FrEDDY: Modular and Efficient Framework to Engineer Decision Diagrams Yourself

Rune Krauss
DFKI
Bremen, Germany
rune.krauss@dfki.de

Jan Zielasko
DFKI
Bremen, Germany
jan.zielasko@dfki.de

Rolf Drechsler
University of Bremen / DFKI
Bremen, Germany
drechsler@uni-bremen.de

***Abstract*—The hardware complexity in electronic devices used by today's society has increased significantly in recent decades due to technological progress. In order to cope with this complexity, data structures and algorithms in electronic design automation must be continuously improved. *Decision Diagrams* (DDs) are an important data structure in the design and analysis of circuits because they allow efficient algorithms for their manipulation. The practical relevance of DDs leads to an ongoing quest for appropriate software solutions that enable working with different DD types. Unfortunately, existing DD software libraries focus either on efficiency or usability. Consequences are a disproportionately high effort for extensions or considerable loss of performance. To tackle these issues, a modular and efficient *Framework* to *Engineer Decision Diagrams Yourself* (FrEDDY) is proposed in this paper. Various experiments demonstrate that no compromise with regard to performance has to be made when using FrEDDY. It is on par with or clearly more efficient than established DD libraries.**

***Index Terms*—Framework, decision diagrams, pseudo-Boolean functions, electronic design automation**

I. Introduction

The growing hardware complexity makes it necessary to continuously improve tools in *Electronic Design Automation* (EDA) in order to meet time-to-market constraints [1]. *Decision Diagrams* (DDs) play a central role in the design, verification, and testing of digital circuits because they can encode corresponding Boolean functions compactly and manipulate them efficiently [2]. Breakthroughs were achieved, inter alia, in EDA applications such as model checking using the well-known reduced ordered *Binary Decision Diagrams* (BDDs) [3].

Motivated by the success of BDDs at bit level, numerous data structures and algorithms have been developed to increase the efficiency of specific EDA applications [4]. For example, there are *Kronecker Functional DDs* (KFDDs) that are useful for XOR-based synthesis. The exponential growth of bit-level DDs, e. g., for integer multiplication, was overcome by pseudo-Boolean functions that can be represented by word-level DDs. While word-level DDs such as *Multiplicative Binary Moment Diagrams* (*BMDs) represent multiplier circuits with linear size, *Algebraic Decision Diagrams* (ADDs) are suitable for matrix multiplication. Furthermore, *Multiplicative Power Hybrid DDs* (*PHDDs) were proposed to also efficiently represent floating point numbers for arithmetic circuit verification.

This work was supported by the German Federal Ministry of Education and Research within the projects DI-OCDCpro (contract no. 16ME0938), FAIRe (contract no. 01IS23074), and ECXL (contract no. 01IW22002).

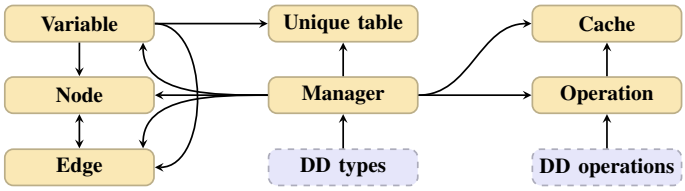


Fig. 1: Dependency graph of modules included in FrEDDY

Even though some of these enhancements can be found in DD software libraries, they generally do not offer the option to implement your own ideas easily and efficiently [5], [6], [7], [8]. The WLD library [8] focuses, e. g., on extensibility but is not performant. The state-of-the-art CUDD library [6], which is most widely used due to its high performance in terms of speedup, originates from the 90s and has not been maintained for several years. Scientific advancements and modern design possibilities are thus not fully exploited, often resulting in codebase changes to tailor custom implementations. Consequences can be program crashes, which is intolerable in EDA.

To address these issues, in this paper we propose an open-source C++23 *Framework* to *Engineer DDs Yourself* (FrEDDY) with the main goal of being modular and efficient. Extendable interfaces are integrated into FrEDDY so that word-level and bit-level DDs can be implemented intuitively. For this purpose, modules are provided to enable high performance. Benchmarks show that arithmetic circuits with word-level DDs implemented using FrEDDY are verified about 15 times faster on average compared to related work. Despite FrEDDY's generalization beyond BDDs, there is no significant performance overhead in symbolic model checking compared to state of the art.

II. Architecture and Design

In this section, we provide an overview of FrEDDY by presenting its highly modular architecture, which is schematically visualized in Fig. 1. In addition to the key feature of extending FrEDDY's software modules with custom DD types and associated operations, design choices based on scientific advances to enable high performance are also described in the following.

Inspired by WLD's concepts, the manager module has a base class that already provides basic functionality such as computing the cofactor $f[0/x]$ [2] of a function f w. r. t. variable x in order to engineer DDs yourself. It abstracts from operators that cannot be generalized for all DD types. They are declared

TABLE I: Feature comparison of DD software solutions

Name	Version	Last release	BDD	KFDD	ADD	*BMD	*PHDD
BuDDy [5]	2.4.0	2004	✓				
CUDD [6]	3.0.0	2015	✓		✓		
Sylvan [7]	1.8.1	2023	✓		✓		
WLD [8]	1.1.0	2003	✓			✓	
FrEDDY	1.0.0	2025	✓	✓	✓	✓	✓

as pure virtual methods and must be implemented by a derived type. For example, there is $(\exists x.f) \equiv f[0/x] \vee f[1/x]$ [3], where the disjunction $\vee$ must be defined in the derived class because it differs at bit and word level.

Another difference exists in the DD encoding. *BMDs, e. g., have numerical edge weights while ADD edges are labeled by Boolean values [9]. To provide maximum flexibility, both the node and edge structure correspond to a template. Specifically, the node entity is a C++ variant, meaning it is either a leaf labeled with any value or a branch consisting of a variable index and successor edges encapsulated by shared pointers, which in turn reference nodes. Even if, depending on instantiation, the node may be slightly larger in terms of memory than a CUDD node, this overhead is negligible. First, a variant does not allocate dynamic memory to guarantee type safety [10]. Second, shared pointers prevent memory leaks by automatically deleting nodes (and edges) that are no longer required.

To reduce DDs during their construction/manipulation and thus ensure canonicity, the unique table is implemented as a flat hash table using the Boost Unordered library [11], which has numerous advantages. For example, all nodes and edges are stored contiguously in memory, which improves cache locality. To avoid hash collisions, multiplicative functions are integrated in orientation on CUDD. Unique tables are used separately for each variable, since it is sufficient to iterate over nodes of the variables to be swapped when reordering variables [2].

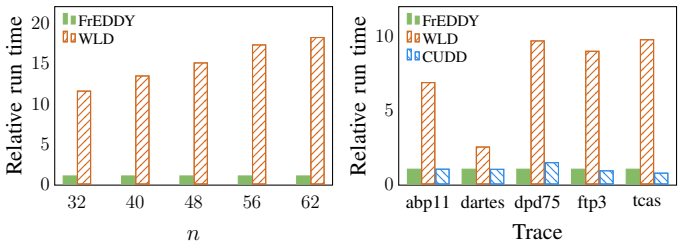
Recursive algorithms such as if-then-else [12] for DD construction are usually called repeatedly, which can be very time-consuming. This is remedied by a computed table that is implemented as the unique table but is designed to cache performed operations. Since there are not only universal operators but also specialized operations like finding satisfying assignments [2], the operation class follows the same polymorphic approach as the base manager. So if the user wants to cache DD operations, they are recognized by the computed table due to inheritance.

Overall, FrEDDY offers rich functionality through the interaction of its modules to implement DDs for efficiently solving EDA tasks. TABLE I gives an overview comparing the features of established DD software libraries and DD types already implemented using FrEDDY. Since FrEDDY is open-source software, more information in terms of usage and development is available at **https://github.com/runekrauss/freddy**.

III. Experiments

In order to evaluate FrEDDY's efficiency, we conducted various experiments that are summarized in this section.

The first experiment was interested in how performant the modules in FrEDDY are in general. Since WLD shares some



(a) Equivalence checking (b) Model checking

Fig. 2: Performance comparison of FrEDDY and related work

concepts, its *BMDs were compared with those of FrEDDY by verifying n-bit multipliers using the hierarchical approach from [9]. The second experiment dealt with FrEDDY's performance at bit level because of its genericity. To this end, FrEDDY was compared with CUDD and WLD by performing BDD-based model checking as in [12]. The experimental setup was fair for all evaluations, i. e., comparable settings were configured for DD software and the same system environment was used. For each benchmark, 100 runs were carried out and mean values calculated in terms of time.

Experimental results are shown in Fig. 2, which are now interpreted. In total, FrEDDY verifies multipliers on average about 15 times faster than WLD due to its handling of shared pointers and the open addressing technique. These main reasons are applicable to the model checking experiment. While FrEDDY can keep up with CUDD despite its additional edge structure, WLD is significantly outperformed.

IV. Conclusion

In this paper, we presented an open-source C++23 framework called *FrEDDY* to engineer DDs yourself, which is modular and efficient. Experiments demonstrated that FrEDDY is on par with or considerably more powerful than established DD software libraries. In summary, these results confirm that the main goal of this work was achieved.

References

[1] R. Reis, "EDA: Overview and some trends," *Journal of Integrated Circuits and Systems*, vol. 17, no. 3, pp. 1–10, 2022.

[2] C. Meinel and T. Theobald, *Algorithms and Data Structures in VLSI Design*. Berlin: Springer, 2012.

[3] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: 10^{20} states and beyond," *Information and Computation*, vol. 98, no. 2, pp. 142–170, 1992.

[4] R. Drechsler and D. Sieling, "Binary decision diagrams in theory and practice," *Journal on STTT*, vol. 3, no. 2, pp. 112–136, 2001.

[5] J. Lind-Nielsen, "BuDDy: A BDD package," 2004. [Online]. Available: https://buddy.sourceforge.net/manual

[6] F. Somenzi, "CUDD: CU Decision Diagram package," 2015. [Online]. Available: https://github.com/ivmai/cudd

[7] T. van Dijk and J. van de Pol, "Sylvan: Multi-core framework for decision diagrams," *Journal on STTT*, vol. 19, no. 6, pp. 675–696, 2017.

[8] M. Herbstritt, "WLD – a C++ library for decision diagrams," 2003. [Online]. Available: https://ira.informatik.uni-freiburg.de/software/wld

[9] R. E. Bryant and Y. Chen, "Verification of arithmetic circuits with binary moment diagrams," in *DAC Proceedings*. IEEE, 1995, pp. 535–541.

[10] M. Gregoire, *Professional C++*. New Jersey: Wiley, 2024.

[11] D. James and P. Dimov, "Boost.Unordered," 2022. [Online]. Available: https://boost.org/libs/unordered

[12] B. Yang *et al.*, "A performance study of BDD-based model checking," in *Proceedings of the Conference on FMCAD*. Springer, 1998, pp. 255–289.