Dolmen: A Validator for SMT-LIB and Much More

Guillaume Bury[1]

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

Abstract

Dolmen provides tools to parse, type, and validate input files used in automated deduction, such as problems written in the SMT-LIB language, but also other languages commonly used in theorem provers, such as TPTP. Dolmen is split into three parts: a command-line binary, an LSP server, and an OCaml API. The command line binary can not only validate files, but more importantly is built so that it can provide informative error messages when input files do not respect their language specification. These capabilities are also provided as an LSP server, which can connect to all current editors to provide instantaneous feedback when editing files. Lastly, Dolmen also provides a flexible API in OCaml so that new projects in the field of automated deduction do not need to re-implement the parsing and typing of input files.

Keywords

parsing, typechecking, conformance checker, SMT-LIB

1. Introduction

There is a difference between the set of files that respect a language specification and the set of files that an automated deduction tool accepts as input. This is in part due to an inherent conflict between, on the one hand, the desirable simplicity of a language specification and, on the other hand, the interest of tools to cover the biggest set of problems that they can actually process. A good example of this disparity is often linear arithmetic: theorem provers (and other tools) will typically accept any arithmetic expression that can be reduced to a linear expression, such as $2 * (a + b + 1)$. On the other side, languages need to specify what expressions can be considered linear, and more often than not that specification comes as a syntactic criterion, which is simple to express, but is more restrictive than what one expects. For instance, the earlier expression: $2 * (a + b + 1)$, is not technically a linear expression in the QF_LIA logic of SMT-LIB [1, 2], but is accepted by CVC4 [3]. This is not necessarily an indication that there is anything wrong with either the specification, or any theorem prover: there are very good reasons for the specification to be a syntactic criterion, most notably in order to have a concise specification, and to be reasonably verifiable; and there are very good reasons why theorem provers would do everything possible to be able to prove as many problems as possible. A more realistic and problematic example would be a problem (still in linear arithmetic), where the expressions $a * (b + 3)$ appears, but where a can be statically known to always be a constant, for instance because it is defined as a constant, or simply let-bound to a particularly big or

SMT Workshop, July 18–19, 2021
guillaume.bury@gmail.com (G. Bury)
https://gbury.eu (G. Bury)

significant constant which would not have been convenient to repeat in an input problem[1]. In such a case, it is good for a prover or tool to be able to determine the expression to actually be linear, and to try and solve the problem, which is easier if the expressions can be determined to be linear (compared to general arithmetic).

Therefore, there exists a gap between a language specification, and what is accepted by tools, and that gap might be a problem for new tools that first target a logic as specified by one of the standard languages used in automated deduction, but might then be confronted with a problem because they do not accept problems accepted by other tools. This underlines two problems, the first being that some tools actually accept an extension of the language, a fact which is not always evident to end-users, but more importantly, these extensions have, more often than not, no specification, which makes it a headache to try and accept what other tools actually accept as inputs. In the case of SMT provers and the SMT-LIB, one of the reasons why it has become so commonplace for most tools to use language extensions is that there were no official way to ascertain whether a problem file conformed to the SMT-LIB specification [4] or not, leaving the task of validating files to the provers, which have no real reason to limit themselves to what the SMT-LIB specifies. That is the problem that Dolmen aims to solve, by providing a way to validate files for the SMT-LIB, *i.e.* to distinguish problems that conform to a given specification from problems that do not. Additionally, for input files that do not respect the specification, the goal is to provide as much information as possible concerning the violation of the specification, so that the file can be fixed, or at least to give a legible explanation of why the file could not be validated.

2. Dolmen

The Dolmen project is available at https://github.com/Gbury/dolmen, under a BSD2 license. It is written in OCaml and can be installed via Opam, the official OCaml package manager. Binary releases are also provided on the release page of the repository. Dolmen provides the ability to validate input files for some of the most commonly used languages in automated deduction, including SMT-LIB and TPTP. This functionality of validating input files is offered by Dolmen in three different ways :

- A command-line binary (2.1), named `dolmen`. This binary can be invoked in a terminal to validate an input file.
- An LSP server (2.2), named `dolmenls`. This allows users to have access to instant feedback while editing files in any compatible text editor or IDE.
- An OCaml API (2.3). The API offers not only to validate input files, but actually returns an abstract syntax tree for the validated file, so that projects that use the Dolmen API need not implement any parser or type-checker.

Even though Dolmen, or more specifically its API, is used, in a few projects[2], it is also an independent and standalone project. That status as an independent project is a reason why

[1]Such a let-binding can also be thought of as a precaution when writing a problem, in order to prevent typing errors and ensure that the same constant is used throughout a problem

[2]Current projects that use Dolmen include: the Alt-Ergo [5] SMT solver whose frontend is being rewritten to use Dolmen, the ArchSAT [6] SMT prover, and the COLIBRI [7] CP solver.

```
(set-logic ALL)
(assert (as (blah) bar))
```

```
# dolmen parsing_error.smt2
File "./parsing_error.smt2", line 2, character 13-17:
Error while parsing an identifier, read the symbol 'blah',
  but expected an underscore: identifiers starting with an opening
  parenthesis must be indexed identifiers, of the form "(_ symbol
  index+)".
```

Figure 1: Dolmen parsing error message example

Dolmen can avoid the temptation of trying to accept as many inputs as possible. Instead, Dolmen strives to accept only files that conform exactly to the official specification of each language. As far as the author knows, that makes Dolmen the first project to provide a checker for strict conformance with the SMT-LIB specification[3].

2.1. The Dolmen command line binary

The Dolmen command line binary is called `dolmen`. It is invoked as `dolmen myfile.ext` to validate `myfile.ext`. There are additional options available on the command line to fine tune some aspects of the execution, such as limits for the time and memory used by `dolmen`, all of which are documented in the man page and via the `--help` option.

As mentioned above, when writing Dolmen, a lot of care went into ensuring that it produced useful error messages when the input file did not meet the specification. Two examples of such error messages are provided in Figures 1 and 2.

Typing errors Figure 2 shows how Dolmen reacts to the linear arithmetic example mentioned in the introduction. In this instance, the `QF_LIA` logic is used. According to the specification of this logic [2], the use of the multiplication symbol `*` is forbidden, except as specified by the following:

> Terms containing * with *concrete* coefficients are also allowed, that is, terms of the form c, (* c x), or (* x c) where x is a free constant and c is a term of the form n or (- n) for some numeral n.

In the expression `(* 2 (+ a b))`, we have on one side the numeral 2, and on the other side the expression `(+ a b)` which is not a 'free constant', and this is why an error is raised by Dolmen. Additionally, and more surprisingly it can be remarked that expressions such as

[3]The jSMTLIb [8] project seems to also have targeted that goal, although the current state of the project as well as whether it supports version 2.6 is unclear.

```
(set-logic QF_LIA)
(declare-fun a () Int)
(declare-fun b () Int)
(assert (= 0 (* 2 (+ a b))))
(check-sat)
```

```
# dolmen typing_error.smt2
File "./typing_error.smt2", line 4, character 13-26:
Error This is a non-linear expression according to the smtlib spec.
      Hint: multiplication in strict linear arithmetic expects an integer or
        rational literal and a symbol (variable or constant) but was given:
        - an integer coefficient
        - a complex arithmetic expression
```

Figure 2: Dolmen typing error message example

```
(set-logic QF_LIA)
(declare-fun a () Int)
(assert (let ((a 0)) (= a a)))
(check-sat)
```

```
# dolmen warning.smt2
File "./warning.smt2", line 3, character 15-16:
Warning Shadowing: ‘a‘ was already declared at line 2, character 0-22
```

Figure 3: Dolmen warning message example

`(* 2 3)` are not allowed by the specification above. It is for such corner cases that Dolmen can be especially useful by giving detailed error messages.

Other errors and warnings In addition to errors for non-conforming files, Dolmen also raises warnings to alert users to potentially suspicious situations. For instance, Dolmen will warn in cases where there are bound variables that are unused, or when constants are shadowed. For instance, in the case of SMT-LIB, shadowing of identifiers can be allowed, or forbidden depending on the circumstances: bound variables are forbidden to shadow constants from builtin theories, whereas constants are forbidden from shadowing other constants, whether they come from builtin theories or were declared earlier in the file[4]. Dolmen implements these rules and will raise a proper error when these rules are not respected. However a bound variable is allowed to shadow a declared constant in SMT-LIB, in which case `dolmen` will by default

[4]See page 32 of the SMT-LIB specification [4]

raise a warning[5], as shown by Figure 3.

2.2. Dolmen's LSP server

In addition to the command-line binary, Dolmen also provides an LSP server. LSP [9], aka language server protocol, is an open, JSON-RPC-based protocol for use between source code editors or integrated development environments (IDEs) and servers that provide programming language-specific features. The goal of the protocol is to allow programming language support to be implemented and distributed independently of any given editor or IDE. The language server protocol is typically used by editors by connecting to a server that will handle requests and provide diagnostics (*i.e.* warnings and error messages) to the editor. LSP standardizes the interaction between the editors and the language-specific checkers, eliminating the need to implement a checker for each pair of an editor and a language. Instead, each editor implements support for the protocol, and each language only needs one LSP server to get support for eevery editor that supports the protocol (which is most editors nowadays). A demonstration of the Dolmen LSP server in action is available at https://asciinema.org/a/325668.

2.3. Dolmen API

Dolmen exposes its API via a few OCaml libraries, the two main libraries being the parsing library and the typechecking library. Both libraries make heavy use of OCaml's parameterized modules, also known as functors, in order to be parameterized and be as versatile as possible. This design choice was made so that the parsers of the Dolmen library can easily be used to replace parsers in use in existing projects without changing the representation of terms used in those projects. Dolmen also offers a standard representation of terms that can be used to instantiate any of its functors, so that users starting a new project do not have to re-implement the term structure.

Parameterized parsers The Dolmen library provides parameterized parsers. These parsers are typically parameterized over four representations:

- A representation of locations in files, with functions to create locations from line and column numbers. This is used for reporting parsing and lexing errors.
- A representation of identifiers. Identifiers include constant and function names as well as variable names. In some languages, there are more than one syntactic scopes, which are handled using namespaces for identifiers.
- A representation of terms, with functions that will be used by the parser to build the various types, terms and formulas corresponding to the grammar of the input language. All functions of this module typically take an optional location argument.
- A representation of top-level directives. Languages usually defines several top-level directives to more easily distinguish type definitions, axioms, lemma, theorems to prove,

[5]There is a command-line option to limit the maximum number of warnings printed by `dolmen`: by using the `--max-warn=<n>` , only the first n warnings will be printed, and at the end of validation, the number of omitted warnings will be printed by `dolmen`

new assertions, or even sometimes direct commands for the solver (to set some options for instance), as is the case for SMT-LIB where this includes all command names such as `assert`, `declare-fun`, ...

3. The SMT-LIB benchmarks

The latest release of Dolmen, version 0.6, has been used to check the whole set of benchmarks from the SMT-LIB, including both incremental and non-incremental problem files. Out of the 363 750 files of the SMT-LIB benchmarks, there are only 8 465 (2.3%) that fail validation using Dolmen's default options. Most of the errors raised on these files actually comes from linear arithmetic, and in most cases it happens on expressions that are in essence linear, but that do not fit into the SMT-LIB specification of linear expressions. To better handle such cases, the `dolmen` binary has a `--strict=<bool>` option. When provided with the `--strict=false` option, Dolmen turns some, but not all, errors into simple warnings, in cases where it is reasonable to do so. One example of such a case is the example mentioned in the introduction, *i.e.* an expression such as `(* 5 (+ a b))`.

Using the `--strict=false` option results in only 355 (0.1%) files that do not pass Dolmen's validation. Here is a summary of these files:

- 23 files in `incremental/QF_AUFLIA` contain arrays that are not of the sort `(Array Int Int)` (which is the only sort allowed for array terms). Most of these seem to use arrays of sort `(Array Int (Array Int Int))`.
- 13 files in `QF_IDL` contain additions, although the specification does not allow to use addition, only subtraction[6].
- 9 files in `UFIDL` contain additions between two arbitrary terms, although the logic specifies that at least one of the two terms in an addition must be a numeral.
- 5 files in `AUFBVDTLIA` contain expressions of the form: `(* (div j i) i)`, which both makes use of `div` which is not allowed in linear arithmetic, and use multiplication in a non-linear way. Such formulas could be rewritten as `(- j (mod j i))`, though `mod` is not allowed in linear arithmetic either.
- 2 files in `AUFBVDTLIA` contain expressions using non-linear multiplication, of the form `(* (size!210 list!215) (ite ...))`, which are not allowed in linear arithmetic.
- 2 files in `incremental/QF_UFLRA` contain expressions of the form `(* (- (/ 1 2)) (..))`, which is not allowed because `(- (/ 1 2))` is not a rational constant as defined by the specification[7]. Following the specification, it should be instead written as `(/ (- 1) 2)`.
- 65 files in the `QF_SLIA` set of benchmarks directly use Euclidean division. Although there is no explicit definition of the `QF_SLIA` logic available on the SMT-LIB official website, all logics with linear integer arithmetics forbid the use of Euclidean division, so Dolmen assumes that this also holds true for `QF_SLIA`.

[6]These addition appear in the context of let-bound expressions, and it happens that after substitution of all let-bound variables, *and* arithmetic simplification and re-ordering or terms, the resulting expressions in fact can belong to `QF_IDL`

[7]These might be allowed in `--strict=false` mode in a later release of Dolmen

- 45 files in the `QF_SLIA` set of benchmarks contain quantified formulas, with every occurrence being of the form `(exists ((v Int)) (= (* 2 v) (<some term>)))`. These quantified formulas could be re-formulated as `(= (mod (<some term>) 2) 0)`, so while the direct problem is the use of quantifiers in a quantifier-free logic, the root problem in these files is the indirect use of `mod`, which is not allowed in linear arithmetic.
- 191 files in `LIA` use Euclidean division or modulo, which is not allowed in linear arithmetic.

These results show that apart from the array sort problems in `QF_AUFLIA`, all of the problem found were related to linear arithmetic. Considering that these problems do not seem to trouble most solvers, this may suggest that the current specification of linear arithmetic in the SMT-LIB is more restrictive than it needs to be[8]. It might be an important point to look at for the upcoming version 3 of the SMT-LIB.

4. Conclusion

In summary, Dolmen offers a wide range of methods to check the conformance of input files against the SMT-LIB specification: a library API, a command-line binary, and an LSP server, all of these supporting the whole of the SMT-LIB v2.6. Ongoing and future work on Dolmen include:

- Dolmen has recently been extended to support higher-order terms, as will be needed for SMT-LIB v3.
- There is planned work to add to Dolmen the ability to export valid problems into any language supported by Dolmen, which would enable Dolmen to translate problems written in one language (e.g. TPTP, or SMT-LIB) into another language.
- Together with the work on exporting of problems, there are also plans to implement a minimal problem logic detection algorithm in Dolmen. This is important when trying to translate a problem from another language (such as TPTP) into a problem in the SMT-LIB format: indeed, the generated problem needs to be assigned an SMT-LIB logic. In such a case, the translation would either require a manual intervention to specify the desired logic for the generated problem, or the `ALL` logic could be used, both of which are not entirely satisfactory. Instead, one solution would be to scan the input problem in order to determine the smallest SMT-LIB logic in which the problem can be expressed, and use that logic in the generated problem.

By providing an easy way to check for conformance with the SMT-LIB specification, Dolmen will hopefully help both the community move towards conforming with the strict specification more often, and encourage the SMT-LIB specification to evolve. It is the author's hope that this will also encourage SMT-LIB extensions used by SMT provers to either be formally specified (in which case, Dolmen could be extended with these), or to be adopted into the main SMT-LIB specification.

[8]Additionally, there are actually two subtly different specifications of linear arithmetic in the SMT-LIB. The difference concerns whether terms whose top symbol is not an arithmetic operator are allowed to be multiplied by constants. More precisely, only the `AUFLIA` and `QF_AUFLIA` logics allow terms such as `(* 2 (f a))` (according to the specification), whereas logics such as `AUFLIRA` and `QF_AUFLIRA` technically forbid such terms.

References

[1] 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.

[2] Official Specification of the QF_LIA logic of SMTLIB, http://smtlib.cs.uiowa.edu/logics-all.shtml#QF_LIA, 2021.

[3] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, C. Tinelli, CVC4, in: G. Gopalakrishnan, S. Qadeer (Eds.), Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings, volume 6806 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 171–177. URL: https://doi.org/10.1007/978-3-642-22110-1_14. doi:10.1007/978-3-642-22110-1_14.

[4] C. Barrett, P. Fontaine, C. Tinelli, The SMT-LIB Standard: Version 2.6, Technical Report, Department of Computer Science, The University of Iowa, 2017. Available at http://smtlib.cs.uiowa.edu/language.shtml.

[5] F. Bobot, S. Conchon, E. Contejean, S. Lescuyer, Implementing Polymorphism in SMT solvers, in: C. Barrett, L. de Moura (Eds.), SMT 2008: 6th International Workshop on Satisfiability Modulo, volume 367 of *ACM International Conference Proceedings Series*, 2008, pp. 1–5. URL: http://www.lri.fr/~conchon/publis/conchon-smt08.pdf. doi:10.1145/1512464.1512466.

[6] G. Bury, S. Cruanes, D. Delahaye, Smt solving modulo tableau and rewriting theories, in: SMT: Satisfiability Modulo Theories, 2018.

[7] B. Marre, F. Bobot, Z. Chihani, Real behavior of floating point numbers, in: SMT Workshop, 2017.

[8] jSMTLIB: SMTLIB resources, http://smtlib.github.io/jSMTLIB/, 2015.

[9] Microsoft, Official page for Language Server Protocol, https://microsoft.github.io/language-server-protocol/, 2021.