The EPFL Logic Synthesis Libraries

Mathias Soeken Heinz Riener Winston Haaswijk Eleonora Testa Giovanni De Micheli
Integrated Systems Laboratory, EPFL, Lausanne, Switzerland

https://github.com/lsils/lstools-showcase⋆

***Abstract*—We present a collection of open source C++ libraries as well as a set of benchmarks, both of which can be used in the development, testing, and benchmarking of logic synthesis applications. The libraries range from shell interfaces, to exact synthesis and from logic networks to ESOP minimization. All libraries are well documented and well tested. Furthermore, being header-only, the libraries can be readily used as core components in complex logic synthesis systems. We also present the EPFL benchmarks, which are freely available and have already been used to test and benchmark a variety of EDA toolchains.**

I. Introduction

Many problems in logic synthesis are solved by combining a set of common techniques in an efficient way. In this paper, we present a collection of modular open source C++-14/17 libraries that provide efficient implementations of common reappearing logic synthesis tasks. Each library targets one general aspect of design automation. The libraries are well documented and well tested. Being header-only and requiring no strong dependencies such as Boost, the libraries can be easily integrated into existing and new projects. The developer saves time by not having to re-implement common core components, and can focus on tackling more complex logic synthesis problems. A summary of all libraries presented in this paper is collected in a "showcase" repository,[1] which contains links to all library repositories, and several examples in which one or more libraries are used.

In the following, we describe the main features and design decisions of the individual libraries. A more detailed description, which also combines several of the libraries in a running example, can be found at [1].

II. alice: A command shell library

The C++ library *alice* helps to create shell interfaces, which are a common user interface for EDA tools. Users can enter commands that interact with internal data structures. The interface supports standard shell features such as command and file auto-completion as well as a command history. Combining several commands allows users to create synthesis scripts. One can write *alice* programs in a way that separates the core of a library from the shell interface. Inside the default shell interface, only simple scripts in terms of sequences of commands without control flow are supported. However, *alice* shell interfaces can be automatically compiled into Python modules and C libraries. Exposing the shell interface to a Python module offers to use the various modern Python libraries and frameworks for scripting, including data processing (e.g., *pandas*), plotting (e.g., *matplotlib*), and interactive notebooks (e.g., *jupyter*). Export to a C library makes the developer's C or C++ library readily accessible from many other programming languages such as JVM-based languages (e.g., Java), .NET-based languages (e.g., C#), or Tcl, a scripting language that is included in many commercial electronic design automation tools.

In *alice*, different data structures are organized inside stores. Each store can contain several instances of a data structure, and if a store is not empty it points to some *current* store element. A macro API in *alice* makes it easy to register such stores:

```
ALICE_ADD_STORE(Gia_Man_t*, "aig", "a", ...)
ALICE_ADD_STORE(Wlc_Ntk_t*, "wlc", "w", ...)
```

A store is defined by means of the data type of elements it contains. All store related functionality is associated to library code by referring to the store type. No modification of the library code and no wrappers around library types are needed. The second and third argument are command line flags to address specific stores in the command. For example, in order to print statistics about the current AIG one writes '`ps --aig`', or '`ps -a`'; and to print statistics about the current word-level design one writes '`ps --wlc`' or '`ps -w`'. The command '`ps`' is one of several generic store commands, which are by default contained in the shell interface. Macros are used to associate functionality to these commands.

One feature of *alice* is that each command can log data in JSON format into a log file. This eases the retrieval of data and avoids cumbersome program output parsing using regular expressions and string manipulation. Each custom command can control which data should be logged, but also some generic store commands can be configured to log.

Alice code can be compiled into a Python module, by just changing some compile definitions. Each command then corresponds to a Python function, whose arguments are according to the command arguments and whose return value is according to the log data it produces. Compilation to C# and Scala modules is also supported.

III. lorina: A parsing library

The C++ library *lorina* offers parsers for simple formats commonly used in logic synthesis. A parser reads a logic network in a certain format from a file (or input stream) and invokes a callback method of a visitor whenever the parsing of a primitive of the respective format (e.g., an input, an output, or a gate definition) has been completed. These callback methods allow users to customize the behavior of the parser and execute their code interleaved with the parsing. On parse error, a

[1] https://github.com/lsils/lstools-showcase

similar callback mechanism—the *diagnostic visitor*—is used to emit customizable diagnostics.

Each parser is implemented in its own header and provides a *reader function* `read_<format>` and a *reader visitor* `<format>_reader`, where `<format>` has to be substituted by the name of the respective format, e.g., `aiger`, `bench`, `blif`.

The following example shows how to parse a two-level logic network described as a programmable logic array from a file.

```
#include <lorina/pla.hpp>
using namespace lorina;

...

const auto r = read_pla("func.pla", pla_reader());
if (r == return_code::success)
{
  std::cout << "parsing successful" << std::endl;
}
else
{
  std::cout << "parsing failed" << std::endl;
}
```

A user can modify the default behavior of any parser by deriving a new class from a reader visitor and overloading its virtual callback methods. Each method corresponds to an *event point* defined by the implementation of the parsing algorithm, e.g., the completion of the parsing of the format's header information, or a certain input or gate definition.

The listing below shows how to customize the `on_term` event point of the reader visitor `pla_reader` such that after a term is parsed, it is printed. Note that the signatures of the methods in derived classes have to exactly match their counterparts in the base class. The C++ keyword **`override`** causes modern C++ compilers to warn on signature mismatch and permits users to spot these errors quickly.

```
class reader : public pla_reader
{
public:
  void on_term(const std::string& term,
               const std::string& out) const override
  {
    std::cout << term << ' ' << out << std::endl;
  }
}; /* reader */
```

As a third parameter each reader function can optionally take a diagnostic engine. The engine is used to emit diagnostics when the parsing algorithm encounters mistakes. The possible error messages are specified by the implementation of the parsing algorithm.

```
#include <lorina/diagnostics.hpp>

...

diagnostic_engine diag;
read_pla("func.pla", reader(), &diag);
```

Possible diagnostics for the programmable logic array format could look as follows.

```
[e] Unable to parse line
line 1: `i 16`
[e] Unsupported keyword `abc`
in line 4: `.abc`
```

The diagnostic engine supports different levels of diagnostic information and can emit one or multiple diagnostics depending on the severity of the problem. A diagnostic typically consists of a short description of the problem and the line information to ease debugging.

Diagnostics can also be customized by overloading the `emit` method as shown below.

```
class diagnostics : public diagnostic_engine
{
public:
  void emit(diagnostic_level level,
            const std::string& message) const override
  {
    std::cerr << message << std::endl;
  }
}; /* diagnostics */
```

IV. KITTY: A TRUTH TABLE LIBRARY

The C++ library *kitty* provides data structures and algorithms for explicit truth table manipulation. Truth table data structures and algorithms are helpful for Boolean function manipulation, if the functions are small, i.e., if they consist of up to 16 variables. (For some algorithms, also functions with more variables can still be efficiently manipulated.) In such cases explicit truth table representations can be significantly faster compared to symbolic representations such as binary decision diagrams, since truth tables require less overhead to manage than complicated data structures. The following listing is an example of how *kitty* is used to create truth tables that describe the two output functions of a full adder, and to print them in hexadecimal format to the output.

```
#include <kitty/kitty.hpp>
using namespace kitty;

...

dynamic_truth_table a(3), b(3), c(3);

create_nth_var(a, 0);
create_nth_var(b, 1);
create_nth_var(c, 2);

const auto sum = a ^ b ^ c;
const auto carry = ternary_majority(a, b, c);

std::cout << "sum =   " << to_hex(sum) << "\n"
          << "carry = " << to_hex(carry) << "\n";
```

Inside the data structures, a truth table is represented in terms of 64-bit unsigned integers, called *words*. Each bit in a word represents a function value. For example, the truth table for the function $x_0 \wedge x_1$ is `0x8` (which is 1000 in base 2) and the truth table for the majority-of-three function $\langle x_0 x_1 x_2 \rangle$ is `0xe8` (which is 11101000 in base 2). A single word can represent functions with up to 6 variables, since $2^6 = 64$. A truth table for functions with 7 variables requires two words, functions with 8 variables require four words, and so on. In general, an n-variable Boolean function, with $n \geq 6$, can be represented using 2^{n-6} words. On such truth table representations, many operations for function manipulation can be implemented using bitwise operations which map to efficient machine instructions on a processor. For a broader overview on how to implement truth table operations using bitwise operations, we refer the reader to the literature [2], [3].

The two main data structures for truth table manipulation in *kitty* are a static and a dynamic truth table. The choice on which to use depends on whether one knows the number of

variables for the function to represent at compile-time. A static truth table is more efficient at runtime, because it does not need to store its number of variables and for many operations, the number of iterations in a loop are compile-time constants. For both data structures, the number of variables is initialized when constructing an instance and cannot be changed afterwards. This avoids reallocation of memory. If the size of a truth table needs to be changed, a new truth table must be created. The previous example to create the full adder functions uses dynamic truth tables. Changing the data type in Line 6 allows one to use a static instead of a dynamic truth table; no other line has to be changed:

```
6 static_truth_table<3> a, b, c;
```

V. PERCY: AN EXACT SYNTHESIS LIBRARY

The *percy* library provides a collection of SAT based exact synthesis engines. These include engines based on conventional methods, as well as state-of-the-art engines which can take advantage of DAG topology information and perform parallel exact synthesis [3], [4]. The constraints and algorithms of such synthesis engines may be quite dissimilar. Moreover, it is not always obvious which combination will be superior in a specific domain. It is often desirable to experiment with several methodologies and solving backends to find the right fit. The aim of *percy* is to provide a flexible common interface that makes it easy to construct a parameterizable synthesis engine suitable for different domains.

The *percy* library also serves as an example of the ideas presented in this paper. It is built on top of *kitty*, which it uses to construct synthesis specifications. Thus, it shows how the lightweight libraries proposed here can be easily composed to build up ever more complex structures.

Synthesis using *percy* concerns four main components:

1) *Specifications* – Specification objects contain the information essential to the synthesis process such as the functions to synthesize, I/O information, and a number of optional parameters such as conflict limits for time-bound synthesis, or topology information.
2) *Encoders* – Encoders are objects which convert specifications to CNF formulæ. There are various ways to create such encodings, and by separating their implementations it becomes simple to use encodings in different settings.
3) *Solvers* – Once an encoding has been created, we use a SAT solver to find a solution. Currently supported are ABC's `bsat` solver, the Glucose and Glucose-Syrup solvers, and the CryptoMinisat solver [5], [6], [7]. Adding a new SAT solver to *percy* is as simple as declaring a handful of interface functions.
4) *Chains* – Boolean chains are the result of exact synthesis. A Boolean chain is a compact multi-level logic representation that can be used to represent multi-output Boolean functions.

A typical workflow will have some source for generating specifications. These are then synthesized, during which optimum Boolean chains are produced. Internally, the synthesis will compose some encoder and SAT solver in a specific synthesis flow. For example, a resynthesis algorithm might generate cuts in a logic network which serve as specifications. They are then fed to a synthesis flow, and if the resulting optimum Boolean chains leads to an improvement, are replaced in the logic network. In optimizing this workflow, *percy* makes it easy to swap out one synthesis flow for another, to change CNF encodings, or to switch to a different SAT solver.

VI. MOCKTURTLE: A LOGIC NETWORK LIBRARY

The C++ library *mockturtle* provides generic logic synthesis algorithms and logic network data structures. The main philosophy of *mockturtle* is that all algorithms are generic in the sense that they are independent from the implementation of the logic network data structure. In order to achieve this, *mockturtle* makes use of concept-based design using some modern C++-17 language features. The library design is based on a principle of 4 layers that depend on each other in a linear order, with the most fundamental layer being the *network interface API*, followed by *algorithms*, followed by *network implementations*, and finally a layer for *performance tweaks* on the top. Note how this order of layers reflects the algorithm's independence of the network data structures.

Key is to not use dynamic polymorphism, as this would (i) harm performance and (ii) add an unnecessary dependency on the implementation of network data structures, e.g., by extending an abstract base class. The base of the framework is provided by the *network interface API*. It defines naming conventions for types and methods for types and methods in classes that implement network interfaces, of which most are optional. The network interface API does not provide any implementations for a network though. Instead, it helps to agree on names; and if developers follow these names in their own implementation of logic network data structures, then these can be used together with the logic synthesis algorithms developed in *mockturtle*.

For example, the network interface API suggest names like `size`, `is_pi`, and `is_constant` to implement functionality that returns the number of nodes in a network, and check whether a node is a primary input or a constant, respectively. The algorithm `cut_enumeration`, which can enumerate priority k-cuts [8] in a network, requires these functions together with `get_node`, `node_to_index`, `foreach_node`, and `foreach_fanin`. Now this algorithm can be called with any network implementation as long as it implements these functions—independent of the underlying gate library and implementation details. The following code shows how to call the algorithm and then print all cuts for all nodes on some network `ntk`.

```
1 auto cuts = cut_enumeration(ntk);
2
3 ntk.foreach_node([&](auto node) {
4   std::cout << cuts.cuts(ntk.node_to_index(node)) << "\n";
5 });
```

The third layer consists of actual network implementations for some network types that implement the network interface API, e.g., And-inverter graphs, Majority-inverter graphs, XOR-majority graphs, or k-LUT networks. Static compile-time assertions in the algorithms are guaranteeing that compi-

lation succeeds only for those network implementations that do provide all required types and methods. We make use of static inheritance in so called *views* to extend or modify a network implementation's functionality. For example, if `Network` is a network type, then `topo_view<Network>` is also a network type that guarantees that nodes are visited in topological order. Other views exist in the library and they can be composed arbitrarily. Since the composition is based on static inheritance, it does not add any runtime overhead.

Finally, to guarantee a fast implementation of algorithms, we use static conditional checks on the network type to execute specific implementations based on the network data structure's properties. For example, cut enumeration requires to enumerate over all elements in all sets of the children of a node. If the network data structure is heterogeneous and has a different number of fanins for internal nodes, then this loop must be computed dynamically. However, if the network data structure is homogeneous (e.g., in the case of an And-inverter graph, each node has two children), then two nested for-loops would simply suffice. The library *mockturtle* implements such performance tweaks in various places without losing the generic interface.

VII. EASY: AN ESOP LIBRARY

The C++ library *easy* provides implementations of verification and synthesis algorithms for exclusive-or sum-of-product (ESOP) forms. An ESOP form is a two-level logic representation that consists of one level of multi-fanin AND-gates, followed by one level of multi-fanin XOR-gates. For instance, the Boolean expression

$$x_1\overline{x}_2\overline{x}_3\overline{x}_4 \oplus x_0\overline{x}_3x_4 \oplus x_0x_2\overline{x}_3 \oplus x_0x_1x_2x_3 \oplus 1$$

is an ESOP form that realizes the Boolean function `0xcafeaffe`.

ESOP forms have good testability properties and allow for a compact representation of arithmetic circuits. The inherent reversibility of the XOR-gate, moreover, sparked interest in ESOP forms in application fields like cryptography and quantum computation.

The *easy* library implements truth table based algorithms using *kitty* to verify that an ESOP form realizes a completely- or incompletely-specified Boolean function or to verify that two ESOP forms are functionally equivalent. Being truth table based, these algorithms are particularly effective when Boolean functions with 16 or less Boolean variables are represented.

The ESOP representation is not canonical. One Boolean function can be expressed by multiple structurally different, but semantically equivalent ESOP forms. For many applications, ESOP forms with small (or minimum) costs with respect to a cost criterion, e.g., the number of product terms or the number of AND-gates, are of interest. The *easy* library provides various heuristic and exact methods for synthesizing ESOP forms from a given Boolean function:

1) *Decomposition based ESOP synthesis methods* recursively decompose the Boolean function and re-compose its ESOP form from the individual parts. These methods are fast, but in general do not lead to an ESOP form of minimum size. Within *easy*, decomposition algorithms using *Pseudo-Kronecker Reed-Muller* (PKRMs) and *Positive Polarity Read-Muller* (PPRMs) forms are implemented, which are both special cases of ESOP forms.
2) *SAT based ESOP synthesis methods* formulate the problem of synthesizing an ESOP form with a fixed number of k product terms as a constraint satisfaction problems. A satisfying assignment for the constraint system directly corresponds to a realization of an ESOP from with k product terms. The *easy* library supports two formulations of SAT based ESOP synthesis—one based on the Helliwell equation, the other based on Boolean learning [9]. The latter formulation uses a counterexamples-guided abstraction-refinement loop to cope with many Boolean variables and large don't care sets. In an iterative search procedure, these synthesis algorithms allow a user to compute an ESOP form of minimum size.

VIII. THE EPFL BENCHMARKS

The *EPFL Combinational Benchmark Suite* [10] was introduced in 2015 with the aim of defining a new comparative standard for the logic optimization and synthesis community. It originally consisted of 23 combinational circuits designed to challenge modern logic optimization tools. The benchmark suite is divided into arithmetic, random/control, and MtM (more-than-a-million gates) circuits, and each circuit is distributed in Verilog, VHDL, BLIF and AIGER formats. Along with the benchmarks there is an ongoing competition to compare logic optimization algorithms in terms of their ability to optimize 6-LUT mappings for size and depth.

IX. ACKNOWLEDGMENTS

This research was supported by the Swiss National Science Foundation (200021-169084 MAJesty).

REFERENCES

[1] M. Soeken, H. Riener, W. Haaswijk, and G. De Micheli, "The EPFL Logic Synthesis Libraries," *arXiv preprint arXiv:1805.05121*, 2018.
[2] H. S. Warren, Jr., *Hacker's Delight*. Addison-Wesley, 2002.
[3] D. E. Knuth, *The Art of Computer Programming, Volume 4A*. Addison-Wesley, 2011.
[4] W. Haaswijk, A. Mishchenko, M. Soeken, and G. De Micheli, "SAT based exact synthesis using DAG topology families," to appear in Design Automation Conference, 2018.
[5] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Int'l Conf. on Theory and Applications of Satisfiability Testing*, 2003, pp. 502–518.
[6] G. Audemard and L. Simon, "Glucose and Syrup in the SAT Race 2015," in *Reports on the SAT 2015 Competition*, 2015.
[7] M. Soos, K. Nohl, and C. Castelluccia, "Extending SAT solvers to cryptographic problems," in *Int'l Conf. on Theory and Applications of Satisfiability Testing*, 2009, pp. 244–257.
[8] A. Mishchenko, S. Cho, S. Chatterjee, and R. K. Brayton, "Combinational and sequential mapping with priority cuts," in *Int'l Conf. on Computer-Aided Design*, 2007, pp. 354–361.
[9] H. Riener, R. Ehlers, B. Schmitt, and G. De Micheli, "Exact synthesis of ESOP forms," *arXiv preprint arXiv:1807.11103*, 2018.
[10] L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "The EPFL combinational benchmark suite," in *Int'l Workshop on Logic and Synthesis*, 2015.