`libmzn`
A modular CP infrastructure based on MiniZinc

Guido Tack

Katholieke Universiteit Leuven, Belgium
guido.tack@cs.kuleuven.be

Abstract. The main obstacle to a successful integration of MiniZinc models within general applications is the monolithic, text-based interface to the MiniZinc toolchain. Both the frontend and the individual solver backends require a custom, error-prone implementation of data exchange via text files.
This position paper proposes to develop an *infrastructure* for constraint modeling based on MiniZinc, rather than just a modeling language. This will ensure a greater impact of MiniZinc, and a better chance of it being accepted as a standard. Such an infrastructure could be based on a modular library for MiniZinc, called `libmzn`, featuring Application Programming Interfaces (APIs) in C and C++ for both modeling and solving. This architecture will make it easy to integrate the MiniZinc toolchain into applications and general-purpose programming languages, as well as provide a direct interface for solver backends.

1 Introduction

Since its inception several years ago, MiniZinc [9] has gained considerable momentum, and the MiniZinc distribution has become stable and useable. The toolchain is available for all major platforms, easy to install, and easy to experiment with. The language is expressive enough for non-trivial problems.

Still, the use of MiniZinc is very much limited to research and education. A major obstacle to a broader impact of MiniZinc is its monolithic, text-based architecture. Applications and solvers have to use MiniZinc via text files, leading to cumbersome, error-prone interfacing code and an unnecessary performance overhead.

MiniZinc and FlatZinc are Domain Specific Languages (DSLs). At its core, the MiniZinc toolchain is a compiler from one DSL to another. So let us look at the most successful compiler project in recent years: LLVM [7]. The *low-level virtual machine* provides a modular infrastructure for building optimizing compilers for general-purpose programming languages. Why has LLVM been so successful?

1. It is **modular** and **extensible**: different optimization passes can be plugged together; the intermediate representations are well-defined and serve as interfaces between different compiler phases.
2. It supports **different frontends and backends**. For instance, the `clang` project provides production-quality compiler frontends for C, C++ and Objective C, and C-like languages such as OpenCL; other frontends for languages like Haskell, Java, and Python, are being developed. Backends for x86, ARM, and different GPU architectures exist.

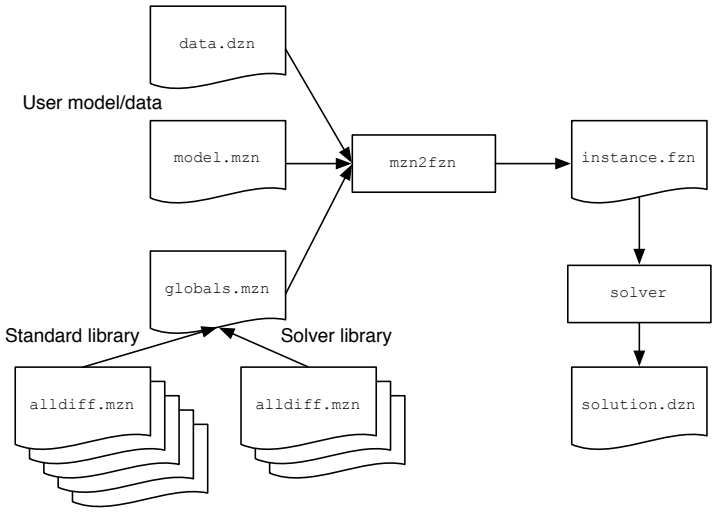

Fig. 1. The MiniZinc toolchain

3. It is written in **standards-compliant C++** and can be **integrated** through a **clean API**. Applications, backends, assemblers, and debuggers can be plugged in directly, without the detour through plain text.
4. It is **efficient**. Compilation using `clang` is several times faster than using the GNU compiler `gcc`, while the resulting code performs comparably.
5. It is **well documented**. This is essential for an infrastructure project.

Arguably, the current MiniZinc toolchain has deficits in all these areas. The message of this paper is therefore to not only develop MiniZinc as a standard modeling language for the CP community, but as a *standard infrastructure*:

Let us turn MiniZinc into the LLVM for constraint problems.

Such a standard constraint infrastructure would come in the form of a C/C++ library, `libmzn`, which provides parsing, compilation, and solver interfacing for MiniZinc models. In order to remain backwards compatible, the current toolchain can be reimplemented easily on top of `libmzn`.

Overview. The next section recapitulates the current state and the drawbacks of the MiniZinc toolchain. Sect. 3 gives a brief overview of the architecture of LLVM. The following section designs an architecture for `libmzn` based on the LLVM example and discusses its advantages. Sect. 5 presents related work, and Sect. 6 concludes the paper.

2 State of MiniZinc

Fig. 1 shows the current architecture of the MiniZinc toolchain. The user creates a model and possibly several data files. The model includes the MiniZinc library of global

constraint definitions. The target solver provides specialized definitions of the global constraints it supports natively. The `mzn2fzn` tool compiles all this MiniZinc code into a single FlatZinc output file, which is then passed to the solver. Solutions are written into an output file, again in valid MiniZinc syntax[1].

The MiniZinc approach has the following advantages compared to using solvers directly:

- The user can experiment with different solvers and even solving techniques.
- MiniZinc is a high-level language, many models can be expressed quite elegantly (e.g. using comprehensions, predicates, multi-dimensional arrays).
- The MiniZinc distribution comes with a comprehensive library of global constraints, with clearly defined interfaces and a semantics defined in terms of decompositions.

While the MiniZinc language and the general approach have proven successful, the actual implementation has a number of drawbacks.

Application integration issues. An application that wants to solve combinatorial problems using MiniZinc needs to invoke the MiniZinc toolchain in a separate process, provide the model and instance data as text files, and retrieve the solutions or error messages from the resulting textual output. This requires custom, error-prone interfacing code and has a performance overhead compared to a direct API.

Solver integration issues. In order to support MiniZinc, a solver has to parse FlatZinc (either directly or in its XML form). Although this is a relatively straight-forward requirement (FlatZinc parser skeletons are available), a direct API could increase adoption as well as performance.

Compilation issues. The compilation of MiniZinc to FlatZinc is completely monolithic. Apart from supplying solver-specific global constraint definitions and redefinitions of built-in predicates, a solver provider does not have any influence on the translation. Furthermore, if an application has to solve the same problem with different parameters over and over again, the MiniZinc approach requires the complete compilation to be performed every time, at runtime. Separate compilation is not possible.

Mercury. Implementing the MiniZinc toolchain in Mercury certainly has advantages, as a high-level functional language is better suited for compiler construction than e.g. C++. On the other hand, Mercury is a niche language, which seriously raises the threshold for adoption and integration into mainstream applications.

3 State of the Art: LLVM

While the source and target languages of LLVM and MiniZinc differ substantially, the architecture of LLVM can serve as a starting point for designing an architecture for `libmzn`.

[1] The processing of output items is omitted here to keep things simple. In the target scenario of this paper, the processing of solutions would be handled by the application.

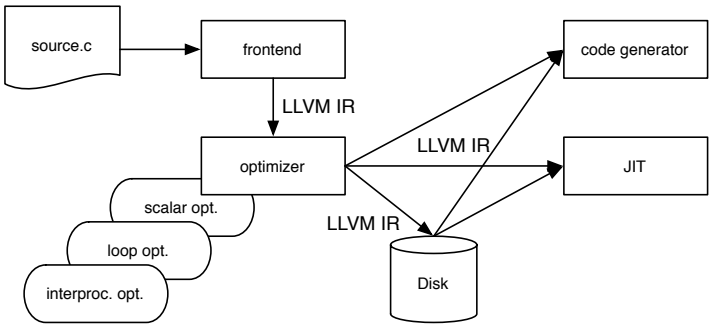

Fig. 2. The LLVM architecture

Fig. 2 shows how the individual components are connected in LLVM. The main lessons to learn from this architecture are:

- Components are connected by a common, well-defined intermediate representation (LLVM IR).
- The translation phase is modular, the user can pick and choose the optimization passes and code generation backends that are suitable for the particular application.
- The IR is not only used at runtime, but can also be stored on disk, enabling partial separate compilation without giving up optimization potential at runtime.

4 An Architecture for `libmzn`

This section presents a rough sketch of an architecture for `libmzn`, based on the design of LLVM, and discusses the potential benefits of such a library.

4.1 Basic Design

The basic design of `libmzn` follows that of LLVM. The library is implemented in C++, built around a well-defined intermediate representation (IR), with a frontend that creates IR from MiniZinc, optimization passes that transform the IR, and solver backends.

Intermediate Representation. The MiniZinc IR represents MiniZinc models in tree-shaped data structures. A model in IR can be constructed programmatically using a well-defined API. The library provides modules for static analysis (such as type checking). In addition to the in-memory data structures, `libmzn` also defines a persistent, on-disk format for the IR and methods for storing and loading models in IR.

Frontends. The MiniZinc frontend creates IR from source files. Other frontends should include a convenient interface for direct modeling in C++, Java, and Python, as well as possibly an XML parser.

Translation and optimization. This phase not only performs the usual optimizations (such as constant propagation or common subexpression elimination), but also *flattens* the IR so that it is suitable for the solver backends.

Solver backends. There are two possible designs for the solver API. The first approach would pass a model in IR to the solver, which would then walk the IR tree, build its internal model, and solve the problem. The second approach would use call-backs into the solver to build the solver-internal model. These two approaches correspond roughly to DOM-based ([1], *Document Object Model*) and SAX-based ([11], *Simple API for XML*) XML parsing, respectively.

Solution access. LLVM can produce code that adheres to the C calling convention, so that the main application can simply invoke the result of compilation. In `libmzn`, this corresponds to the solvers communicating the solutions back to the application through `libmzn`. The application can access values of decision variables directly through the IR of the model.

Interfacing. C++ may not be the ideal language for implementing program transformations. For experiments in particular, it would be useful to have interfaces to `libmzn` from mainstream functional programming languages such as OCaml or F#. For the core functionality, however, C++ is a good choice because of its role as a lowest common denominator.

What's new? Many of the above components are also present in the `mzn2fzn` tool of the MiniZinc distribution. For instance, the MiniZinc code is parsed into an IR implemented as Mercury data types. The difference is that `libmzn` would *expose* these components and allow the user to extend and combine them, while `mzn2fzn` *hides* them.

Avoiding brand dilution. Making `libmzn` extensible entails the risk of fragmentation into several slightly incompatible systems – one per backend solver in the extreme case. Obviously, this would not at all help establish MiniZinc as a standard. An important task is therefore to make the *modeling API* attractive and versatile enough to be accepted as a standard, much like the MiniZinc language. Different but API-compatible `libmzn` derivatives would be less than ideal, but not detrimental to the overall project goal.

4.2 Beyond `mzn2fzn`

The increased modularity and open APIs of `libmzn` would have a number of key benefits compared to the current MiniZinc toolchain.

More frontends. When the core of MiniZinc is available as a library, it is easy to add frontends for different concrete modeling languages such as Essence [2] or XCSP [10], or add more features from full Zinc [8].

Language integration. A huge potential for a `libmzn` would be the integration into general-purpose or scripting languages. For example, a thin modeling layer in C++, Java, or Python could translate directly into `libmzn` IR, and take advantage of the rich underlying compilation and execution engine. This would make projects such as Numberjack [3] very easy to implement, and immediately give access to all the solver backends available for `libmzn`. With a common IR, it is even possible to mix languages. The library of global constraint decompositions can still be defined in MiniZinc, while the main application accesses `libmzn` directly through a C++ modeling layer.

Solver-specific optimizations. Some solvers would benefit from special treatment of variable types (e.g., deciding whether to implement certain variables as Boolean or 0/1 integer variables), native handling of complex disjunctions (such as in [4] or [6]), or direct support for multi-dimensional arrays. Currently, solvers have to rediscover these potential optimizations in the FlatZinc code. In a modular system, these solver adaptations could be implemented as special compilation passes.

Pre-compiled models and libraries. A persistent IR would enable the precompilation of constraint libraries and even whole models. Although a full separate compilation is not desirable as many optimizations require the model parameters to be known, partial precompilation could at least perform all the parsing and type checking. This would speed up repeatedly solving the same model with different parameters.

4.3 A `libmzn` Prototype

A prototype implementation of `libmzn` is available at

`http://people.cs.kuleuven.be/~guido.tack/libmzn.php`

It currently comprises a set of C++ data structures implementing a simple version of a MiniZinc intermediate representation, as well as a MiniZinc parser based on flex/bison, and a type-inst checker. The plan is to extend the prototype into a working replacement for `libmzn` (i.e., work with text-based input and output), while already designing the interfaces to support direct APIs later.

5 Related Work

There are a number of related projects and products.

IBM ILOG Concert Technology is the C++ modeling layer for IBM ILOG Solver, CPLEX, and CP Optimizer. It is solver-independent in the sense that solver-specific models can be *extracted* from a Concert model and then solved with the different backends that ILOG provides. It does not provide a full modeling language, though, but relies on the usual abstraction mechanisms like functions and objects available in C++.

Numberjack is a Python library for constraint modeling [3]. It adopted MiniZinc's approach to decompose global constraints if the target solver does not support them natively. The library provides an *embedded domain specific language*, making use of Python's abstraction, reflection, and overloading mechanisms.

JSR 331 , the Java Standardization Request for a Constraint Programming API [5], aims at defining a standard, solver-independent API for constraint modeling in Java. The API is close to the solver interface, it does not provide any complex modeling constructs that would require compilation techniques to implement.

6 Conclusions

This position paper proposes to develop `libmzn`, a CP compiler infrastructure based on MiniZinc and inspired by the LLVM framework.

On the frontend side, `libmzn` provides a clean API for constructing MiniZinc models. The API can be used directly for interfacing to applications, or to build higher-level interfaces, for example in languages such as Java, C++ or Python. The backend API of `libmzn` makes it easy to add solvers to the MiniZinc toolchain without implementing a FlatZinc parser. A MiniZinc parser and a FlatZinc text-based interface ensure backwards compatibility of `libmzn` with the current toolchain.

Acknowledgements. I am very grateful to Håkan Kjellerstrand, who tested the prototype on his huge library of MiniZinc models, which helped to find and fix many bugs in the parser and type checker. I also thank the anonymous reviewers for raising some interesting questions.

References

1. DOM, the XML Document Object Model. http://www.w3.org/DOM/ (2011)
2. Frisch, A.M., Harvey, W., Jefferson, C., Hernández, B.M., Miguel, I.: Essence : A constraint language for specifying combinatorial problems. Constraints 13(3), 268–306 (2008)
3. Hebrard, E., O'Mahony, E., O'Sullivan, B.: Constraint programming and combinatorial optimisation in Numberjack. In: Lodi, A., Milano, M., Toth, P. (eds.) CPAIOR. Lecture Notes in Computer Science, vol. 6140, pp. 181–185. Springer (2010)
4. Jefferson, C., Moore, N.C.A., Nightingale, P., Petrie, K.E.: Implementing logical connectives in constraint programming. Artif. Intell. 174(16-17), 1407–1429 (2010)
5. JSR 331, Constraint Programming API. http://www.jcp.org/en/jsr/detail?id=331 (2011)
6. Lagerkvist, M.Z., Schulte, C.: Propagator groups. In: Gent, I.P. (ed.) CP. Lecture Notes in Computer Science, vol. 5732, pp. 524–538. Springer (2009)
7. LLVM, low-level virtual machine. http://www.llvm.org (2011)
8. Marriott, K., Nethercote, N., Rafeh, R., Stuckey, P.J., de la Banda, M.G., Wallace, M.: The design of the Zinc modelling language. Constraints 13(3), 229–267 (2008)
9. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: Bessière, C. (ed.) CP. LNCS, vol. 4741, pp. 529–543. Springer (2007)
10. Roussel, O., Lecoutre, C.: XML representation of constraint networks: Format XCSP 2.1. CoRR abs/0902.2362 (2009)
11. SAX, the simple API for XML. http://www.saxproject.org/ (2011)