Minimal logic detection and exporting SMT-LIB problems with Dolmen

Guillaume Bury[1]

[1]*OCamlPro SAS, 21 rue de Châtillon, 75014 Paris, France*

Abstract
provides tools to parse, type, and validate input files used in automated deduction, and in particular problems from the SMT-LIB. This work adds to Dolmen the ability to export in SMT-LIB format any problem that it successfully parsed and typed. More than just printing terms, this work includes the ability for Dolmen to compute the minimal SMT-LIB logic needed for a set of statements, so that problems from other languages can be given an adequate logic when being exported to SMT-LIB format. This also means that Dolmen can now be used to compute the minimal logic of an arbitrary SMT-LIB problem.

1. Introduction

Dolmen [1, 2] is a project providing parsers and a typechecker for some of the most commonly used languages in automated deduction, such as SMT-LIB [3] and TPTP[4] [1]. Dolmen is used as a validator for the submission of new benchmarks to the SMT-LIB, where it checks that new problems conform to the SMT-LIB specification. While this guarantees that new problems are conformant, problems already in the SMT-LIB do not have such guarantee, and it has been shown in [1] that there is a small number of problems in the SMT-LIB that are not conformant. Furthermore, manual analysis of problems in the SMT-LIB has revealed that some were misclassified: the logic used by such problems, while correct, was not the minimal logic in which the problems belonged, potentially hampering the efforts of solvers on these, as well as making benchmarks harder to correctly interpret.

The work presented in this extended abstract is a first step towards building a tool that would help solve these problems, by adding to Dolmen the ability to export to the SMT-LIB format any problem that it parsed and typed successfully. In this case, exporting means more than simply printing a problem in SMT-LIB format. Rather, it means considering a problem made up of statements over arbitrary typed terms, and generating an SMT-LIB problem that is equivalent. One major aspect of this generation is determining an adequate logic, because the input problem may not have an explicit logic (this is the case for languages accepted by Dolmen other than SMT-LIB, such as TPTP, Zipperposition's format[5] and Alt-Ergo's[6] native format, called æ) or, in the case of an input problem in the SMT-LIB language, because the original logic is incorrect or not precise enough. This work actually discovered an issue with the current specification of the SMT-LIB which is that there are problems that do not have a single minimal logic. This will be developped in Section 4.

Another non-trivial aspect of exporting problems to the SMT-LIB format is to respect the syntax and typing conventions of the SMT-LIB language. Indeed, the export feature of Dolmen is not meant only as a formatting tool. Instead, it can already translate problems from other languages to SMT-LIB, and these other languages often have different syntactic conventions than the SMT-LIB language, for instance with respect to variable names, and shadowing. In addition to this translation feature, it is planned to add to Dolmen the ability to transform and rewrite terms before exporting. In such a situation, it

SMT Workshop, July 22–23, 2024

guillaume.bury@gmail.com (G. Bury)

https://gbury.eu (G. Bury)

[1] As well as Dimacs, iCNF, and Zipperposition and Alt-Ergo's native formats

would be problematic to try and enforce that each of these transformations/rewrites respect some target language's conventions: what happens if some transformation happens to be used for multiple target languages with different (or even incompatible) conventions ? Instead Dolmen has its own conventions on how it represents terms and the exporting mechanism has the duty of taking as input terms that respect Dolmen's conventions, and generating (or rather printing in this case) terms that respect the target language's conventions, while preserving satisfiability.

Dolmen is written in OCaml[7] and is available at https://github.com/Gbury/dolmen, under a BSD2 license. Recent releases include pre-built binaries for Linux, MacOs and Windows, for ease of use.

2. The export functionality in practice

2.1. Basic usage and logic detection

The export functionality of Dolmen is easily used with its binary, by specifying on the command line an input problem, followed by an output filename, as shown in Figure 1. Dolmen auto-detects both the input and output language in that case, based on the filenames. In order to explicitly specify the output language, one can use the `-o lang` option, which combined with no output file on the command line will print the result on the standard output.

```
1 ; input.smt2
2 (set-logic ALL)
3 (declare-const a Int)
4 (declare-fun b () Int)
5 (assert (<= (- a b) 0))
6 (check-sat)
```

```
# dolmen input.smt2 output.smt2
```

```
1 ; output.smt2
2 (set-logic QF_IDL)
3 (declare-const a Int)
4 (declare-const b Int)
5 (assert (<= (- a b) 0))
6 (check-sat)
7 (exit)
```

Figure 1: Example of using Dolmen to export an SMT-LIB problem

In the example in Figure 1, one can notice a few changes that Dolmen made to the problem:

- the `declare-fun` on line 3 has been changed to a `declare-const`
- an `(exit)` statement has been added at the end
- lastly, Dolmen has automatically detected that there were no quantifiers, and that only integer difference logic was used, resulting in it changing the logic to `QF_IDL`.

2.2. Translating a problem from another language

Another interesting example is what happens when one tries to translate a problem from another language to SMT-LIB, as shown in Figure 2. In this example, the input problem is in Alt-Ergo's native language[6], which was designed to be very close to OCaml, and in typical ML fashion, it defines the type of polymorphic lists and the length function over these lists[2]. That problem then defines two goals, which are formulas that must each be proved separately, using the context preceding them.

```
; input.ae
type 'a list =
    Nil
  | Cons of { head : 'a; tail : 'a list}

function length (l : int list) : int =
  match l with
  | Nil -> 0
  | Cons(_, l') -> length(l') + 1
end

goal g1: (exists l : 'a. (length(l) <= 4))

goal g2: (length(Nil) = 0)
```

```
# dolmen input.ae output.smt2
```

```
; output.smt2
(set-logic UFDTLIA)
(declare-datatype list (par (|'a|)
  ((Nil) (Cons (head |'a|) (tail (list |'a|))))))
(define-fun-rec length ((l (list Int))) Int
  (match l ((Nil 0) ((Cons |_| |l'|) (+ (length |l'|) 1)))))
(push 1)
(assert (not (exists ((l (list Int))) (<= (length l) 4))))
(check-sat)
(pop 1)
(push 1)
(assert (not (= (length (as Nil (list Int))) 0)))
(check-sat)
(pop 1)
(exit)
```

Figure 2: Example of using Dolmen to translate a problem in Alt-Ergo'a native language

In that example in Figure 2, there are quite a few interesting transformations that have been done by Dolmen :

- A suitable logic, `UFDTLIA` has been automatically chosen for the problem[3]. Similarly, the type variable used in the datatype definition has been quoted: indeed Alt-Ergo's native language encourages uses of variables starting with an apostrophe, which are interpreted as type variables. However in SMT-LIB, type variables are not allowed to start with an apostrophe, except if they are quoted[4].
- In the `length` function's pattern matching, the ML-style wildcard used in the `Cons(_, l')` constructor has also been quoted as required, since _ is a reserved word in SMT-LIB.

2

The `length` in this example is intentionally restricted to integer lists because although SMT-LIB allows polymorphic datatypes, it does not allow polymorphic function definitions.

3

In this case, the choice of `LIA` over `IDL` is slightly arbitrary: in the presence of the integer types and literals, but without any arithmetic operator, it is currently the default choice, but that may change in the future.

4

SMT-LIB identifiers can be quoted that is, put between pipes e.g.,`|a quoted identifier|`.

- Each goal has been translated to a series of `push`, `assert`, `check-sat`, `pop`, to respect Alt-Ergo's language semantics.
- In the first goal, the type of the quantified variable has been inferred to be that of a list of integers
- In the second goal, the use of the `Nil` polymorphics nullary constructor has been disambiguated using the `as` annotation, as required by the SMT-LIB specification.

Besides showing the results, these two examples have shown some instances of the normalization process used by Dolmen to make logic detection and language translation easier and scalable. The next section will explain why that normalization is useful, and how it is done.

3. Exporting from Typed Terms

3.1. Context

One of the goals of exporting problems in SMT-LIB format in Dolmen is to allow translation of problems from one language into another. The common issue with such endeavor is that the naive approach of writing translations and/or printers from each input language to each output language results in a number of translation/printing function that is quadratic in the number of languages supported, assuming all input language is also an output language, which is the case here. Since Dolmen supports multiple input languages, one would have to write a dedicated translation/printer to SMT-LIB for each input language to account for small differences in semantics of each language. To address this issue, Dolmen uses a common representation into which every input problem is converted, with defined semantics. That representation is then exported to the desired language. Instead of a quadratic number of translation functions (from each input language to each output language), this only requires a linear number of translation functions: one for conversion from each input language to this common representation, and then one export function for each output language. In Dolmen, that common representation is simply called the typed terms.

3.2. Parsed and Typed Terms in Dolmen

Dolmen distinguishes two kinds of terms: the parsed terms and the typed terms. This distinction comes from the architecture of Dolmen which separates parsing from type-checking. Parsers in Dolmen generate parsed terms, which are meant to simply be a structured representation of the input: terms are represented as trees instead of strings. This means that the semantics of a parsed term (or even if it has defined semantics) depends on the input language considered. For instance :

- The parsed term corresponding to (+ 1 "foo") can not be given semantics, at least not in any language currently accepted by Dolmen.
- The parsed term corresponding to (and a b c) can be given some semantics in SMT-LIB where it is actually syntactic sugar for (and (and a b) c), which can be given its usual semantics. However, other languages may simply prohibit such uses of conjunction, only allowing it to be applied to two arguments.
- Similarly, the parsed term corresponding to (+ 1 2.0) can be given a semantic in SMT-LIB where it is actually syntaxic sugar for (+ (to_real 1) 2.0), whereas it is a typing error in TPTP.

That process of assigning semantics to terms that have one[5] is actually done during type-checking: parsed terms are inspected, and those that match the typing rules set by each language are accepted and transformed into typed terms, where they are given semantics independent of any context or language. For instance :

- Syntactic sugar is expanded, and every operation made explicit e.g., associativity, chainability, implicit to_real operations in SMT-LIB arithmetic, …

5

One example of terms that do not have semantics is terms that do not type-check.

- Symbols that may be overloaded in input languages are disambiguated e.g., division can sometimes share the same symbol for integers and reals in inputs, but after typechecking, there are different symbols for euclidean integer division, and real division.
- Some builtin symbols may have different arguments (or order of arguments) depending on the languages; in such cases, one order is chosen as the canonical one for the typed terms.

This can be seen as some kind of normalization step, and it provides useful guarantees. Most notably, it means that Dolmen is easy to use as a frontend for solvers written in OCaml[6]: solvers only have to implement a translation function from Dolmen's typed terms to their own internal terms. That translation function is independent of the input language, and only has to translate the terms according to the semantics given by the documentation in Dolmen. Exporting to SMT-LIB is then simply another application of that idea: instead of translating typed terms to a solver's internal terms, one simply generates strings according to the SMT-LIB language, without worrying about the input language.

A slight limitation of that approach is that, while one does not need to consider the input language in order to manipulate typed terms, it does not mean that all typed terms can be directly exported in any language as they are. Indeed, Dolmen's typed terms can express all of the problems from multiple input languages: in that sense, the semantics of these typed terms are a superset of each language's semantics. In particular, it can happen that a language has more builtin symbols than another, and it is not obvious how to export a typed term that contains a builtin not available in a given language. Fortunately, in the case of the SMT-LIB, it is the language with the most builtins, and it is almost a superset of every other language supported by Dolmen currently. The only exception is the exponentiation symbol that is present in Alt-Ergo's native language, but is not present in SMT-LIB's theory of integers.

3.3. Printing Typed Terms

Identifiers, shadowing and renaming Given typed terms, the task of exporting them to SMT-LIB format is now reduced to printing, although some care needs to be taken to respect the SMT-LIB's syntax conventions. The main limitation is that of naming of identifiers, such as quantified variables, function symbols, etc…Indeed, all languages have restrictions on what names can be used, but there are no such limitations in typed terms, where names are only used for pretty printing[7]. Therefore names coming from the typed terms need to be checked against the target language's syntax rules, which can result in any of these outcomes :

- The name is printable as is
- The name can be printed if it is quoted
- The name cannot be printed at all, and must be renamed to another name that can be printed.

Additionally, typed terms may use the same name multiple times in the same scope. This is not a problem for most term manipulations, but it becomes problematic when printing terms. This means that in some cases it may also be necessary to rename a variable or constant that could have been printed, but whose name is already used by another variable or constant in scope.

To address these issues, during printing, Dolmen maintains a bijection between the typed variables and constants in scope, and their printed names (i.e. their name after any potential renaming), to ensure that all variables and constants in scope can be unambiguously printed.

View While this paper mainly shows examples using the Dolmen binary, Dolmen also provides all of its features as OCaml libraries, which aim at being as versatile as possible. To follow that objective, the SMT-LIB printer in Dolmen (as well as all other printers that will be added in the future), do not operate directly on the typed terms of Dolmen but instead on a view of these terms. This means that a user does

[6] Dolmen is currently used as frontend by at least two solvers: Alt-Ergo and COLIBRI[8].

[7] For comparison, unique integers assigned at creation time are used.

not need to use dolmen's typed terms to make use of the printers, but only needs to provide a `view` function, as well as some other small functions, for intance to compute equality of function symbols. A simplified version of the interface that a `view` function needs to implement is presented in Figure 3.

```
type _ term_view =
  | Var : 'term_var ->
    < term_var : 'term_var; .. > view
  (** Term Variables*)
  | App : 'term_cst * 'ty list * 'term list ->
    < ty : 'ty; term : 'term; term_cst : 'term_cst; builtin : 'blt; .. > view
  (** Polymorphic application of a function, with explicit type arguments *)
  | Match : 'term * ('t pattern * 'term) list ->
     (<term_var: 'term_var; term_cst: 'term_cst; term: 'term; ..> as 't) view
  (** Pattern matching. *)
  | Binder :
    <ty_var: 'ty_var; term_var: 'term_var; term: 'term; ..> binder * 'term ->
    <ty_var: 'ty_var; term_var: 'term_var; term: 'term; ..> view
  (** Binders over a body term. *)
(** View of terms in first-order. Defined by the Dolmen library. *)

(** The module type that need to be implemented by users to use printing. *)
module type View = sig
  type ty_var
  type ty
  type term_var
  type term_cst
  type term
  (** User-defined types *)

  val term_view : term ->
    < ty_var: ty_var; term_var: term_var; term_cst : term_cst;
      ty : ty; term: term; > view
  (** User-defined view on terms. *)
end
```

Figure 3: Simplified version of the View signature for printing

Undoing Syntactic Sugar Expansion Lastly, one feature of the SMT-LIB printer in Dolmen is the ability to undo the syntactic sugar expansion that was done during type-checking. Indeed, SMT-LIB defines a few expressions as being syntactic sugar for more complex expressions. Instances of syntactic sugar are expanded by Dolmen during type-checking. While it is correct to print the typed terms after expansion of the syntax sugar, it is useful to try and "undo" the expansion while printing. Indeed, it improves readability of generated problems, and it also results in smaller problems being generated, which is always useful. Therefore, Dolmen tries to "undo" such syntax sugar expansion.

Currently, Dolmen does recognize and "undo" the expansion of the following syntactic sugars :

- Type aliases can be defined using the `define-sort` statement. Such aliases are to be substituted/expanded during type-checking.
- In arithmetic using both integers and reals, most (e.g. addition) can take arguments which are either all integers, or all reals. However, if they are applied to one integer argument and a real argument, then it is considered syntactic sugar for an expanded term where the `to_real` function has been applied to the integer argument.
- A few operators/functions in the SMT-LIB are defined as being left associative, right associative, or chainable, which are all syntactic sugar to allow the application of binary functions to an

arbitrary number of arguments, by considering it syntactic sugar. As mentioned previously, an instance of that is the and function, which is left associative, meaning that (and a b c) is allowed (even though and is a binary function), and is actually syntactic sugar for (and (and a b) c).

Most of that work is done during printing. First, nested applications of associative and chainable functions are detected and the n-ary application is printed instead of the expanded form. Separately, the printer maintains a state to keep track of whether the term being printed is directly under an arithmetic operator, in which case the application of the `to_real` can be omitted.

Finally, Dolmen only expands type aliases as needed during type-checking applications, and even this expansion occurs on copies of the types that are used specifically to check that typing rules are respected. This means that the original type (before alias expansion) is kept in the typed terms, which enables Dolmen to print the alias when exporting to the SMT-LIB format.

4. Minimal Logic in the SMT-LIB

4.1. Minimal Logic detection

The previous section has detailed how Dolmen handles normalisation of input problems and printing to the SMT-LIB format. This essentially is enough to use Dolmen as a formatter for SMT-LIB problems. However, when translating problems from other languages, a logic has to be chosen for the resulting SMT-LIB problem, since most languages do not have a notion of logic that can be set for each problem. For individual problems of small to medium size, it might be possible to choose a logic manually, and for some languages, it might also be possible to choose a single logic for all problems in that language: most of the languages other than SMT-LIB have very few builtin theories, and could actually be given the UFDTNIRA logic. However, this is not a very satisfying solution. Indeed, using this logic may induce solvers to use expensive algorithms needed to handle complex theories such as non-linear arithmetic, instead of using more optimized algorithm that only works on linear arithmetic or difference logic. Furthermore, always using the same over-estimated logics would also make it harder to compare provers' reasoning capabilities on specific theories, as well as potentially exclude some provers in competitions, since provers are sometimes registered only for some specific and targeted logics.

Additionally, existing problems in the SMT-LIB library could also benefit from a detection of the minimal logic of a given problem: indeed previous manual analysis proved that some problems could be given a smaller logic than they had been given. All of this motivated the work to add detection of the minimal logic of a problem to Dolmen in the course of this work.

Most of the work for determining the minimal logic is straightforward: for theories such as FP, BV, S, DT, it is simple to iterate over all the function symbols used and identify to which theory they belong to. A minor level of complexity is introduced due to the overlap between the FP and BV theory, where problems in FP are allowed to use bitvector literals, but that is easily solved by distinguishing uses of builtin function symbols of BV from uses of bitvector literals. Similarly, detecting the presence of quantifiers in terms is easy.

The most complex part of finding the minimal logic is arithmetic, which is divided in multiple fragments. The number of fragments is a first source of complexity. There are two axis of fragmentation: arithmetic can use integers, reals or both, and arithmetic can be non-linear, linear, or part of difference logic.

However, the most problematic part is that there are actually much more than the 6 combinations one could think the above two axis provide. Indeed, while integer arithmetic, real arithmetic, and combined arithmetic are defined as theories, this is not the case for linear arithmetic and difference logic. Instead, each logic that uses linear arithmetic (or difference logic) actually defines its notion of linearity (resp. difference logic) in the logic description. This description sometimes refers to the description of another logic to avoid redundancy, but even then, there are two distinct variant of linear arithmetic, and three variant of difference logic. This significantly complexifies the task of finding the minimal logic.

Fortunately, Dolmen already has code for checking and enforcing each variant during type-checking. This code proved to be a good guide in how to detect and track each variant of arithmetic.

Dolmen is now able to detect the use of each theory and variant of arithmetic present in any given problem. However, this brought an interesting problem to light: it can happen that some problems do not a minimal logic; these problems can actually belong to many different logics, but not one is truly minimal, and this is because of the fragmentation of the definitions of linearity in arithmetic.

4.2. Problems with no minimal logic

There are actually two distinct definitions of linear arithmetic that can be found in logic descriptions. In both cases, the complexity of the restriction revolves around uses of multiplication, where terms of the form `(* x c)` and `(* c x)` are only allowed under some conditions :

- Most logic[8] requires that ·
  - `x` is a free constant
  - `c` is an integer or rational coefficient
- However, the logics AUFLIA and QF_AUFLIA instead require :
  - `x` is a free constant or a term with top symbol not in `Ints`
  - `c` is a term of the form `n` or `(- n)` for some numeral `n`

First thing of note is that, according to these definitions, `(* 2 3)` is not a valid term in linear arithmetic, since neither `2`, nor `3` is a free constant or a term with top symbol not in `Ints`. Second, and more important, is that, assuming a function `f` from integers to integers, the term `(* 2 (f x))` is accepted in AUFLIA but not in UFLIA, because only the former allows one side of the multiplication to be a term with top symbol not in `Ints`. This is an actual and real problem for the existence of a minimal logic. Indeed consider the following problem :

```
(set-logic ALL)
(declare-const x Int)
(declare-fun f (Int) Int)
(assert (= 0 (* 2 (f x))))
(check-sat)
(exit)
```

One would naturally give this problem the logic QF_UFLIA, since there are no quantifiers, at least one uninterpreted function, and some arithmetic which could be seen as linear. However, It does not belong to that logic, at least according to the current specification (as explained above). The problem is then to chose which logic to give to that problem: as show previously, the QF_AUFLIA logic would fit, but it seems unfortunate that this logic mentions the Array theory, which is not used in our problem. On the other side we could use a non-linear logic, but that would also be less than ideal. Essentially, this is a case where it appears that there is no one minimal logic for this problem, because of somewhat arbitrary restriction imposed by the SMT-LIB specification, and instead in the current state, there could be multiple logics adequate for this problem, but not one truly minimal among them.

Currently, Dolmen will produce an error for cases such as these, because no truly satisfying solution was yet found on how to solve the issue. It would be a very interesting subject of discussion with the community to see what solution can be found, both for the current version of SMT-LIB, but also for the upcoming version 3, which could be an opportunity to simplify the specification and avoid the very problematic fragmentation that the current specification creates, particularly around linearity.

8 See for instance the specification of QF_LIA [9]

5. Conclusion

Dolmen is now able to both translate problems from other languages into problems in the SMT-LIB format, as well as reformat and rewrite existing problems in order to minimize their logic. That work is still at the prototype stage, in particular it has not yet been tested at a large scale, and the errors produced are also currently not user friendly. That being said, it has reached a state where it can be used, and members of the community are encouraged to use it and report potential bugs.

These new features of Dolmen should prove useful to the community, both those managing the SMT-LIB, but also those proposing new problems to the SMT-LIB benchmark library (who could use it to correctly classify their problems without too much work). Lastly, this work could also potentially open new possibilities for the upcoming version 3 of the SMT-LIB language: with a translator being available to translate problems from version 2 to version 3[9], it might be reasonable to break backwards compatibility, since there would be an easy way to convert a problem from one version to another.

References

[1] G. Bury, Dolmen: A validator for smt-lib and much more., in: SMT, 2021, pp. 32–39.
[2] G. Bury, F. Bobot, Verifying models with dolmen, in: SMT, 2023, pp. 62–70.
[3] C. Barrett, A. Stump, C. Tinelli, The SMT-LIB Standard: Version 2.0, in: A. Gupta, D. Kroening (Eds.), Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK), 2010.
[4] G. Sutcliffe, The Logic Languages of the TPTP World, Logic Journal of the IGPL (2022). doi:10.1093/jigpal/jzac068.
[5] S. Cruanes, Extending superposition with integer arithmetic, structural induction, and beyond, Ph.D. thesis, École polytechnique, 2015.
[6] The Alt-Ergo theorem prover, https://https://alt-ergo.ocamlpro.com/, 2024.
[7] The OCaml Official website, http://https://ocaml.org/, 2024.
[8] B. Marre, F. Bobot, Z. Chihani, Real behavior of floating point numbers, in: SMT Workshop, 2017.
[9] Official Specification of the QF_LIA logic of SMTLIB, http://smtlib.cs.uiowa.edu/logics-all.shtml#QF_LIA, 2024.

[9]

Although Dolmen does not yet have a printer for SMT-LIB vesion 3, it is the author's intention to implement one as soon as SMT-LIB version 3 is released.