Zimpler - Integer Programming, easier

Javier Martínez-Viademonte
Universidad Nacional de General Sarmiento (UNGS), Argentina

2026

Abstract

This paper introduces *Zimpler*, a free tool built on the ZIMPL modeling language to streamline the solution of mixed-integer linear programs (MILP). Zimpler extends existing ZIMPL workflows by integrating native data sources—such as Excel spreadsheets—without requiring manual conversion to text-based tables. In addition, it supports solution refinement by adapting solver outputs into alternative formats, including machine-friendly exchange files like JSON. Finally, Zimpler automates the end-to-end process of model creation, solver invocation, and solution refinement, lowering the engineering effort needed to deploy optimization models in practice, unaware of the involved tools, platform and file formats.

1 Introduction

Mathematical optimization addresses the problem of finding an element, from a set of candidate solutions, that maximizes or minimizes an objective function. Different conditions on these sets and functions have given rise to specialized algorithms for this task [Sch98]. For example, cutting planes methods pioneered by Gomory [Gom58] can solve mixed-integer linear programs (MILP) [Wol21] dealing with optimization of a linear objective function, with sets described by linear expressions over both integer and real sets.

There are excellent optimization software packages, each able to tackle particular classes of problems. They are usually referred to as *codes*, *optimizers* or *solvers*. To benefit from such powerful tools, there are two main methods to express problem instances: using a solver's Application Programming Interface (API) or resorting to a compatible Modeling Language (ML).

Using its API allows for unrestricted access to the solver capabilities, often together with reduced runtime and memory usage, compared to the alternatives. However, these API-based approaches are programming language and solver-specific, which limits reusability and requires programming effort. In contrast, MLs adopt a mathematical syntax that avoids programming, makes problem formulation more concise and solver-independent [FGK90], at the cost of introducing an intermediate representation that adds computational overhead and can reduce fine-grained control over the optimization process.

Hybrid approaches also exist. For instance, a ML can be used to write an external model that is later read using an API, to exploit the advantages of both paradigms: model reusability and solver independence, together with the full expressive power of a solver's native API. However, such integration often reintroduces some of the programming complexity, and full solver-specific control may still be restricted to a single implementation.

There are solvers that integrate their own ML and are practical in some situations. However, if you want to define a model that is independent of the tool used, a common representation is necessary. In the absence of consensus, there are multiple proposals, none of which is preferred by users and supported by solver developers, at the same time. Technical, economic, and legal reasons favor one alternative or another.

From a technical point of view, modeling languages can be classified according to their level of abstraction. *Instance Modeling Languages* (IML) avoid abstractions and use concrete values. They are verbose but simple to understand and implement, yet provide all the information a solver needs to clearly define a problem. Thus, they are a sure option for a solver developer to support. In the MILP arena, most solvers accept dialects of the MPS and LP format families, as industry standard IMLs, usually as translations from other languages. At a higher level of abstraction, *Algebraic Modeling Languages* (AML) use symbolic representations for sets and parameters, are succinct, expressive and abstract, most similar to math; their models are easier to build, understand, debug, reuse and maintain. A distinctive feature is their ability to separate data from model definition. But solvers need concrete data, so an additional translation step from an AML to a compatible IML is usually required, since no AML is generally built into solvers.

Depending on their relationship with programming languages, MLs can also be classified as *embedded* or *agnostic*. A programming language embedded ML uses the syntax and utilities of an existing host, like Java or Python, to define its own modeling dialect. It leverages available libraries, brings new capabilities like visualization or runtime control, but otherwise shares traits with API usage. A programming language agnostic ML detaches from a host and defines a *Domain Specific Language* (DSL) tailored to the task. It may ignore how to process the model. In theory, its models are reusable across solvers and programming languages; in practice, they require significant supporting tools.

Regardless of its classification, every optimization model depends on data, which often originates from heterogeneous sources such as databases, local files, or remote services. These sources differ in format and access mechanism, and must be unified before optimization. Common data quality issues like inconsistency, ambiguity or incompleteness often require supplementary transformation and validation steps. Embedded MLs lead in this area due to their versatility, whereas agnostic MLs need specialized tools to be on par.

While looking for solutions, solvers produce valuable output that must be inspected. This includes *metadata*—such as termination conditions, feasibility status, and solution quality—and the *solutions* themselves, which map model variables to concrete values. As with input, differences in content, format and

interface across solvers can make handling this output cumbersome.

To put them into perspective, Bussieck and Meeraus [BM04] conceptualized these technical aspects as belonging to the solver and the modeling layers, with a higher domain or application layer representing the problem-specific context where mathematical optimization is just another analytical tool that supports decision-making. This perspective provides a better understanding of the need to present results clearly—historically, in the form of reports [Orc84], and more recently through interactive visualizations.

Comprehensive software packages now exist that address these, and other challenges, to varying degrees. Specialized packages, known as *modeling systems*, are structured around a ML; their strength lies in their attention to detail in connection with other tools, mainly an integrated development environment (IDE), solvers, data sources and programming languages. General-purpose mathematical packages, on the other hand, support broader applications such as numerical computation, statistics, or symbolic manipulation; their broader scope can introduce additional complexity, sometimes motivating interest in more focused alternatives.

Economic and legal considerations also influence tool selection. Some products are freely available, while others can be prohibitively expensive. Commercial tools regularly offer limited free versions. Licensing terms range from permissive, allowing access to full source code and unrestricted use, to restrictive, limiting access and redistribution.

Chronologically, before MLs evolved, *matrix generator* programs built the representations that the existing solvers needed. Orchard-Hays [Orc84, p. 307] tells us that, by 1965, "the best matrix generator produced until that time and an equally good report writer were designed and implemented". Still, by 1976 Meeraus [Mee76, p. 185] asserts that "mathematical programming (...) is seriously hampered by inadequate software. The major source of inadequacy is the absence of a commonly interpretable representation of the problem, model and data", and that "users were frustrated with the high cost, skill requirements, and overall low reliability of applying optimization tools" [BM04, p. 138]. Fast forward some 45 years and, by 2021, Legat et al [Leg+21, p. 673] consider that each addition "to the modeler's toolbox (...) has led to a fracturing of the optimization community as each sub-community developed a different standard form and solver for the problems of their interest". Kallrath [Kal04, p. 20] sketches an argument: "In general, the creators of established matrix generation software did not lead the switch to modeling languages; instead the first modeling languages tended to come from new developers, a number of whom were working independently toward the same goals."

The persistent demand for an unrestricted, programming language agnostic, algebraic modeling language that effortlessly works with most solvers motivates this work. To this end and to streamline the MILP solution process, this paper introduces *Zimpler*: a free tool based on the existing ZIMPL [Koc04] language.

ZIMPL is an agnostic and algebraic **language** to model optimization problems as MILP, and also a **tool** to translate this language into popular IMLs.

Zimpler is a new tool that builds on ZIMPL, with these contributions:

- accepts native input data, such as Excel spreadsheets,
- adapts solutions into new formats, like JSON exchange files,
- automates the solution process, with solver independence.

The existing ZIMPL tool requires external data to be stored in text files, formatted as tables—typically Comma-Separated Values (CSV). Database queries, JSON files and other native input data formats can't be directly used; they have to be converted first. To allow for modeling using arbitrary native data, without manual conversion, Zimpler introduces a general strategy and provides reference implementations. To the extent of our knowledge, no other free and open, programming language agnostic AML offers this capability.

Once obtained, the output from the solution process is usually refined. Sometimes, a report generated from the solution helps to interpret the result; sometimes, an easily parsable exchange file is preferred. To refine the solution, Zimpler introduces an extensible strategy and generates reference transformations. To the extent of our knowledge, no comparable tooling provides this flexibility.

The solution process involves model creation, solver invocation and solution refinement. To automate these tasks, also downloading a default solver if none is found, Zimpler introduces a platform independent strategy and implements a reference mechanism, capable of using any solver. To the extent of our knowledge, no equivalent application currently combines this versatility and scope.

Zimpler is distributed under the GNU GPLv3 license. Source code and native binaries for Windows, Mac, and Linux are available at `https://gitlab.com/zimpler-tool`. A companion tutorial [Mar24] provides complete usage examples and additional details.

The remainder of this paper is organized as follows. Section 2 reviews related work and existing tools. Section 3 presents an overview of Zimpler, before Sections 4–6 describe the three main contributions of this work—Native Data Integration, Solution Refinement, and Process Automation. Finally, Section 7 concludes discussing current limitations and outlining directions for future work.

2 Related Work

This section reviews a selection of literature and software products, using the categories introduced earlier. To provide context for this work, focus is on modeling approaches for mixed-integer linear programming (MILP).

Historically, matrix generators were the first programs specialized in loading and organizing available data according to the needs of the optimization algorithms that processed it. Orchard-Hays [Orc84] offers a detailed view of the early systems. Matrix generators were soon followed by the first modeling languages [Fou83], now referred to as Instance Modeling Languages (IML). Their family includes MPS, NL, OSiL, PuLP, and many others.

In the mid-1960s, IBM introduced Mathematical Programming System/360 (MPS) [IBM67]. As was customary at the time, the language prioritized ease of use with punched cards over readability—an apparently clever choice, as it became widely adopted. Several extensions were later developed to overcome its limitations, not always mutually compatible. More recent versions [ILO07] evolved into an industry standard supported by most solvers. The other language currently supported directly by many solvers is LP, known for its simplicity and readability. Unfortunately, multiple dialects coexist, such as those used by CPLEX, Xpress, or lp_solve, among others [lps25; GAM25c].

Beyond languages directly supported by solvers, some IMLs serve as intermediaries for Algebraic Modeling Languages (AML), relying on auxiliary programs for integration. Created for this purpose were the NL format [Gay05] and FlatZinc [Raf+06], associated with AMPL and MiniZinc, respectively.

An alternative path proposes defining an open format that serves as a standard for instance exchange, with independent implementations embracing it. Among the candidates, "Optimization Services instance Language" (OSiL) [FMM10] stands out, based on XML and integrated with COIN-OR solvers. "Extended MPS" (xMPS) [Tho99] was similarly promoted by Maximal Software [HTK06].

A final significant group of IMLs consists of programming libraries that define embedded instance modeling dialects within general-purpose languages. PuLP [MOD11] is a notable example for Python, while MathOpt [Goo26], part of OR-Tools [Per11], supports both C++ and Python.

IMLs were succeeded by AML, which are now habitually adopted. This family includes AMPL, GAMS, JuMP, and ZIMPL, among others.

A report in 1976 outlined progress toward the first algebraic modeling system [Mee76]. That effort led to the creation of "General Algebraic Modeling System" (GAMS) [GAM25a], a comprehensive modeling system that evolved well beyond MILP into a multi-platform product integrated with major solvers. GAMS can read and transform a wide range of data sources and, over time, incorporated many complementary technologies—from a model editor to a cloud-instance manager, an interactive application generator, and programming interfaces for .NET, C++, Java, Python, MATLAB, and R. Interestingly, "the major part of GAMS source code is dedicated to data calculations and reporting" [BM04, p. 145].

Other prominent modeling systems include "Advanced Interactive Multidimensional Modeling System" (AIMMS) [RB23], "A Mathematical Programming Language" (AMPL) [FGK03], "Linear, INteractive, and Discrete Optimizer" (LINDO) [LIN03] with its LINGO language [LIN24], Xpress [FIC26b] with its Mosel language [FIC26a], "Mathematical Programming Language" (MPL) [Max18], and "CPLEX Optimization Studio" [IBM25] with its "Optimization Programming Language" (OPL) [Van99].

In the broader family of mathematical computing software, we find Mathematica [Wol25] with its Wolfram language, MATLAB [Mat25a] and its Optimization Toolbox [Mat25b], Octave [Eat+25], SageMath [Tea25], and TOMLAB

[HGE10], among others.

A final group of specialized languages model MILPs and, sometimes, broader problem classes, though without the full suite of auxiliary tools. Examples include "GNU Mathematical Programming Language" (GMPL or MathProg) [MS20], MiniZinc [Net+07], and "Zuse Institute Mathematical Programming Language" (ZIMPL) [Koc04].

As with IMLs, there are AML proposals emphasizing interoperability rather than modeling convenience; they favor processing ease or flexibility, over readability. In this category lies MathOptFormat (MOF), a JSON-based direct serialization of models created with MathOptInterface (MOI) [Leg+21].

AMLs also feature libraries that define dialects embedded in general-purpose programming languages. Examples include "Julia for Mathematical Programming" (JuMP) [DHL17] for Julia, and GAMSPy [GAM25b], "Python Optimization Modeling Objects" (Pyomo) [Byn+21] or "Python Optimization Interface" (PyOptInterface) [Yan+24] for Python. Also worth mentioning are YALMIP [Löf04], a MATLAB dialect, and GEKKO [Bea+18], a Python dialect with modeling system features.

Just as convex optimization generalizes linear programming, restricting some variables to be integers gives rise to mixed-integer convex programming (MICP) as a generalization of MILP. Among embedded AMLs, some can model MICP and therefore MILPs as well. Examples include CVX [GB08; GB24] for MATLAB, and its derivatives CVXPY [DB16] for Python, CVXR [FNB20] for R, and Convex.jl [Ude+14] for Julia.

Table 1 summarizes the reviewed modeling languages.

	instance		algebraic	
	open	closed	open	closed
agnostic	FlatZinc LP MPS NL OSiL	xMPS	GMPL MiniZinc ZIMPL	AIMMS AMPL GAMS LINGO Mosel MPL OPL
(general purpose) embedded	PuLP MathOpt	–	Convex.jl CVXPY CVXR GEKKO JuMP Pyomo PyOptInterface SageMath	GAMSPy
other	–	–	CVX Octave YALMIP MOF	Optimization Toolbox TOMLAB Wolfram

Table 1: Summary of reviewed modeling languages

All reviewed IMLs have free and open-source implementations, with the sole exception of xMPS. The same applies to AMLs embedded in general-purpose programming languages, with the sole exception of GAMSPy. Non-embedded AMLs are divided between open and closed-source (proprietary) tools, with newer languages somehow more likely to be open source.

Many other modeling languages and features exist, each contributing in its own specific ways. See "Modeling Languages in Mathematical Optimization" [Kal04] and "Algebraic Modeling Systems" [Kal12] for a comprehensive treatment with a historical perspective; they cover both theoretical and practical aspects, and survey a broad range of languages and systems, including some not highlighted here.

3 Zimpler Overview

Zimpler is a free tool, based on ZIMPL, that leverages open-source applications to streamline MILP optimization. ZIMPL defines an Algebraic Modeling Language (AML) for MILP problems. Zimpler integrates existing applications and augments their capabilities, delivering a unified optimization workflow. The design of Zimpler is centered around three interfaces that decouple the optimization process from important external dependencies:

`Loader:` Responsible for reading native input data formats (Excel, DOT, ...) and solvers' output formats.

`Optimizer:` Abstracts accidental variations from existing toolchains, including differences in IML, solvers, platforms and pipelines.

`Presenter:` Transforms raw solver output into readily usable form, such as reports and exchange files.

In a traditional setup, data in a spreadsheet could be exported as a CSV file. Columns may need to be rearranged or the model edited to fit the data. An AML reader and some solver would need to be installed. A solver-dependent API or solution file format would need to be considered, to assess feasibility, quality, or, eventually, a solution. No readily usable final result is typically available from open-source toolchains.

Zimpler reads ZIMPL models, translates and binds native input data, downloads platform-compatible open-source applications as needed, invokes a solver, and presents its output in a user-friendly form. For example, if `model.zpl`, `model.xlsx` and `model.properties` exist in the working directory, running

```
$ zimpler
```

from the command line will find the default-named `model.zpl`, bind two data sources and generate file `result.xlsx` with a solution, if found, and metadata about the process.

The following sections detail how Zimpler achieves this through Native Data Integration (Section 4), Solution Refinement (Section 5), and Process Automation (Section 6). Installation and Basic Usage instructions can be found in the project's repository [Mar26c]. Advanced usage instructions and complete examples can be found in the companion tutorial [Mar24], as well as the project documentation [Mar26b].

4 Native Data Integration

Input data for mathematical optimization problems is often found in a variety of formats, such as spreadsheets, databases or graphs, not always supported by modeling languages. What follows details Zimpler's strategy to allow for modeling with the ZIMPL language and unmodified tools, unaware of the different input data formats.

The ZIMPL language requires data elements to be strictly **defined**; i.e.: *Parameters* and *Sets* have to be assigned concrete values. In contrast, many languages allow for elements to be **declared**, but left undefined. The separation of data declarations from their definition enables the introduction of a binding mechanism, with new capabilities. The process of finding and assigning values to existing **data elements** is referred to as *resolution*, not to be confused with *solution*, where values are assigned to **variables**. An extended ZIMPL grammar and parser were developed [Mar26a] to support a new resolution mechanism. The grammar accepts the declaration of undefined data elements and the generated parser is the foundation of Zimpler.

After reading model files, Zimpler analyzes data elements. Defined elements remain unchanged. Declared but undefined elements cause invocation of the new resolution algorithm. Together with Zimpler, a collection of data loaders were developed, currently accepting Excel spreadsheets, Graphviz DOT graphs, Properties files and Comma Separated Values (CSV). Zimpler searches the local storage for available data sources and their content is scanned for candidate data definitions. For example, Sheet and Column names in Excel spreadsheets are compared against undefined declarations. Each data loader conforms to a `Loader` interface defined by Zimpler and implements its own scanning mechanism. See their documentation for format specific details, available in the "Usage" section of the Zimpler documentation [Mar26b].

The purpose of the new resolution algorithm is to assist in mathematical modeling, disregarding input data nuances. Data format is only one of such nuances. The `Loader` interface also permits abstraction of data location; data sources need not be restricted to local files. It's also often the case that data sources relate to existing, maybe predating, information systems. Adapting these systems to fit a mathematical model may be inconvenient in practice. Adapting the model is sometimes easier, but it sacrifices reusability. The resolution mechanism provides an *aliasing* facility that solves this problem when relabeling is enough for matching to succeed. To bind a model declaration for `X` with an attribute `Y` in an input file, defining "`X = Y`" in file `alias.properties` should bridge the resolution gap.

If an undefined model element cannot be matched against any candidate data source definition, the resulting element will remain unchanged. If multiple matches are found, an arbitrary one is chosen and a warning message is displayed. When a match is found a decision is made about how to realize the definition. Simple cases end up inlined while complex definitions require further processing; e.g.: a declared parameter `n` and a Properties file containing "`n = 8`" result in the definition "`param n := 8;`", inlined within the model. More elaborate definitions follow different paths.

The simplest such cases are CSV definitions, because the original ZIMPL tool can directly load them. The phrase "`param weight[Item];`", declares a `weight` parameter, indexed over the set `Item`. If this declaration were resolved against file `model.csv`, with content:

```
Item, weight
rock, 3
paper, 1
scissors, 2
```

the resulting definition would be

```
param weight[Item] := read "model.csv" as "<1s> 2n" skip 1;
```

indicating that an appropriate `model.csv` data source has been found, that the indexing `Item` corresponds to the first column of data, of string type, that the parameter values correspond to the second column, of numeric type, and that

one (header) row should be skipped from data reading. Here, a reading phrase is enough for the resulting model to be fully compatible with the original ZIMPL tool.

Data formats not supported by the original tool additionally require generation of acceptable data files. For example, the current implementation for the Graphviz DOT format loader generates a whitespace delimited TXT file containing the graph's node set, as well as a CSV file containing the graph's edge set and their attributes, if any; see [Mar24] for a complete example. A read phrase is used again to bind the generated data files with undefined declarations.

To recap, a new grammar for the ZIMPL language was created and used to build Zimpler's extensible and resolving parser. Optimization models can now declare input data, unaware of their native format, location or data labeling. To accept a new data format, an implementation of Zimpler's `Loader` interface can either inline or translate input data into a ZIMPL-readable format. Resulting models are fully compatible with the original ZIMPL tool and, thus, enable native data to be integrated into an existing toolchain, without need for manual data conversion.

5 Solution Refinement

In practice, the output from the solution process is often refined into a user-friendly form. What follows details the general solution refinement strategy that Zimpler develops, and presents its current implementations.

Zimpler reads solution files for the different solvers with the same `Loader` interface defined earlier and used to integrate native data. It currently provides concrete implementations to read CBC [For24] and SCIP [Bol+24] solutions. Since each solver outputs information in similar but incompatible ways, the rationale for this version of the tool was to aim for a basic, yet functional enough design. A solution is considered a plain mapping from model variable names to numeric values. To accommodate metadata present in existing solution file formats, within this mapping, synthetic identifiers starting with a forbidden first character "#" for variable names were used.

In addition to metadata found in solution files, such as infeasible or unbounded instance status, or value of the objective function at the best solution found, basic bookkeeping information like model filename or involved solver can also be accommodated. Each solver requires a custom solution loader and specifies its own set of metadata elements. See their documentation for specific details, available in the "Usage" section of Zimpler documentation [Mar26b].

After the solution is extended with metadata, Zimpler distinguishes between standalone and indexed variables. Standalone variables are grouped together with the metadata, while indexed variables are considered grouping criteria and returned separately.

A table named "Summary" collects metadata and maps standalone variables to values. An additional table is created and named after each indexed variable. Indexes and the associated value are treated as columns. Zero valued assign-

ments are omitted by default. Tables are named, bidimensional arrays with a mandatory first row of text labels, called *header*. Solution refinement produces a list of tables, beginning with the Summary. See Table 2 for an example.

	key	value	V	C	value
`x_1_1 = 1`	#solver:cbc		1	1	1
`x_1_2 = 0`	#gap:0		2	2	1
`...`	alone	2			
`alone = 2`					
`cbc-output.sol`	Summary		x		

Table 2: Refinement of `cbc-output.sol` into tables "Summary" and "x".

Sometimes, a traditional textbook problem is used as a model for practical applications. Its formulation often names generic elements, while their applications usually name very specific ones. In order not to force either the model, nor the data, to adapt, the same aliasing device introduced for native data integration is reused here. The refinement mechanism replaces model names with their aliased partners. A subtle but related detail concerns the singular and plural forms of text labels. For presentation purposes, sometimes a plural form is preferred over its singular counterpart. Refinement assumes plural forms for the presentation of sets and singular forms for elements. The simple strategy of adding a trailing letter "s" for the plural is combined with an exceptions map that covers more elaborate cases. A complete example showing aliases, plurals and exceptions can be found in the "Usage" section of the Zimpler documentation [Mar26b].

The purpose of refinement is to ease solution consumption. From the generated tables, a `Presenter` interface allows its implementations to draw concrete realizations. Zimpler currently offers two complementary alternatives: an Excel spreadsheet, as an example of a finished user-facing report, and a JSON output file, as an example of an exchange format intended for external further processing.

From the results of the solution process, Zimpler developed a general solution refinement scheme that allows both the creation of finished products or intermediate exchange files. Leveraging the `Loader` interface, solver specific solution formats are turned into a uniform list of tables that assign values to variables while accommodating for extra metadata, effectively eliminating format differences between solvers. Simple aliasing and plural handling enhance exhibition details. While Zimpler currently generates popular output formats, its `Presenter` interface is able to follow alternative routes, including creation of interactive visualizations or in-memory database integration, without intermediate files.

6 Process Automation

The solution process typically involves a sequence of discrete steps. What follows details Zimpler's strategy to automate this process.

A simplified sequence to solve a mathematical program is:

1. generate model
2. invoke solver
3. read solution

In practice, however, it may be necessary to deal with additional aspects: load input from heterogeneous sources, combine data with an abstract model, consider differences between solvers and translate their specific solution formats into a final document. Additionally, tools tackling particular steps need to be provisioned; for example, if ZIMPL is used in step 1 and CBC is used in step 2, both tools need to be installed.

Not only file formats differ between solvers; they also have varying configuration and invocation requirements. Also, environment conditions such as cloud services pricing may favor or restrict usage of resources like storage. To account for this diversity, Zimpler defines an `Optimizer` interface, representing a toolchain scheme. Zimpler's only implementation to date is `CliOptimizer`, for a command-line interface (CLI) approach. Alternative designs are possible, e.g., tailored for solver-specific APIs without access to storage.

The supplied `CliOptimizer` communicates with programs using text files. It writes a resolved, ZIMPL-compliant temporary model file, then invokes a platform-dependent shell script: Windows, Mac and Linux are currently supported. File `solve.bat` (Windows) or `solve.sh` (Mac & Linux) is looked for; if it can't be located in the current directory, a default version is created. Existing or created, Zimpler transfers control to this file. The default script looks for installed versions of the original ZIMPL tool and the CBC solver. If they are not found locally installed, the user is prompted to allow for the missing programs to be downloaded from a default mirror, and a platform specific runtime is saved in the current directory, execution permissions adjusted; if any of them is found, no downloading is attempted for it. The default script uses the ZIMPL tool to translate the resolved model into a temporary LP file, then invokes the CBC solver with appropriate configuration and parameters, handling invocation errors. After execution, control is returned from the script to Zimpler, allowing solution refinement to proceed. To use an alternative solver, IML or processing pipeline, editing the default shell script is enough.

Zimpler streamlines MILP optimization by automating the solution process, and thus avoiding repetitive and error-prone effort. It takes an abstract model, provisions the required programs, translates and merges native input data, generates a compatible instance model, solves the problem and outputs a refined solution, unconstrained by the computing platform.

7 Conclusion

According to the NEOS Server's [GM97] Statistics Report [Uni26], the preferred input formats for mathematical optimization problems correspond to closed-source, programming language agnostic (PLA), algebraic modeling languages (AML). Their open-source counterparts account for a very small fraction of the submitted jobs, even if all instance modeling languages (IML) were added to this count, assuming translations. Ease of use is probably a distinguishing feature and it motivates this work.

Zimpler delivers integration capabilities not available in PLA, open-source AML, but frequent in the closed-source realm. It compares fairly to the early versions of proprietary AML, but is still years behind their modern incarnations, with a catch: its source code is provided completely free. Learning from the past, a design principle for Zimpler has been to integrate with existing tools, unmodified, in order to avoid community fragmentation. With the classroom in mind, emphasis was put on easing novice usage. It allows modeling, unaware of the data sources, and yields friendly results, independent from the solver vendor. Out of the box, Zimpler provides implementations for popular solvers and input and output formats, with an extension mechanism for further needs.

Despite its contributions, Zimpler has many pending issues left. The small but important ones: add more data sources and formats, more configuration options and more solvers. The demanding ones: the current design constrains development to a single programming language for all tasks; inter-process communication (IPC) could allow data loaders and modeling functions to be defined externally, in any language; in addition, it could pave the way for some form of algebraic dynamic cuts, a feature we are unaware of previous existence. The social ones: Zimpler could be easily turned into a library, with an API for each programming language; it could also be added to the existing marketplaces of the different computing platforms; and, most challenging, it could be built-into solvers, pending community and licensing considerations.

The last few options focus on a narrow context, but there are relevant pieces missing in the puzzle of mathematical optimization. Preprocessing, term rewriting and model decomposition, each opens vast opportunities for further work.

Finally, a question. Should we, as a community, follow the footsteps of the "Language Server Protocol" (LSP) [Mic22] and the "Model Context Protocol" (MCP) [Ant25], into a new "Mathematical Optimization Protocol" (MOP) that builds on the ideas of MathOpt Interface [Leg+21]?

References

[Ant25] Anthropic. *Model Context Protocol (MCP)*. Version 2025-11-25. 2025. URL: `https://modelcontextprotocol.io`.

[Bea+18] Logan D. R. Beal et al. "GEKKO Optimization Suite". In: *Processes* 6.8 (2018). Article number: 106. DOI: `10.3390/pr6080106`.

[BM04] Michael R. Bussieck and Alex Meeraus. "General Algebraic Modeling System (GAMS)". In: *Modeling Languages in Mathematical Optimization*. Springer, 2004, pp. 137–157. DOI: 10.1007/978-1-4613-0215-5_8.

[Bol+24] Suresh Bolusani et al. *The SCIP Optimization Suite 9.0*. ZIB-Report 24-02-29. Zuse Institute Berlin, Feb. 2024. URL: https://nbn-resolving.org/urn:nbn:de:0297-zib-95528.

[Byn+21] Michael L. Bynum et al. *Pyomo — Optimization Modeling in Python*. 3rd. Vol. 67. Optimization and Its Applications. Springer, 2021. DOI: 10.1007/978-3-030-68928-5.

[DB16] Steven Diamond and Stephen P. Boyd. "CVXPY: A Python-Embedded Modeling Language for Convex Optimization". In: *Journal of Machine Learning Research* 17.83 (2016), pp. 1–5. URL: http://jmlr.org/papers/v17/15-408.html.

[DHL17] Iain Dunning, Joey Huchette, and Miles Lubin. "JuMP: A Modeling Language for Mathematical Optimization". In: *SIAM review* 59.2 (2017), pp. 295–320.

[Eat+25] John W. Eaton et al. *GNU Octave Manual*. Version 10.2. May 2025. URL: https://docs.octave.org/octave.pdf.

[FGK03] Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. 2nd. Duxbury / Thomson, 2003. ISBN: 0-534-38809-4.

[FGK90] Robert Fourer, David M. Gay, and Brian W. Kernighan. "A Modeling Language for Mathematical Programming". In: *Management Science* 36.5 (May 1990). INFORMS, pp. 519–554. ISSN: 0025-1909. DOI: 10.1287/mnsc.36.5.519.

[FIC26a] FICO. *Mosel Help*. 2026. URL: https://www.fico.com/fico-xpress-optimization/docs/latest/mosel.

[FIC26b] FICO. *Xpress Optimization*. 2026. URL: https://www.fico.com/xpress.

[FMM10] Robert Fourer, Jun Ma, and Kipp Martin. "OSiL: An instance language for optimization". In: *Computational Optimization and Applications* 45.1 (2010), pp. 181–203. DOI: 10.1007/s10589-008-9169-6.

[FNB20] Anqi Fu, Balasubramanian Narasimhan, and Stephen Boyd. "CVXR: An R Package for Disciplined Convex Optimization". In: *Journal of Statistical Software* 94.14 (2020), pp. 1–34. DOI: 10.18637/jss.v094.i14.

[For24] John Forrest. *COIN Branch and Cut (CBC)*. Version 2.10. Aug. 2024. URL: https://github.com/coin-or/Cbc.

[Fou83] Robert Fourer. "Modeling Languages Versus Matrix Generators for Linear Programming". In: *ACM Transactions on Mathematical Software (TOMS)* 9.2 (1983), pp. 143–183. DOI: 10.1145/357456.357457.

[GAM25a] GAMS. *Documentation Center*. Version 50. 2025. URL: https://www.gams.com/latest/docs/.

[GAM25b] GAMS. *GAMSPy*. Version 1.18. 2025. URL: https://github.com/GAMS-dev/gamspy.

[GAM25c] GAMS. *MPS2GMS Tool Manual*. Version 50. 2025. URL: https://www.gams.com/50/docs/T_MPS2GMS.html.

[Gay05] David M. Gay. *Writing .nl Files*. Tech. rep. SAND2005-7907P. Sandia National Laboratories, Nov. 2005. URL: https://www.osti.gov/biblio/1727372.

[GB08] Michael C. Grant and Stephen P. Boyd. "Graph Implementations for Nonsmooth Convex Programs". In: *Recent Advances in Learning and Control*. 2008, pp. 95–110. DOI: 10.1007/978-1-84800-155-8_7.

[GB24] Michael C. Grant and Stephen P. Boyd. *CVX: Matlab Software for Disciplined Convex Programming*. Version 2.2.2. Apr. 2024. URL: https://cvxr.com/cvx.

[GM97] William Gropp and Jorge J. Moré. "Optimization Environments and the NEOS Server". In: *Approximation Theory and Optimization*. Cambridge University Press, 1997, pp. 167–182.

[Gom58] Ralph E. Gomory. "Outline of an Algorithm for Integer Solutions to Linear Programs". In: *Bulletin of the American Mathematical Society* 64.5 (Sept. 1958), pp. 275–278.

[Goo26] Google. *MathOpt*. Version 9.15. Jan. 2026. URL: https://developers.google.com/optimization/math_opt.

[HGE10] Kenneth Holmström, Anders O. Göran, and Marcus M. Edvall. *User's Guide for TOMLAB 7*. May 2010. URL: https://tomopt.com/docs/TOMLAB.pdf.

[HTK06] Bjarni V. Halldórsson, Erlendur S. Thorsteinsson, and Bjarni Kristjánsson. "A Modeling Interface to Non-Linear Programming Solvers". 2006. URL: https://www.maximalsoftware.com/resources/xmps.

[IBM25] IBM. *CPLEX Optimization Studio*. Version 22.1. 2025. URL: https://www.ibm.com/products/ilog-cplex-optimization-studio.

[IBM67] IBM. *Mathematical Programming System/360 Linear Programming User's Manual*. Form H20-0291, file number 360A-CO-14X. 1967.

[ILO07] ILOG. *CPLEX 11.0 File Formats*. Reference Manual. Sept. 2007.

[Kal04] Josef Kallrath, ed. *Modeling Languages in Mathematical Optimization*. Vol. 88. Applied Optimization 5634. Kluwer, 2004. ISBN: 978-1-4020-7547-6. DOI: 10.1007/978-1-4613-0215-5.

[Kal12] Josef Kallrath, ed. *Algebraic Modeling Systems: Modeling and Solving Real World Optimization Problems*. Vol. 104. Applied Optimization 5634. Springer, 2012. ISBN: 978-3-642-23591-7. DOI: 10.1007/978-3-642-23592-4.

[Koc04] Thorsten Koch. "Rapid Mathematical Prototyping". Report 04-58, ZIB. PhD thesis. Technische Universität Berlin, Dec. 2004. URL: https://opus4.kobv.de/opus4-zib/files/834/ZR-04-58.pdf.

[Leg+21] Benoît Legat et al. "MathOptInterface: A Data Structure for Mathematical Optimization Problems". In: *INFORMS Journal on Computing* 34.2 (2021), pp. 672–689. DOI: 10.1287/ijoc.2021.1067.

[LIN03] LINDO. *User's Manual*. 2003. URL: https://lindo.com/downloads/PDF/LindoUsersManual.pdf.

[LIN24] LINDO. *LINGO - The Modeling Language and Optimizer*. 2024. URL: https://lindo.com/downloads/PDF/LINGO.pdf.

[Löf04] Johan Löfberg. "YALMIP : A Toolbox for Modeling and Optimization in MATLAB". In: *In Proceedings of the CACSD Conference*. Taipei, Taiwan: IEEE, Sept. 2004. ISBN: 0-7803-8636-1. DOI: 10.1109/CACSD.2004.1393890.

[lps25] lp_solve. *LP file format*. lp_solve 5.5.2.14 Reference Guide. 2025. URL: https://lp-solve.github.io/lp-format.htm.

[Mar24] Javier Martínez-Viademonte. "Zimpler: use native input data with ZIMPL". In: *IFORS Newsletter* 19.3 (Sept. 11, 2024), pp. 5–6. ISSN: 2223-4373. URL: https://ifors.org/newsletter/ifors-news-sept-2024.pdf#page=5.

[Mar26a] Javier Martínez-Viademonte. *ANTLR grammar and Java parser for Zimpl*. Version 0.4. 2026. URL: https://gitlab.com/antlr-java-parser/zimpl.

[Mar26b] Javier Martínez-Viademonte. *Zimpler Documentation*. Version 0.5. 2026. URL: https://gitlab.com/zimpler-tool/main/blob/main/README.md.

[Mar26c] Javier Martínez-Viademonte. *Zimpler Project Repository*. Version 0.5. 2026. URL: https://gitlab.com/zimpler-tool.

[Mat25a] MathWorks. *MATLAB User's Guide*. R2025A. 2025. URL: https://www.mathworks.com/help/matlab.

[Mat25b] MathWorks. *Optimization Toolbox User's Guide*. R2025a. 2025. URL: https://www.mathworks.com/help/optim.

[Max18] Maximal. *MPL Modeling System - Manual*. Release 5.0. Arlington, VA, USA, Jan. 2018. URL: https://www.maximalsoftware.com/mplman.

[Mee76] Alexander Meeraus. "Toward a General Algebraic Modelling System". In: *IX International Symposium on Mathematical Programming*. Ed. by András Prékopa. Budapest, Hungary: Bolyai János Mathematical Society, 1976, p. 185.

[Mic22] Microsoft. *Language Server Protocol (LSP)*. Version 3.17. Oct. 2022. URL: https://microsoft.github.io/language-server-protocol (visited on 01/24/2026).

[MOD11] Stuart A. Mitchell, Michael O'Sullivan, and Iain Dunning. *PuLP: A Linear Programming Toolkit for Python*. Sept. 2011. URL: https://optimization-online.org/?p=11731.

[MS20] Andrew Makhorin and Heinrich Schuchardt. *Modeling Language GNU MathProg*. GLPK 5.0. Free Software Foundation. Dec. 2020. URL: https://www.gnu.org/software/glpk.

[Net+07] Nicholas Nethercote et al. "MiniZinc: Towards a Standard CP Modelling Language". In: *Principles and Practice of Constraint Programming – CP 2007*. Ed. by Christian Bessière. Berlin, Heidelberg, Germany: Springer, 2007, pp. 529–543. ISBN: 978-3-540-74970-7. DOI: 10.1007/978-3-540-74970-7_38.

[Orc84] William Orchard-Hays. "History of mathematical programming systems". In: *IEEE Annals of the History of Computing* 6.3 (July 1984), pp. 296–312. ISSN: 1058-6180. DOI: 10.1109/MAHC.1984.10032.

[Per11] Laurent Perron. "Operations Research and Constraint Programming at Google". In: *Principles and Practice of Constraint Programming – CP 2011*. Ed. by Jimmy Lee. Berlin, Heidelberg, Germany: Springer, 2011, p. 2. ISBN: 978-3-642-23786-7. DOI: 10.1007/978-3-642-23786-7_2.

[Raf+06] Reza Rafeh et al. "From Zinc to Design Model". In: *Practical Aspects of Declarative Languages*. Ed. by Michael Hanus. Vol. 4354. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg, Germany: Springer, 2006, pp. 215–229. ISBN: 978-3-540-69611-7. DOI: 10.1007/978-3-540-69611-7_14.

[RB23] Marcel Roelofs and Johannes Bisschop. *The Language Reference*. AIMMS, June 2023. URL: https://documentation.aimms.com/language-reference/index.html.

[Sch98] Alexander Schrijver. *Theory of Linear and Integer Programming*. Discrete Mathematics and Optimization. Wiley, June 1998. ISBN: 978-0-471-98232-6.

[Tea25] The Sage Development Team. *Sage Documentation*. Version 10.6. 2025. URL: https://doc.sagemath.org.

[Tho99] Erlendeur S. Thorsteinsson. *xMPS, the Extended MPS Format for Non-Linear Programs*. Tech. rep. 99-224. Pittsburgh, PA, USA: Carnegie Mellon University, Dec. 1999. DOI: 10.1184/R1/6480236.v1.

[Ude+14] Madeleine Udell et al. "Convex Optimization in Julia". In: *SC14 Workshop on High Performance Technical Computing in Dynamic Languages* (2014). arXiv: 1410.4821 [math-oc].

[Uni26] Madison University of Wisconsin. *NEOS Solver Statistics*. 2026. URL: https://neos-server.org/neos/report.html.

[Van99] Pascal Van Hentenryck. *The OPL Optimization Programming Language*. Cambridge, MA, USA: MIT, 1999. ISBN: 0-262-72030-2.

[Wol21] Laurence A. Wolsey. *Integer Programming*. 2nd. NJ, USA: Wiley, 2021. ISBN: 978-1-119-60653-6. DOI: 10.1002/9781119606475.

[Wol25] Wolfram. *Mathematica*. Version 14.3. 2025. URL: https://www.wolfram.com/mathematica.

[Yan+24] Yue Yang et al. *PyOptInterface: Design and implementation of an efficient modeling language for mathematical optimization*. 2024. arXiv: 2405.10130 [cs.MS].