is only $\approx 1.2\%$. Although the error in the estimated endsize shows an increasing trend in this case, increasing the number of circuits to calculate the growth factors will help in decreasing the error percentage. The error in the estimated peaksize does not show a constant increase as the endsize and is very small in contrast to the endsize. Thus, the values that are calculated for the memory required by the BDD nodes using the estimated peaksize can be reliable.

5.3 Memory Usage Estimation

A single node on CUDD package requires 32-bytes when compiled using 64-bit pointer system (16-bytes for 32-bit pointers) [18]. Table 2 shows the estimated values for endsize, peaksize, and memory requirement of BDD nodes constructed using reverse ordering for a signed multiplier with Array PPA and Brent-Kung FSA (S_AR_BK) for larger sizes. Using these values, for the given structures the minimum memory requirement for a 16×16 multiplier, excluding the auxiliary memory required by the CUDD package, is ≈ 9 GB memory for 64-bit systems. Based on the estimated values, for the 18×18 multiplier, we ran it on system with memory resource less than estimated values (Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40 GHz, main memory = 64 GB). As expected, the BDD construction failed after running for an extended period (runtime $>$ 24 h). Later, we constructed the BDD on a system having resources greater than our memory estimate (Intel(R) Core(TM) i9-11900KF @ 3.50 GHz, main memory = 125 GB), and the construction was successfully completed in a reasonable time ($\approx$ 1 h). Thus, it reinforces the confidence in the estimated values and in the idea that an early memory estimation allows for a more efficient selection of resource and utilization of time.

6 Discussion

In this section, we discuss some observations and possible extensions of our work. Although we applied our methodology to only traditional BDD construction methods, we believe that it can easily be adapted to other methods used for constructing BDDs like the one proposed in [10]. In addition to that, the estimated

Table 2 Estimated endsize, peaksize, and memory requirement for S_AR_BK multiplier using reverse ordering

Multiplier sizes	Estimated endsize	Estimated peaksize	Estimated memory required by CUDD[a]
16 × 16	2.18 × 10^7	2.80 × 10^8	9 GB
18 × 18	1.67 × 10^9	2.30 × 10^9	74 GB
32 × 32	2.54 × 10^14	5.87 × 10^15	1.88 × 10^5 TB
64 × 64	3.23 × 10^28	2.58 × 10^30	8.26 × 19^19 TB

[a] conservative estimate for nodes only

memory requirement is not just useful for resource selection for BDD construction; it can also help in exploring other options for construction of multipliers in case the available resources appear to be insufficient. The effects of approaches which strive to reduce memory usage can also be explored and how these methods effect the growth factors. Another interesting aspect would be the assessment of methodology for arithmetic circuits other than multipliers, and it would be insightful to see how the methodology and estimation extends to these circuits.

7 Conclusion

In this paper, we presented a methodology to choose an optimal static variable ordering heuristic for larger multipliers with early estimation of the endsize, peaksize, and memory requirements for constructing the BDD. Using the smaller version of the target multiplier structure, we were able to find an optimal static variable ordering heuristic that also works equally well for the target multiplier. For the endsize estimation, we reused the chosen heuristics and collected a set of multiplier circuits of the same structure with incremental increase in size to find a growth factor per bit for the endsize and peaksize. This growth factor was used to estimate the endsize and the peaksize of the target multiplier. Using the estimated peaksize, we were also able to project the memory required in constructing the BDD. We demonstrated the applicability of our methodology on various multiplier circuits.

References

1. Hu, A.J.: Formal hardware verification with BDDs: An introduction. In: PACRIM, vol. 2, pp. 677–682 (1997)
2. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Comput. **100**(8), 677–691 (1986)
3. Butler, K.M., Ross, D.E., Kapur, R., Mercer, M.R.: Heuristics to compute variable orderings for efficient manipulation of ordered binary decision diagrams. In: Design Automation Conference, 417–420 (1991)

4. Fujii, H., Ootomo, G., Hori, C.: Interleaving based variable ordering methods for ordered binary decision diagrams. In: International Conference on Computer-Aided Design, pp. 38–41 (1993)
5. Fujita, M., Fujisawa, H., Kawato, N.: Evaluation and improvement of Boolean comparison method based on binary decision diagrams. In: International Conference on Computer-Aided Design, vol. 88, pp. 2–5 (1988)
6. Malik, S., Wang, A., Brayton, R., Sangiovanni-Vincentelli, A.: Logic verification using binary decision diagrams in a logic synthesis environment. In: International Conference on Computer-Aided Design, pp. 6–9 (1988)
7. Drechsler, R.: Evaluation of static variable ordering heuristics for MDD construction. In: International Symposium on Multi-Valued Logic, pp. 254–260 (2002)
8. Drechsler, R., Mahzoon, A.: Polynomial formal verification: Ensuring correctness under resource constraints. In: International Conference on Computer-Aided Design (2022)
9. Mahzoon, A., Große, D., Drechsler, R.: GenMul: Generating architecturally complex multipliers to challenge formal verification tools. In: Recent Findings in Boolean Techniques, pp. 177–191. Springer, Berlin (2021)
10. Burch, J.R.: Using BDDs to verify multipliers. In: DAC, pp. 408–412 (1991)
11. Kumar, J., Srivastava, A., Fujita, M.: Formal analysis of integer multipliers by building binary decision diagram of adder trees. In: ISQED, pp. 58–63. IEEE, New York (2022)
12. ichi Minato, S.: Streaming BDD manipulation. IEEE Trans. Comput. **51**(5), 474–485 (2002)
13. Shiple, T.R., Brayton, R.K., Sangiovanni-vincentelli, A.L.: Computing Boolean expressions with OBDDs (1993)
14. Jain, J., Narayan, A., Sangiovanni-Vincentelli, A., Coelho, C., Brayton, R.K., Khatri, S.P., Fujita, M.: Decomposition techniques for efficient ROBDD construction. In: Formal Methods in Computer-Aided Design, pp. 419–434 (1996)
15. Hett, A., Drechsler, R., Becker, B.: MORE: an alternative implementation of BDD packages by multi-operand synthesis. In: Proceedings EURO-DAC'96. European Design Automation Conference with EURO-VHDL'96 and Exhibition, pp. 164–169 (1996)
16. Beyer, D.: Improvements in bdd-based reachability analysis of timed automata. In: FME, pp. 318–343. Springer, Berlin (2001)
17. Beyer, D., Lewerentz, C., Noack, A.: Rabbit: A tool for bdd-based verification of real-time systems. In: International Conference on Computer Aided Verification, pp. 122–125. Springer, Berlin (2003)
18. Somenzi, F.: CUDD: CU decision diagram package release 2.7.0 (2018). https://github.com/ivmai/cudd
19. Wolf, C., Glaser, J., Kepler, J.: Yosys-a free verilog synthesis suite. In: Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip) (2013)

Index

R. Drechsler, S. Huhn (eds.), *Advanced Boolean Techniques*,
https://doi.org/10.1007/978-3-031-28916-3