ScaLP: A Light-Weighted (MI)LP Library

Patrick Sittel
University of Kassel
sittel@uni-kassel.de

Thomas Schönwälder
University of Kassel
uk000595@student.uni-kassel.de

Martin Kumm
University of Kassel
kumm@uni-kassel.de

Peter Zipf
University of Kassel
zipf@uni-kassel.de

Abstract. Many design flows involve the process of automatically solving complex optimization problems using linear relations. Especially, modern circuits and systems are often implemented using linear programming (LP). While code integration and portability are critical topics, it is not obvious to the designer which of the numerous LP solvers is the most ideal regarding runtime performance, problem modeling and licensing. Additionally, current state-of-the-art integration tools include many dependencies, long compile times, are complicated to handle or provide interfaces that are susceptible for design errors. To provide a solution for these disadvantages, we present the Scalable LP library `ScaLP` - an open-source `C++` library that uses a unified interface to generically model LP problems and enable runtime dynamic exchange of solvers. Also, it reduces code complexity by providing denser ILP problem formulations.

1. Introduction

The method of mixed integer linear programming (MILP) is a general approach for solving mathematical optimization problems which can be modeled using linear relationships. There are plenty of fields where MILP finds application, such as agriculture, telecommunication or operations research. The real-world goals in these fields may be different, but as long as the underlying systems can be represented as an LP model, a mathematical statement and objective can be constructed. In the early days of LP, shortly after George Dantzig published the simplex method in 1947 [5], the method of transforming the complex mathematical descriptions into the solver specific input format using low-level code was, in reality, holding back the efficiency of the development process. As a solution, the first algebraic modeling languages (AMLs) were published in the late 1970s. It provided system developers with the ability to formulate LP problems in a more natural and generic way [23]. This led to ALM representations that are readable by both humans and computers [8]. After a problem is formulated, the solutions are generated using sophisticated *solvers*. With these advantages, problems could be modeled easily and solved automatically, which allowed designers to focus on high-level design problems.

Listing 1: Dynamic ILP solving with ScaLP

```
#include <iostream>
#include <ScaLP/Solver.h>
#include <ScaLP/Exception.h>

int main()
{
  try
  {
    ScaLP::Solver s{"CPLEX","Gurobi","SCIP"};
    s.quiet=true;

    ScaLP::Variable x1 = ScaLP::newBinaryVariable("x1");
    ScaLP::Variable x2 = ScaLP::newIntegerVariable("x2");
    ScaLP::Variable x3 = ScaLP::newIntegerVariable("x3",0,ScaLP::INF());
    ScaLP::Variable x4 = ScaLP::newRealVariable("x4");
    ScaLP::Variable x5 = ScaLP::newRealVariable("x5",12.5,88);

    s.setObjective(ScaLP::maximize(2*(x1+x3+x4)+4*x2-x5));

    s << (5*x1+4*x2+3*x3+7*x4+  x5<=135);
    s << (  x1+7*x2+5*x3+4*x4+2*x5<=135);
    s << (8*x1+9*x2+2*x3+  x4+3*x5<=135);

    ScaLP::status r = s.solve();

    if(r==ScaLP::status::OPTIMAL)
    {
      ScaLP::Result res = s.getResult();
      std::cout << res << std::endl;
    }
    else
      std::cout << "No optimal solution" << std::endl;
  }
  catch(ScaLP::Exception& e) { std::cout << e << std::endl; }
  return 0;
}
```

Nevertheless, there still existed several major issues that needed to be resolved: portability, integration and extendibility. To fix this, the modeling process has been integrated with object-oriented programming languages [12] as embedded domain specific language (eDSL). When Nielsen published his C++ library for mathematical optimzation models [21] in the early 90s, the benefits of using object oriented languages in the context of LP was widely recognized. Using such languages for modeling and solving LP problems has many privileges. It allows designers to describe their mathematical models in an object-oriented language which is easy to be integrated into a large number of software projects.

However, commonly used interfaces are solver specific. As a result, the formulation and solving process often are not separated from each other. We have also observed that other tools which are often used for LP, e.g., `OR-Tools` [10] and `JuMP` [7], come with other dependencies and packages that consume a lot of disk space and compile time. As a solution to those disadvantages, we present `ScaLP`, a light-weighted and solver flexible mixed-integer linear programming (MI)LP library for C++ applications. `ScaLP` was utilized with success in several projects ranging from the modulo scheduler [22] used in a high level synthesis (HLS) tool Origami HLS [15] to arithmetic optimizations like the (reconfigurable) constant multiplication [20] or compressor tree optimization [17] used in the floating point core generator (FloPoCo) project [6].

1.1. Example Program

The following ScaLP example program that is shown in Listing 1 illustrates its solver flexibility, unified interface and natural translation of the mathematical LP formulation into C++ code. The output of this example code is shown in Listing 2.

The dynamic solver interface of the ScaLP library is shown in line 9 of Listing 1, where a solver object s is generated. The code in line 9 of the example program shows that either CPLEX, Gurobi or SCIP can be used for solving. The used solver depends on what is installed on the target system. Since the code does not depend on one specific solver, the example program is portable, flexible and robust. If more than one solver is available, the first available solver in the constructor of the ScaLP::Solver class will be chosen. We call this feature of ScaLP a *wishlist*. A detailed description is given in Section 2.4.

An example LP problem is formulated in the following lines. The variables are declared in lines 12–16. In this example, there is one binary variable and, limited as well free, integer and real typed variables. While x2 and x4 are unlimited, the constraints $x3 \geq 0$ and $12.5 \leq x5 \leq 88$ have to hold. An objective is formulated and provided to the solver object in line 18. The constraints are passed to s in the lines 20–22. Finally, the problem is solved and the information whether the solution found was feasible or optimal is stored in r and evaluated in the lines 26–32. The results are stored in a ScaLP::Result object res in line 28.

In this case, an optimal solution will be identified and it is printed to the console in line 29. The output, which reports all relevant values and solving time statistics, is shown in Listing 2. After solving, all of the determined values and additional statistics are stored in the Result object and can be accessed using C++ API.

Listing 2: ScaLP output of the example program

```
1   Objective: 47.0714
2  Variables:
3    x1          =      1
4    x2          =      8
5    x3          =      1
6    x4          =      11.7857
7    x5          =      12.5
8  Durations:
9    preparation:   2.7e-05
10   construction:  6.1e-05
11   solving:       0.012401
```

1.2. Related Work

Traditionally, the problem of generating and solving LP problems is driven by software development. Using the benefits of the AML representation, several different tools and frameworks were developed. They can be divided into three categories: general and solver specific AML systems and object-oriented modeling languages (OOMLs).

Specific AML systems apply the model-data-solver independence model, supporting the use of different solvers and data models. Information about popular representatives of this idea is shown in Table 1. Solver specific AML systems are closed approaches. They are developed to support exclusively one solver. Therefore, they provide a more complete and specific support. Most AMLs are distributed commercially. And in general, all design concepts of the displayed AML systems are based on the same fundamental ideas, but differ in data models and interfaces.

Table 1: Widely used AMLs. The top half shows stand alone tools while the bottom half shows language extensions or libraries

ALM	First Published	Concept	Solver	Availability
AMPL [9]	1987	AMS	generic	commercial
GAMS [4]	1992	AMS	generic	commercial
ZIMPL [14]	2004	C command line tool	generic	open source
OPL	1987	AMS	CPLEX	commercial
LINGO	1989	Excel plugin	LINDO	commercial
FICO XPress	2002	AMS	XPress Solver	commercial
YALMIP [18]	2004	Matlab Toolbox	generic	open source
COIN-OR Osi [1]	2004	C++ bib	generic	open source
FlopC++ [12]	2007	C++ bib	generic	open source
Pyomo [11]	2011	Python package	generic	open source
OR-Tools	2012	C++/C#, Java, Python bib	generic	open source
JuMP [7]	2017	Julia package	generic	open source

Another popular approach is to implement algebraic models for LP from OOMLs. Most of the time, objects are used for the presentation of common expressions such as constraints, objectives and variables. In general, relevant operators and functions are overloaded to model LP formulations intuitively. While `YALMIP`, `FlopC++` and `Pyomo` use the commonly used approach of operator overloading, the Julia package `JuMP` adopts the syntactic macro concept that is provided by the `Julia` language. The use of OOMLs is beneficial in many cases, because they are able to take advantage of a much more comprehensive programming environment provided by standard libraries.

Today, the commercial AML systems, like AMPL and GAMS, are widely used and can be considered as state-of-the-art in academia and industry. However, each system has established its own syntax which is entirely separated from any other programming language. As a result, it is difficult to embed a commercial AML system into a complex software project. Open source alternatives either have other dependencies like Matlab and COIN-OR [19] or have many dependencies like the google `OR-tools` [2]. As a result, light-weighted and robust open source alternatives without dependencies are needed. With `ScaLP` which only uses the `C++` standard library functions, we reduce dependency complexity.

1.3. Contribution

We present the light-weighted open source (MI)LP library `ScaLP` [16] that is a `C++11` compatible library which offers a unified user friendly interface for MILP solvers. This interface can be seen as a `C++`-wrapper for LP. `ScaLP` provides the following features, which (altogether) are unique to `C++`-libraries

- a natural interface that heavily uses the concept of operator overloading,
- a generic solver backend that enables the solver selection at runtime,
- solver support from open-source to commercial high-end solver.

To the best of our knowledge, dynamic solver flexibility combined with a unified and easy to manage constraint interface was not addressed so far. All supported solvers, currently CPLEX,

Gurobi, SCIP [24] and LPSolve [3], can be switched dynamically once ScaLP was built and linked to the project, because of ScaLP's unified C++ interface for generating MILP formulations. By detaching the ILP formulation modeling from the process of solving, the interface is intuitive and robust to use. Additionally, all constraints can be formulated using less C++ code than state-of-the-art MILP interfaces.

2. ScaLP Design Paradigms

The presented ScaLP library enables the generation and solving of LP problems using dynamically interchangeable solvers. ScaLP is written in C++, which makes it easy to integrate in one of the numerous C++ based toolflows for system design. Another main feature of ScaLP is the independency from any tool or library other than the C++-standard-libraries, `libdl` for linking and the used solvers, which can be switched dynamically. One benefit of this concept is that a designer does not have to worry about user-compile-time dependencies. Nevertheless, problems are solved using an arbitrary solver, which can be switched dynamically without recompilation of ScaLP.

2.1. Problem Formulation

The formulation of LP problems is based on objectives and constraints that describes relations of variables. In our approach, these objectives and constraints are formulated using terms that are formulated with operator overloading and declared variables. The relevant classes for this construction process are shown in Figure 1. Variables are used to generate objects of the term class using +, - or * operators. These terms can be used to describe relations and generate constraints or to generate an objective. As already shown in the motivational example, we tried to keep the LP formulation as intuitive as possible. As a result, LP problem formulation with ScaLP is straight-forward, accessible and easy to debug.

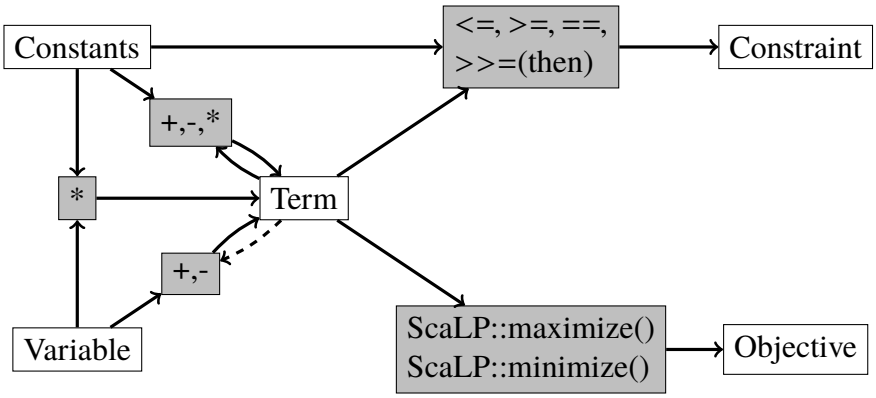

Figure 1: Construction process

2.2. Interface

ScaLP offers an unified interface for various (MI)LP solvers that is designed to detach the problem formulation from the solving process. In that way, variables, constraints and the objective can be formulated without setting any solver specifics. This allows designers to focus on high-level

Table 2: Comparison of the definition of the example objective max $10x_0 + 20x_1 + 30x_2$ in the frameworks Coin-OR Osi OR-Tools and ScaLP

Coin-OR Osi [1]	OR-Tools [10]	ScaLP

`double *o =`
`  new double[3];`
`  o[0] = 10.0;`
`  o[1] = 20.0;`
`  o[2] = 30.0;`
`  solver->setObjSense(max);`
	`MPObjective* const o =`
	`  solver->MutableObjective();`
	`o->SetMaximization();`
	`o->SetCoefficient(x0,10);`
	`o->SetCoefficient(x1,20);`
	`o->SetCoefficient(x2,30);`
		`ScaLP::Objective o =`
		`ScaLP::maximize(10*x0+20*x1+30*x2);`

Table 3: Comparison of the definition of the example constraint $10x_0 + 4x_1 + 5x_2 <= 600$ in the frameworks Coin-OR Osi, OR-Tools and ScaLP

Coin-OR Osi [1]	OR-Tools [10]	ScaLP

`CoinPackedVector row1;`
`row1.insert(0, 10.0);`
`row1.insert(1, 4.0);`
`row1.insert(2, 5.0);`
`row_lb[0] = -1.0`
`* si->getInfinity();`
`row_ub[0] = 600.0;`
	`MPConstraint* const c =`
	`  solver.MakeRowConstraint`
	`  (-infinity, 600.0);`
	`c->SetCoefficient(x0, 10);`
	`c->SetCoefficient(x1, 4);`
	`c->SetCoefficient(x2, 5);`
		`ScaLP::Constraint c =`
		`  10*x0+4*x1+5*x2 <= 600;`

problems while generating their code. As a side effect, this concept forces designers to use more generic and immutable data structures, which leads to less mutation bugs. This is very useful for the integration of LP solving into larger projects, because a well formulated LP problem is independent from the solver backend.

The general idea is to design an interface that is as simple as possible and still allows proper LP problem generation and solving. For example, `ScaLP` has only one type for constraints, while `OR-Tools` distinguish between range constraint, indicator constraints, quadratic constraints. By automatically detecting the proper constraint type, `ScaLP` removes this unnecessary burden from the designer's hands. When necessary, specific features, like indicator constraints, can be activated using the `ScaLP::Feature` class. This is described in Section 2.4.

We use examples to compare the way LP formulations are defined in `ScaLP` and the two state-of-the-art `C++` libraries `Coin-OR Osi` [1] and `OR-Tools` [10]. Table 2 and Table 3 show code examples how the objective

$$\max 10x_0 + 20x_1 + 30x_2 \tag{1}$$

and the constraint

$$10x_0 + 4x_1 + 5x_2 <= 600 \tag{2}$$

are defined in the different frameworks.

While several containers and ranges have to be defined when the `Coin-OR Osi` or the `OR-Tools` framework is used, the definition of constraints using `ScaLP` is as simple as it should. Taking advantage of `C++` operator overloading, `ScaLP` is able to reduce code usage to a one line statement which is as readable as the mathematical statement in (2).

2.3. Solver Integration

Dynamic solver integration is a central feature of the `ScaLP` library. The supported solvers are integrated by wrappers that are managed as plugins, which are based on shared libraries. This allows `ScaLP` to load any supported solver plugin during runtime. As a result, it is possible to compile the main library and the application once and load or add any solver as needed.

Figure 2 shows the dynamic solver integration of the `ScaLP` library. All plugins implement the *SolverBackend* class, which is used as a unified interface to the solvers. Every plugin needs to export a specific C-Symbol, based on the solver name (e. g. "newSolverCPLEX"), that initializes an instance of the *SolverBackend* class on the heap. In that way it is possible to instantiate the class using the omnipresent loading mechanism for C-libraries. When there is more than one solver given during the initialization of an instance of the main *solver* class, `ScaLP` tries to determine a suitable solver-plugin. We call this a whishlist of solvers and describe this feature in Section 2.4.

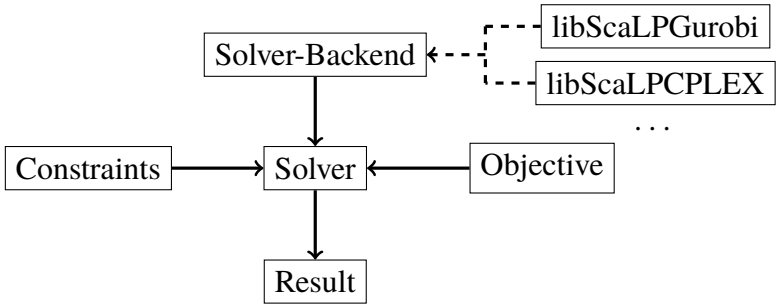

Figure 2: Solving process using ScaLP

2.4. Solver Wishlist and Feature Selection

As described in the motivational example, it is possible to pass a list of solvers to the `ScaLP::Solver` class that specifies the preferred solvers in a user specified descending order. The solvers mentioned are prepended to the list passed to `ScaLP::Solver` constructor. In that way, a designer can specify some default solvers and switch or add solvers without recompiling using the environment variable. The first found solver is used.

The general concept of `ScaLP` is to keep the LP problem generation and solving process as simple and general as possible. Still, in some cases an experienced designer might want to take advantage of solver specific features, e.g., indicator constraints that are often used to avoid so called `big-M` constraints [13]. Often times these features are not available for all supported solvers. In these cases, it is useful to branch the generating `C++` code to act accordingly. For this reason `ScaLP` offers a possibility to check whether a feature is supported or not.

Listing 3 shows an example of branched generation code. Using the wishlist concept, a solver object `s` is generated in line 1. While the `CPLEX` solver allows the use of indicator constraints, this feature is not supported by `LPSolve`. In line 3, it is checked whether the used solver of the object `s` supports the indicator constraints feature. Using this coding style, advanced solver specific features can be used without breaking the concept of generating portable code for LP.

Table 4: Code density comparision

Problem	Coin-OR Osi		OR-Tools		ScaLP		Char Reduction	
	lines	chars	lines	chars	lines	chars	to Osi[%]	to OR[%]
fixedCharge[1]	93	1965	66	1609	62	1202	38.8	25.3
projectSelection[1]	52	1194	54	1771	28	775	35.1	56.2
woodysProblem[2]	46	1066	36	1159	25	635	40.4	45.2
ILP example[3]	46	1059	36	1143	25	597	43.6	47.8
total	237	5284	192	5682	140	3209	**39.3**	**43.5**

[1] http://www.ifp.illinois.edu/~angelia/ge330fall09_ilpmodel_l22.pdf
[2] http://www.unc.edu/depts/stat-or/courses/provan/STOR614_web/lect11_IP.pdf
[3] https://en.wikipedia.org/wiki/Integer_programming#Example

Listing 3: Branching between indicator constraints and bigM

```
1  ScaLP::Solver s{"LPSolve","CPLEX"};
2
3  if (s.featureSupported(ScaLP::Feature::INDICATOR_CONSTRAINTS))
4  {
5    // create indicator constraints
6  }
7  else
8  {
9    // use bigM
10 }
```

Additionally, it is possible to specify a list of wanted features and pass them to the `ScaLP::Solver` object. The first solver that supports all stated features is chosen from the wishlist. In that way, the solver wishlist is filtered to guarantee a specific behavior of the solver if a feasible one can be found. Listing 4 shows an example for passing a feature and a wishlist to the `ScaLP::Solver` object. Using the `ScaLP::Solver` constructor, the indicator constraint feature is requested and the solver wishlist, consisting of `LPSolve` and `CPLEX`, is provided. Although it is only stated as second option, `CPLEX` will be chosen as solver, because `LPSolve` does not support indicator constraints.

Listing 4: Select the first solver using indicator constraints

```
1  ScaLP::Solver s({ScaLP::Feature::INDICATOR_CONSTRAINTS},{"LPSolve","CPLEX"});
```

3. Code Density

In this Section, we examine the code density, which is a significant metric for the simplicity and effectiveness of the interface. We compare our approach against `Osi` and `OR-Tools`, which are two state-of-the-art C++-based interfaces. For comparison, we used four different LP problems and implemented them using the before mentioned interfaces.

Table 4 shows the results of the code density comparison. For each interface and problem, the lines and characters used are displayed. Number of lines used depends on the coding style and is an ambiguous metric. But, number of characters can be seen as an accurate measurement for code density. One can see from the results that the proposed `ScaLP` library performs better in all cases.

Considering all problems and interfaces, the character reduction using ScaLP varies from 35.1% up to 56.2%. In total, a significant character reduction of 39.3% compared to Coin-OR Osi and of 43.5% compared to OR-Tools could be observed.

4. Conclusion and Future Work

We presented the open-source C++ library ScaLP for (MI)LP problem formulation. Users are able to access tutorials, support and the latest version of ScaLP via the project website [16]. Using ScaLP, designers are able to generate robust and portable C++ code. Our library provides the ability to switch between installed solvers without recompilation during runtime. The right solver is selected from a wish list according to the problem-specific features. The interface that is used by ScaLP enables intuitive and easy to debug code for solving (MI)LP problems. Using this interface, it could be shown exemplary that the code density is reduced by 39.9% and 43.5% when compared to Coin-OR Osi and OR-Tools, respectively.

In the future, we want to support non-linear problems and unconventional constraints, e.g. special ordered set (SoS) constraints. Additionally, we want to support additional solvers in the backend. In some cases we observed that certain problems had to be solved multiple times. To address this, we want to implement a solution cache for known problems to avoid unnecessary LP solving.

References

[1] *COIN OR Open Solver Interface home page*, 2017. https://projects.coin-or.org/Osi.

[2] *Google Optimization Tools*, April 2017. https://developers.google.com/optimization/.

[3] Berkelaar, Michel, Kjell Eikland, and Peter Notebaert: *lpsolve: Open Source (Mixed-Integer) Linear Programming System*. Eindhoven U. of Technology, 2004.

[4] Brooke, A, D Kendrick, and A Meeraus: *GAMS: A Users Guide, Release 2.25. 1992*.

[5] Dantzig, George: *Linear Programming and Extensions*. Princeton University Press, 2016.

[6] Dinechin, Florent de and Bogdan Pasca: *Designing Custom Arithmetic Data Paths with FloPoCo*. IEEE Design & Test of Computers, 28(4):18–27, 2012.

[7] Dunning, Iain, Joey Huchette, and Miles Lubin: *JuMP: A Modeling Language for Mathematical Optimization*. SIAM Review, 59(2):295–320, 2017.

[8] Fourer, Robert: *Modeling Languages Versus Matrix Generators for Linear Programming*. ACM Transactions on Mathematical Software (TOMS), 9(2):143–183, 1983.

[9] Fourer, Robert, David M Gay, and Brian W Kernighan: *AMPL: A Mathematical Programming Language*. Citeseer, 1987.

[10] Google Inc.: *Google Optimization Tools*. https://developers.google.com/optimization/.

[11] Hart, William E, Jean Paul Watson, and David L Woodruff: *Pyomo: Modeling and Solving Mathematical Programs in Python*. Mathematical Programming Computation, 3(3):219–260, 2011.

[12] Hultberg, Tim: *flopC++ An Algebraic Modeling Language Embedded in C++*. Operations Research Proceedings 2006, pages 187–190, 2007.

[13] Klotz, Ed and Alexandra M Newman: *Practical Guidelines for Solving Difficult Mixed Integer Linear Programs*. Surveys in Operations Research and Management Science, 18(1-2):18–32, October 2013.

[14] Koch, Thorsten: *Rapid Mathematical Programming*. PhD thesis, Technische Universität Berlin, 2004.

[15] Kumm, Martin, Patrick Sittel, and Konrad Möller: *Origami HLS - Project Website*, 2017. http://www.uni-kassel.de/go/origami.

[16] Kumm, Martin, Patrick Sittel, and Thomas Schöwälder: *ScaLP - Project Website*, 2018. http://www.uni-kassel.de/go/scalp.

[17] Kumm, Martin and Peter Zipf: *Pipelined Compressor Tree Optimization Using Integer Linear Programming*. In *IEEE International Conference on Field Programmable Logic and Application (FPL)*, pages 1–8. IEEE, 2014.

[18] Lofberg, Johan: *YALMIP: A Toolbox for Modeling and Optimization in MATLAB*. In *Computer Aided Control Systems Design, 2004 IEEE International Symposium on*, pages 284–289. IEEE, 2004.

[19] Lougee-Heimer, R: *The Common Optimization Interface for Operations Research: Promoting Open-Source Software in the Operations Research Community*. IBM Journal of Research and Development, 47(1):57–66, 2003.

[20] Möller, Konrad, Martin Kumm, Marco Kleinlein, and Peter Zipf: *Reconfigurable Constant Multiplication for FPGAs*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 36(6):927–937, 2017.

[21] Nielsen, Soren S: *A C++ Class Library for Mathematical Programming*. In *The Impact of Emerging Technologies on Computer Ccience and Operations Research*, pages 221–243. Springer, 1995.

[22] Oppermann, J, A Koch, M Reuter-Oppermann, and O Sinnen: *ILP-based Modulo Scheduling for High-Level Synthesis*. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'16)*, 2016.

[23] Orchard-Hays, William: *History of Mathematical Programming Systems*. Annals of the History of Computing, 6(3):296–312, 1984.

[24] Stephen J. Maher, et. al.: *The SCIP Optimization Suite 4.0*. 2017.