Verifying models with Dolmen

Guillaume Bury[1], Francois Bobot[2]

[1]OCamlPro SAS, 21 rue de Châtillon, 75014 Paris, France
[2]CEA-List, Université Paris-Saclay, F-91120, Palaiseau, France

Abstract

Dolmen provides tools to parse, type, and validate input files used in automated deduction, and in particular problems from the SMT-LIB. We present here an extension of Dolmen which makes it capable of verifying ground models for quantifier-free SMT-LIB problems. While most of the extension was a straightforward implementation of an evaluator for ground expressions, there were some challenges, related to corner cases of the semantics (division by zero, partial functions, …), as well as order of evaluation of statements, most of which could be solved by improvements to the SMT-LIB standard for models.

Keywords

smtlib, model, verification, dolmen, satisfiable

1. Introduction

Model validation is not a particularly rewarding task, particularly for ground formulas: while it is essential to have a model validator in order to organise the model validation track of the SMT-COMP (and others competitions, such as CASC), such a validator involves mostly simple code that often has not much research value in itself. While some solvers implement such a validation internally, it is not easy to find independent tools that could validate the model output by a solver. The only such tool known to the author is a python script [1] (based on pySMT [2]) used by the SMT-COMP for the model validation track from 2019 to 2022, but the only supports a subset of quantifier-free logics from the SMT-LIB.

Dolmen [3] is a project providing parsers and a typechecker for some of the most commonly used languages in automated deduction, such as SMT-LIB, TPTP, Dimacs, iCNF, Zipperposition and Alt-Ergo's native formats. It is written in OCaml and is available at https://github.com/Gbury/dolmen, under a BSD2 license. Dolmen is used as a validator for the submission of new benchmarks to the SMT-LIB, and it's also used in the frontend of some solvers, such as Colibri2 and Alt-Ergo.

In this extended abstract, we present the work that has been done on Dolmen to add model validation capabilities for quantifier-free problems. That work will be part of the 0.9 release of

SMT Workshop, July 5–6, 2023

guillaume.bury@gmail.com (G. Bury); francois.bobot@cea.fr (F. Bobot)

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

```
; example.smt2
(set-logic ALL)
(declare-fun a () Int)
(declare-fun b () Int)
(declare-fun c () Int)
(assert (= (+ a b) c))
(check-sat)
```

```
; example.rsmt2
sat
(model
  (define-fun a () Int 1)
  (define-fun b () Int 2)
  (define-fun c () Int 3)
)
```

```
# dolmen --check-model=true -r example.rsmt2 example.smt2
```

Figure 1: Example of using Dolmen to validate a model

dolmen, and with it, Dolmen can now verify models for all quantifier-free logics of the SMT-LIB except for those using the string theory.

2. Verifying models using Dolmen

2.1. Cli and usage

In order to verify a model, one can simply call the Dolmen binary with both the input problem file, and the model file, as shown in Figure 1.

One interesting thing to note is the `.rsmt2` extension for the model file. The choice was made to distinguish the language used for input problems (i.e. SMT-LIB) from the language used for model files, because the answers that make up model files are not a subset of the SMT-LIB. More precisely, the syntax of terms are identical, but not the syntax of statements: model answers (and therefore model files) start with `sat` ,and then contain a list of definitions in parentheses. Therefore, model files look like

```
sat (
  (define-fun ...)
)
```

This does not match the SMT-LIB's notion of statements, most notably because the `sat` keyword is before the opening parenthesis. This lead to the decision of considering that model files are written in a distinct language, for which the extension `.rsmt2` was chosen, the `r` letter meaning "response" since such files contain solvers' response to input problem files. This may of course evolve and change if the community reaches a consensus on how to handle such solver outputs.

```
type value
(** The type of values *)

type 'a ops
(** A type for operations for values that wrap something of type 'a *)

val mk : ops:'a ops -> 'a -> value
(** Create a value. *)

val extract : ops:'a ops -> value -> 'a option
(** Try and extract a concrete type out of a value. This will return [None]
    if the value does not match the given [ops]. *)

val ops:
  compare:('a -> 'a -> int) ->
  print:(Format.formatter -> 'a -> unit) ->
  unit -> 'a ops
(** Create a new [ops] in order to create values from concrete types. *)
```

Figure 2: Interface for extensible values

2.2. Evaluating expressions

Adding model validation capabilities to Dolmen was straightforward, and mainly required implementing an evaluator for typed expressions: indeed Dolmen was already capable of parsing and typing SMT-LIB problems, as presented in [3]. Just as for the parsing and typing parts of Dolmen, this evaluator is implemented as a standalone library, and then used in the Dolmen binary.

This section describes the evaluator: the evaluator takes an environment and a typed expression, and returns a value. Additionally, the evaluator itself can be separated into a core evaluator and a list of evaluating functions that handle evaluation of symbols from theories.

Values Much like its typechecker, the evaluator in Dolmen was made to be easily extensible, so that new theories can be added easily. In order to make that possible, the representation of values was made to be extensible, so that each theory can define its own values. That way, the theory for integer arithmetic can define its values as gmp integers, while the core theory defines booleans as values.

In practice, this extensibility of values is achieved using the interface in Figure 2. Each theory can extend the type of values by using the `ops` function to create a new way to create values, and then extract them. A value is then equivalent to a pair `'a ops * 'a` for some `'a`. This extensibility has two advantages: the first one is that it makes the code for each theory self-contained, since each theory can define both its own kind of values, as well as the operations on these values. The second advantage is that it makes it possible for users of the evaluator (which is distributed as an OCaml library) to also define their theories.

Typed expressions The typed expressions of Dolmen are quite simple, and can fall into one of the following five cases:

- Variables, these are either let-bound, or are function parameters
- Constants, which cover the interpreted and non-interpreted symbols and functions
- Function application: a function application is made of a callee, and a list of arguments, all of which are typed expressions. The callee in particular is a typed expression because Dolmen handles higher-order expressions, in preparation for the upcoming SMT-LIB v3
- Binders: either a let-binding, or a lambda/anonymous function (technically, existential and universal quantifications are also binders, but we're currently only targeting quantifier-free problems for model validation in Dolmen)
- A pattern match, that is a scrutinee and a list of patterns and expressions (which we'll call arms).

Environment and core evaluator An environment is a function map from variables and constants to values. The environment also maps constants (and not only variables) because it has to store values for the constants which are declared in the input problem and then defined in the solver output. The core evaluator uses this environment to evaluate typed expressions, following their structure:

- For variables their values can simply be fetched from the environment
- Constants are either interpreted by one of the theories, or fetched from the environment
- Let-bindings simply evaluate the defining expression, and add the correct binding to the environment
- Function abstractions create suspensions, that is values that wait until they are fully applied to arguments before actually being evaluated. This suspension is a pair, storing the list of parameters and the body of the function. Once this function value is applied to arguments, bindings from the parameters to these arguments are added to the environment, and then the body is evaluated. The exact environment where the body is evaluated does not matter: the type-checking phase of Dolmen has disambiguated every symbol from the original problem and assigned unique identifiers to every identifiers according to the static scoping rules of the SMT-LIB : in particular two identifiers that share the same name (e.g. because of name shadowing) but are not the same will have different unique identifiers. Therefore, the only requirement on the environment is that all symbols used are bound in the environment.
- Pattern matching is implemented very simply: the scrutinee of the match is evaluated, and then the value is matched against each pattern, in order, until a match is found. At that point, the corresponding arm is evaluated.

Theories and Function values As mentioned above, Dolmen supports higher-order expressions, which incidentally made the implementation of theories simpler. Indeed, since Dolmen supports higher-order, functions and partially-applied functions need to be represented by some value. These functional values store two main pieces of information: a body and an arity,

which is the number of arguments needed in order to evaluate the body. The body can itself be either a typed expression, or can be arbitrary OCaml code. That last case is how most of the symbols from builtin theories are interpreted. Finally, partial applications (i.e. applications with a number of arguments that is lower than the arity) simply store the evaluated arguments until the arity is reached and the body can be evaluated.

That way, theories are implemented by returning functional values containing OCaml code for their operators (such as e.g. addition). This simplifies the implementation because this means that theories simply have to return a value for builtin symbols. So a theory for the evaluator is simply a function from symbols to values. Without higher-order and functional values, theories would need both the symbol and the arguments in order to evaluate to a value, which would complicate the code for theories.

3. Challenges and corner cases

While the implementation of the evaluator was relatively simple and straightforward, there were some challenges that occurred while testing the resulting model validator on real examples. This section explores some of these challenges.

3.1. Partially defined functions

There are quite a few functions in the SMT-LIB standard [4] that have only partially defined semantics. Some of those include:

- Division in arithmetic theories (as well as the modulo/remainder): these are usually not defined when the divisor is 0.
- The min and max functions for floating point numbers are not defined when given as arguments $+0$ and -0. Note however, that contrary to other partially defined function, the min and max functions cannot return arbitrary values when given $+0$ and -0 as input, and must return one of the two, but which one is returned is unspecified. This actually brings up the problem of how to specify specially constrained extensions of functions such as this (or how to check that the extension provided by the solver conforms to the constraint), compared to unconstrained extensions such as for the division by zero.
- The fp.to_$*$ functions from the floating point theory are not defined when their inputs are Nan or infinity values.
- Additinally, the fp.to_ubv and fp.to_sbv are not specified for values outside their range (e.g. fp.to_ubv is not defined for negative floating-point values)
- Destructors of polymorphic datatypes that contain more than one constructor. The semantics of such destructors is only specified when the argument is constructed from the corresponding constructor. For arguments constructed from other constructors, the semantics of destructors is not specified. For instance, consider the following definition of polymorphic lists where the destructor head is not defined on the empty list nil:

```
(declare-datatypes ((list 1)) ((par (alpha) (
  (nil)
  (cons (head alpha) (tail (list alpha)))
```

```
))))
```

Due to these partial specifications, some problems might be satisfiable only under some extensions of these partially defined functions: for instance, it might be necessary that `(div 5 0)` equals 13, and indeed quite a few solvers are able to reason under such assumptions and can end up choosing how to extend partially defined functions. Therefore, in order to validate models for such problems, the extensions chosen by the solver must be specified as part of the model. For a concrete example, consider the following example:

```
(set-logic ALL)
(declare-fun a () Int)
(declare-fun b () Int)
(declare-fun c () Int)
(declare-fun d () Int)
(declare-fun z () Int)
(assert (= z 0))
(assert (= c (div a z)))
(assert (= d (div b z)))
(assert (not (= c d)))
(check-sat)
```

This problem is satisfiable, as shown by the following (partial) model:

$$\begin{cases} z \mapsto 0 \\ a \mapsto 1 \quad c \mapsto 13 \quad \frac{1}{0} \mapsto 13 \\ b \mapsto 2 \quad d \mapsto 42 \quad \frac{2}{0} \mapsto 42 \end{cases}$$

One challenge of that is how to adequately represent the information about the special cases of division in the models generated by solvers, and how to correctly use that information in Dolmen in order to validate models. Historically, some solvers did that by defining a specific symbol, such as the div0 symbol defined by z3. However, after some discussions with Jochen Hoenicke and Martin Bromberger, in preparation for the model validation track of SMT-COMP 2023, it was decided to simply reuse div. This had the advantage that it extends very naturally to all builtin functions that are only partially defined without having to find alternative names (such as div0).

Therefore, the model shown above would be expressed as follows in the SMT-LIB syntax:

```
sat
(
  (define-fun z () Int 0)
  (define-fun a () Int 1)
  (define-fun b () Int 2)
  (define-fun c () Int 13)
  (define-fun d () Int 42)
  (define-fun div ((x Int) (y Int)) Int
    (ite (and (= x 1) (= y 0)) 13
     (ite (and (= x 2) (= y 0)) 42
      (div x y))))
)
```

Note the fallback case of the division as defined: outside of the specific cases needed for the problem, the regular division is returned. While this may seem like a recursive call, we instead interpret it as an instance of constant shadowing: in effect the model defines a new division symbol that shadows the builtin division symbol, and the internal call to div refers to the builtin division, so there is no recursion. This has the nice property that one could use this shadowing behaviour to very easily implement the correct model validation behaviour: uses of div would first evaluate to the body defined in the model, and "fall back" to the regular division when arguments do not match. This is however, not what Dolmen does: indeed it would be difficult to statically determine whether the version of div in the model is actually an extension of the partially defined builtin division (i.e. that the function provided in the model agrees with the builtin division on all inputs where the divisor is not 0). Consider what would happen if the following definition was provided in a model:

```
(define-fun div ((x Int) (y Int)) Int (ite (= x 1) 13 (div x y)))
```

In order to guarantee that the model does not mistakenly and wrongly overwrite the defined behaviour of partially defined symbols, Dolmen only calls the version provided by the model in cases where the builtin's behaviour is undefined.

Comparing thte two solutions, one advantage of the div solution is that it does not require reserving another name, unlike the div0 solution, which cannot work if the input problem declares or defines a symbol named div0. Currently Dolmen implements both solutions for extensions of partially defined symbols, and therefore accepts both div and div0 to extend division for instance, but div is likely to be the preferred solution in the future

3.2. Order of statements

Another challenge encountered was related to performance, and in particular memory consumption, due to the SMT-LIB standard not requiring any order relation between the statements from the input problem and the definitions in models.

For the purpose of validating models, top-level statements in SMT-LIB problems can be classified into the following 4 categories:

- Type declarations, such as `(declare-sort t 0)`
- Term declarations, such as `(declare-fun a () Int)`
- Definitions, such as `(define-fun b () Int a)`
- Assertions, such as `(assert (= b 0))`

There are no restrictions on the order in which statements from these 4 categories can appear in SMT-LIB problems, apart from the scope of definitions (i.e. one cannot use a symbol before its definition). The problem is that:

- In order to correctly parse and type a model (which consists only of definitions), one has to have seen the type and term declarations of the input problem.
- Definitions and declarations from the input problem need to have access to the typed model: indeed as shown in the examples above, definitions can refer to declared symbols, whose values come from the model. Therefore, in order to evaluate a definition from the input problem, we need access to the values of symbol defined previously in the input problem.

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

```
sat
(
  (define-fun a () Int 0)
  (define-fun b () Int 0)
  )
```

```
# dolmen --check-model=true --check-model-mode=interleave -r example.rsmt2 example.smt2
File "example.smt2", line 3, character 0-16:
3 | (assert (= b 0))
    ^^^^^^^^^^^^^^^^
Error The following constant is not defined, and thus has no value: ‘b‘
Hint: The interleave mode for checking model requires constants in the model
  file to be defined in the same order as they are declared in the input
  problem file, you might want to try using the ‘--check-model-mode=full‘
  option.
```

Figure 3: Problem and model with problematic order of statements

- Checking assertions from the input problem requires access to the model, as well as access to the values of symbols defined in the input problem.

Considering these constraint, the straightforward way to validate a model would be to do the following:

1. Parse and type the input problem, and store the definitions and assertions
2. Use the resulting typing environment to type the model
3. Evaluate all the definitions from the input problem and add them to the model
4. Evaluate all assertions from the input problem and error out if any evaluate to false.

This solution has the advantage that it can correctly handle any problem and any model, however it requires storing some definitions and assertions in order to evaluate them later. While this may seem reasonable, and works correctly on reasonably sized problems, some problems in the SMT-LIB benchmarks are remarkably (some would even say unreasonably) large, with some problems of more than 2GB in size. For such problems, storing all definitions and assertions can easily take up more than tens of GB of memory, because of all the meta-data that Dolmen attaches to typed expressions: for instance, Dolmen attaches location information to every term node (to generate good error messages), as well as type information.

In order to circumvent this memory consumption problem, Dolmen implements two different modes for validating models: the first one (and default) is the straightforward approach outlined above and suffers from the memory problem. The second mode tries to interleave the input problem with the model as follows :

1. Dolmen types the input problem until a definition or assertion is reached
2. At that point Dolmen starts (or continues) to type as much of the model as possible, and stops when it encounters a definition (in the model file) for a symbol whose declaration (in the input problem) has not yet been seen
3. Then Dolmen tries to evaluate the definition or assertion (the one found at step 1): if its evaluation succeeds, then repeat the process, else, if the evaluation fails because some symbol's value is missing in the evaluation environment, it means that the input problem and model do not have a suitable ordering and the straightforward method should be used instead.

In this interleaved mode, Dolmen does not need to store any definition/assertion, which greatly reduces memory usage, but it requires that the model definitions appear in a compatible order, which may not always be the case. For instance consider the satisfiable problem and associated model in Figure 3, where the definitions have been sorted using the symbols' names. Dolmen stops at the first assertion at line 3, and starts typing the model, however the first definition in the model is for the symbol `a` whose declaration has not yet been seen (since it is at line 4 in the problem file), which results in an error. As the reader can see, in such cases, Dolmen also displays hints to suggest other options that can be tried (similarly, another hint is displayed when running out of memory while validating a model, suggesting to use the `--check-model-mode=interleave` option).

4. Conclusion

Dolmen now offers a command line tool to validate ground models (i.e. for all `QF_*` logics, excluding the `String` theory). The implementation and testing of that new feature helped discover aspects of the current standard that are either lacking (e.g. for partially defined functions), or that could be improved to help Dolmen and similar tools (e.g. concerning order of statements).

In the same way that Dolmen helps maintainers and submitters of the SMT-LIB benchmark library verify that new benchmarks respect the standard, we hope that the model validation feature of Dolmen can help start a discussion about how to evolve the current standard for easier model verification, as well as help authors of solvers to check the models generated by their tools against the standard. We therefore encourage authors of solvers to try Dolmen on their models, and report their findings.

References

[1] S.-C. organizers, Model Validation script for the SMT-COMP, https://github.com/SMT-COMP/postprocessors/tree/master/model-validation-track, 2023.

[2] M. Gario, A. Micheli, Pysmt: a solver-agnostic library for fast prototyping of smt-based algorithms, in: SMT Workshop 2015, 2015.

[3] G. Bury, Dolmen: A validator for smt-lib and much more., in: SMT, 2021, pp. 32–39.

[4] 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.