SMTpp: preprocessors and analyzers for SMT-LIB

Richard Bonichon[1], David Déharbe[1], Pablo Dobal[2], and Cláudia Tavares[1] *

[1] DIMAp, Universidade Federal do Rio Grande do Norte, Natal, RN, Brazil
richard@dimap.ufrn.br, david@dimap.ufrn.br, claudia@ppgsc.ufrn.br
[2] INRIA, Universite de Lorraine & LORIA, Nancy, France
pablo.dobal@inria.fr

Abstract

We present SMTpp, a tool that operates both as a source-to-source transformer and analyzer for SMT-LIB. The first goal of SMTpp is to offer a framework to which developers of SMT solvers can delegate computing tasks that are both necessary to be competitive (at least on some families of benchmarks), and somehow irrelevant to the essence of SMT-solving. The second goal is to provide facilities to simplify and classify SMT-LIB benchmarks.

1 Introduction

SMTpp is an initiative to separate symbolic and computational aspects of SMT solvers. Most of the ideas developed in this paper come from two initial separated observations:

1. SMT solvers have many components: parser(s), normalizers, heuristics, decision procedures, a SAT solver, etc. One may argue that some of these are not directly related to the initial core of satisfiability modulo theories.
2. Curators of SMT-LIB [4, 3] sometimes need to verify scripts from the library and will not want to have to do it manually, with a growing number of problems.

Observation 1 justifies simplification of SMT-LIB problems as a stand-alone tool. Observation 2 leads to the development of analyses and checks to reduce the load of taking care of a growing SMT-LIB. This part is arguably the most developed of SMTpp.

SMTpp is coded in OCaml[1], an impure functional language which has already been heavily tested in the kind of symbolic analysis and program transformation we do. Frama-C [11], Coq[2] or the OCaml compiler itself are good examples.

SMTpp has grown out of the veriT SMT solver [7] as a solution to externalize simplifications and manipulations that are deemed less central to SMT solving, but also as a future easier way to try experimental features in a higher level language than C. SMTpp is for now available from `http://www.verit-solver.org` under the ISC license.

2 Related work

SatELite [16] is a preprocessor aimed at SAT-solving and has been shown useful to improve overall performance for some classes of problems. Cok's jSMTLIB [9] has the most similar scope

*This work has been supported by the ANR/DFG project STU 483/2-1 SMArT, project ANR-13-IS02-0001 of the Agence Nationale de la Recherche, by the European Union Seventh Framework Programme under grant agreement no. 295261 (MEALS), by the Région Lorraine, by CAPES PNPD grant no. 2314/2011, by the CAPES/STIC AmSud MISMT and CNPq grant 573964/2008-4 (National Institute of Science and Technology for Software Engineering—INES, `www.ines.org.br`).

[1] `https://ocaml.org/`
[2] https://coq.inria.fr/

as SMTpp. Thus we are using it as one measuring stick for SMTpp, both in terms of features and efficiency. The main similarity is that both jSMTLIB and SMTpp aims at parsing, typing and checking a SMT-LIB script before letting a SMT-solver handle the satisfiability search. There are also significant differences in the goals of the projects. jSMTLIB offers a programmatic API or an Elipse plug-in where SMTpp does not try to tackle these problems. Conversely, SMTpp's analyses and transformation goals are more ambitious.

The simplifications we present in Section 3 can also be present in a number of already available tools. Similar ones are for example already implemented in veriT.

The logic detection feature described in the same Section 3 has some similarities to the logic probing of Z3 [12]. It aims to be more precise and complete while retaining correctness. All SMT provers detect undefined symbols out of pure necessity, and the extension to unused symbols presented in Section 3 is commonly found in compilers and static analyses tools.

3 Modules

SMTpp is organized into a set of modules, whose functionality is as clearly delimited as possible. As of now, these modules almost form a straight mapping to OCaml modules. The architecture aims at implementing a set of transformations from abstract syntax tree (AST) to AST so that most modules can be chained one after another. This idea is clearly borrowed from the organization of compilers, where passes usually occur one after another.

Generic SMT-LIB handling As the goal of SMTpp is to be used in the context of SMT-LIB, the first step was to develop a parser and a pretty-printer. The parser is generated with Menhir [19] and can be used to verify the syntactic correctness of SMT-LIB scripts. It reports the locations of errors in the file and supports the draft of SMT-LIB 2.5.

A generated parser has a maintenance advantage. Furthermore, Menhir's standard library also has some further pros such as parametric rules, which bear some resemblance with parser combinators, having the possibility to name elements of the grammar, and nicer conflicts explication than the standard ocamlyacc tool.

SMT-LIB logics and theories are encoded as OCaml modules and form the basis of the other modules, which all use the AST produced by the parser or an extended version thereof.

Multi-scripts generation SMTpp can be used to slice a script with multiple `check-sat` commands with or without `push` or `pop` into several scripts with a single `check-sat` command. This features make it possible to use solvers in non-incremental mode to verify benchmarks that use SMT-LIB's incrementality features. This option writes the AST to files instead of to the standard output. Figure 1 shows on the right-hand side the result of applying this option to the script on the left-hand side.

Logic detection The logic detection module can detect and set the logic of a SMT-LIB script. It can be used as a verification tool such as in Section 4.2 or, to simply (re)set the logic of a script. The latter function was the one that started this module: indeed, proof generating tools can sometimes produce scripts without a beginning `set-logic`. Thus, we could compute a logic among the available logics of SMT-LIB. This absence of `set-logic` has been for example observed in the first version of the SMT-LIB scripts produced by the BWare project [15]. This has been since then corrected, and SMTpp agrees with the choice of logic.

```
(set-logic QF_LIA)
(declare-fun w () Int)
(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun z () Int)
(assert (> x y))
(assert (> y z))
(push 1)
(assert (> z x))
(check-sat)
(pop 1)
(push 1)
(check-sat)
(exit)
```

```
(set-logic QF_LIA)
(declare-fun w () Int)
(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun z () Int)
(assert (> x y))
(assert (> y z))
(assert (> z x))
(check-sat)
(exit)
```

```
(set-logic QF_LIA)
(declare-fun w () Int)
(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun z () Int)
(assert (> x y))
(assert (> y z))
(check-sat)
(exit)
```

Figure 1: Multi-scripts generation

Some SMT provers, like Z3 or CVC4 [1], still work without a `set-logic`. CVC4's documentation states that it uses the "non-standard ALL_SUPPORTED logic". Z3 is a bit more clever, as it probes the AST for hints of specific logics. Others, such as veriT, just do not work properly.

The logic detection module is similar in spirit to Z3's logic probing. It aims to be correct, in the sense that it will compute an over-approximation of the needed logic. In the presence of arithmetic, this can be a problem, as the tool often quickly converges towards non-linear arithmetic. Section 4.2 sums up the test results for this module.

Def/use analysis SMTpp has a simple feature to identified unused and undefined symbols. While undefined symbols are definite errors, unused symbols are code smells that have no impact on correctness. Although the code is still under development, it has generated good results on SMT-LIB (see Section 4.3). This analysis is evolving towards a true data-flow analysis in order to enable optimizations borrowed from compilers, such as constant propagation and common subexpression elimination.

Simplifications The last module is a set of simplifications performed on the scripts and formulas in QF_LRA family. The simplifications steps originate from the LassoRanker subfamily. Each transformation alone is naive but the combination generates a nice boost in performance for the two state-of-the-art SMT solvers Z3 and CVC4.

The most trivial simplifications use properties of neutral and absorbing elements to apply the following rewriting steps:

$$\begin{aligned} t_1 + \cdots + t_n + 0 \ \text{op}\ 0 &\rightsquigarrow t_1 + \cdots + t_n \ \text{op}\ 0 \qquad (\text{where op} \in \{<, >, \leq, \geq, =\}) \\ t_1 * \cdots * t_n * 1 \ \text{op}\ 0 &\rightsquigarrow t_1 * \cdots * t_n \ \text{op}\ 0 \\ t_1 * \cdots * t_n * 0 \ \text{op}\ 0 &\rightsquigarrow 0 \ \text{op}\ 0. \end{aligned}$$

Afterwards, *constraint subsumption* is applied to rewrite the structure of formulas and reduce their size. Given a set of conjunctive constraints, we identify pairs of constraints where one of them is included in the other. For instance, from $A < B \wedge \ldots \wedge A \leq B$ we remove the latter.

Next, the boolean structure can be simplified, similarly to arithmetic, through repeated identifications of neutral and absorbing elements. This amounts to finding trivially *true* ($\top$) or *false*($\bot$) conditions in the constraints involved, by applying the rules below:

Before simplification	After simplification
$\ldots$	$\ldots$
$x_4 + r_1 = 0.0 \ \wedge \ldots$	$x_4 + r_1 = 0.0 \ \wedge \ldots$
$x_0 - x_1 + r_2 = 0.0 \ \wedge \ldots$	$x_0 - x_1 + r_2 = 0.0 \ \wedge \ldots$
$-x_2 + x_3 + r_3 = 0.0 \ \wedge \ldots$	$-x_2 + x_3 + r_3 = 0.0 \ \wedge \ldots$
$x_0 - x_1 + 7.0x_2 - 7.0x_3 - x_4 + d + s < 0.0$	$-r_2 + 7.0r_3 + r_1 + d + s < 0.0$

Figure 2: Simplification example

$t \wedge \bot \rightsquigarrow \bot$ $\quad t \wedge \top \rightsquigarrow t$ $\quad t \vee \bot \rightsquigarrow t$ $\quad t \vee \top \rightsquigarrow \top$

In order to enhance the number of application of the previous step we also identify trivially *true* / *false* conditions on linear constraints. Given n and m numbers, from n op m we derive either *true* or *false* depending on n, m and op $(<, >, \leq, \geq, =)$.

The last step applies to linear expressions and tries to reduce the size of the problem for the arithmetic reasoner. This technique is suitable for benchmarks containing conjunctions of linear constraints with variables local to the conjunction. A motivational example is shown in Figure 2 where the x_i are local variables and so candidates to be subtituted by others.

This technique amounts to performing Gauss elimination so that the matrix corresponding to the linear constraints gets sparser after the transformation. Given a conjunction of linear constraints, we identify all equalities where one side is zero. We then use occurence metrics to identify in the linear expressions the variables that may be substituted.

We observe that the substitution method is dependent of the formula size: the bigger the formula, the better the results. This is not surprising since this method was initially motivated by large formulas. Note that this is a heuristic approach that may be counterproductive in some cases. The impact of these simplifications is measured in Section 4.4. While the heuristics works well in practice, we believe there is still room for improvement.

4 Experimental results

We conducted a series of experiments using SMT-LIB (101684 problems) and scripts from the BWare project (12872)[3] as benchmarks to evaluate the components described in Section 3. The results below only show a short summary of the data. More detailed results and raw data are available at `http://github.com/ossanha/smt2015`, as well as some documentation about how to reproduce our experiments.

4.1 Parser efficiency

The basic functionality of SMTpp is to parse the SMT-LIB. Our goal is to be reasonably efficient while easing maintenance. Thus, this lead to the choice of Menhir as parser generator and of a clean parser which only generates the AST and does nothing else.

Efficiency is usually not that important whenever problems are handled in small quantities. However, as we aim to be relevant for complete SMT-LIB analyses, even a small advantage has more significance. For the evaluation, we have selected the following tools:

[3] Files available from `http://bware.lri.fr/index.php/Benchmarks`

- Z3 4.3.2, patched to just parse the data, to give an idea about where we stand with respect to what can be considered one of the most efficient generic SMT-solvers;
- Alt-Ergo[5] 0.99.1, patched to not elaborate the second AST, with –parse-only option and smtlib2[4], two other tools written in OCaml (their parsers are generated with ocamlyacc);
- jSMTLIB, written in Java, the tool with the most similar goals to ours;
- SMTpp both in native and bytecode versions, and also native code with logic detection to estimate the cost of this analysis. The bytecode version is expected to provide a fairer comparison point with jSMTLIB than the native code one.

Expected results. We expect Z3, with a handwritten C++ parser, to come easily ahead and jSMTLIB to finish last on most benchmarks, due to the cost of using the JVM and making some analyses such as typing. Among OCaml-based parsers, we expect the native code SMTpp to be more efficient both than Alt-Ergo and smtlib2 due to the use of Menhir. Indeed, according to a note in Menhir's documentation, parsers produced by Menhir can be 2 to 4 times faster than the ones generated by ocamlyacc. Thus, observing this kind of relation between SMTpp and smtlib2 or Alt-Ergo would mean that the main gain comes from the choice of parser generator, and are transferable without much effort to those other tools.

Tool comparisons. When comparing two tools with the same return code, a tool is said to beat another on a benchmark if the time difference is at least 20% and higher than 0.02 second or if one fails. A match is declared a draw if draws represent at least half the scores and no tool has more than twice the number of wins of the other. Otherwise, the number of wins decides the winner. The winner of a confrontation scores 2 points, the loser 0; if it is draw, both score 1 point. In Figure 3 the number is the ranking of the tool in the corresponding category.

Tools	ALIA	AUFLIA	AUFLIRA	AUFNIRA	BV	LIA	LRA	NIA	NRA	QF_ABV	QF_ALIA	QF_AUFBV	QF_AUFLIA	QF_AX	QF_IDL	QF_LIA
smtlib2	1	2	4	1	4	1	1	1	1	6	5	5	1	1	5	5
Z3	1	4	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Alt-Ergo	1	6	6	1	6	1	1	1	1	5	4	6	1	1	2	2
SMTpp (bytecode)	1	5	4	1	1	1	1	1	1	3	5	4	1	1	5	5
SMTpp (native)	1	1	1	1	1	1	1	1	1	1	1	1	1	1	3	3
SMTpp (native,detect)	1	2	1	1	4	1	1	1	1	3	1	3	1	1	4	4
jSMTLIB	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7

Tools	QF_LRA	QF_NIA	QF_NRA	QF_RDL	QF_UF	QF_UFBV	QF_UFIDL	QF_UFLIA	QF_UFLRA	QF_UFNRA	UF	UFBV	UFLIA
smtlib2	6	1	6	6	6	5	6	1	6	5	5	5	6
Z3	1	1	1	1	1	1	1	1	1	1	1	1	1
Alt-ergo	3	1	1	2	3	5	2	1	3	2	1	5	1
SMTpp (bytecode)	5	1	1	5	4	4	5	1	5	5	5	4	5
SMTpp (native)	2	1	1	2	2	2	2	1	1	2	1	2	1
SMTpp (native,detect)	4	1	1	4	4	3	2	1	3	2	1	3	1
jSMTLIB	7	7	7	7	7	7	7	7	7	7	7	7	7

Figure 3: Parser rankings on SMT-LIB

The actual results, without reading too much into them, confirm the expectations. The clear winner is Z3, followed by native SMTpp and Alt-Ergo. Activating logic detection keeps SMTpp native slightly faster than its bytecode version. The difference between most tools are actually small and have been observed to be subject to variations. The only significant difference comes from jSMTLIB, which always ranks last. Some of its relative slowness can be attributed to the additional analyses performed, some of it to the use of Java. These results only show that the choice of a generated parser within a high-level language such as OCaml is relevant to the challenge at hand. Even bytecode SMTpp can be competitive with native code Alt-Ergo.

[4] Available from `http://www.cs.uiowa.edu/~astump/software/ocaml-smt2.zip`

4.2 Logic detection in SMT-LIB

We ran the detection module on SMT-LIB and BWare files as a way to get a first feedback on the quality of the detection and root out a number false positives. We have seen in Section 4.1 that this analysis is not very time-consuming, mainly due to approximations.

SMTpp found that a bit more than 5% of SMT-LIB has logic declarations that overestimates the logic used in the script. At the same time, 7% of the logics were overestimated by SMTpp, saying for example that the arithmetic is non-linear whereas it is declared to be linear. These results are summed up in Figure 4.

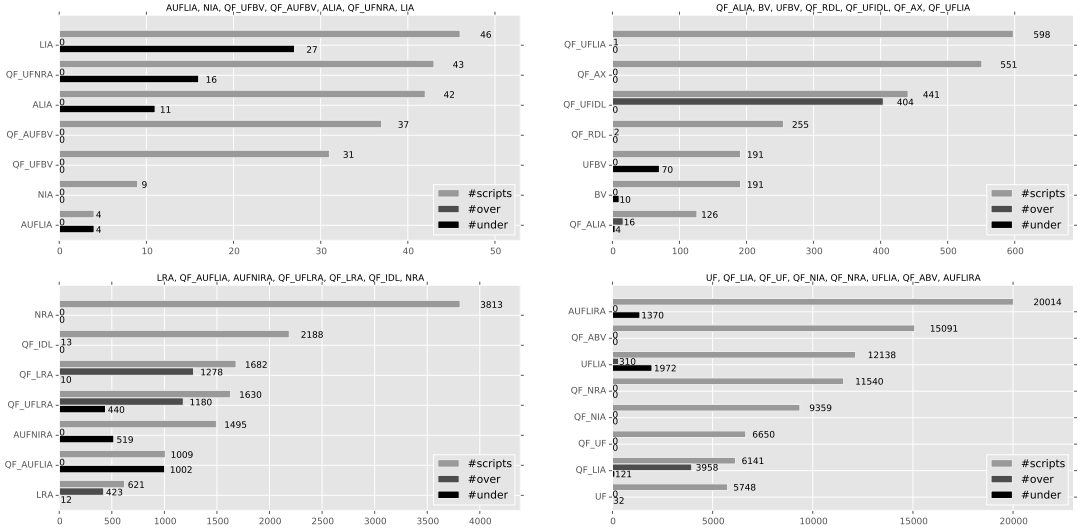

Figure 4: SMTpp logic detection on SMT-LIB

The first kind of difference found consists in discrepancies between the subsets of arithmetic declared and inferred, due to over-approximations of the tool. As of now, we do not consider this to be a problem and consider that the SMT-LIB classification is right in this case. We know that SMTpp often triggers a non-linear arithmetic verdict as a quick safe over-approximation.

Most of the under-approximations that SMTpp detects can be easily verified but even verifying 5% of SMT-LIB is tedious business due to the sheer size of the library itself. Here is a digest of some elements we have checked:

- 1368 scripts in AUFLIRA/FFT and AUFLIRA/why do not use the word Array in their body. The explanation is that they were automatically generated with this overestimated logic, but they should be UFLIRA: a category to be created?
- There is a whole family of scripts under UF/grasshopper that are fully propositional, if that, in that they mainly sport a single `(assert false)` assertion.
- More surprisingly, a number of scripts in QF_UFLRA, almost 20%, do not use arithmetic at all, all they do is define two boolean constants, assert them and check satisfiability.
- Most of QF_AUFLIA only uses the Int sort and no arithmetic functions and a fourth of the problems have neither free sorts nor functions.
- We also found a number of arithmetic-based problems which were declared to be linear and do not even use the + symbol. These are inferred to be difference logic. This is not

always true. However, the difference between what constitutes difference logic or linear arithmetic is not always clear cut in SMT-LIB. Indeed, we have found a number of over-approximations to linear arithmetic due to the presence of + in difference logic directories, even though this symbol should not occur according to the category definition.

This module is still a work in progress, and we need to be cautious about the soundness of all the approximations we make. These first results are an encouraging step to continue the development of this part of SMTpp.

4.3 Unused symbols in SMT-LIB

As in the case of the logic inference module, we tested the def/use module on SMT-LIB, as a way to check the correctness of the analysis while possibly getting some information on SMT-LIB. We expect this analysis to help SMT-LIB back-ends generate trimmed down scripts. This section focuses on unused symbols since undefined symbols do not occur in SMT-LIB.

Actually, a certain number of scripts do have unused symbols. While this has no effect on the soundness of the satisfiability of the problem, it would be worthwhile to sanitize the offending files. One of the causes is that automatically generated scripts often include the same exact prelude. The files generated for the BWare project are known to have this type of issue. Figure 5 sums up our experimental results for SMT-LIB.

This analysis is easy (but slow) to check with grep -c. This verification helped us find the third point below. Here is a short list of observations.

- SMT-LIB scripts of the BWare projects all have a common prelude which always has more declarations than necessary: all of them have unused symbols.
- A number of scripts, such as QF_LIA/fft/Sz64_1160.smt2, bind variables by `let` constructs without using them afterwards.
- The entire subdirectory LRA/keymaera exhibits a pattern of declaring a constants, then binding a variable *with the same name and type* by an `exists`.
- QF_ABV/stp has big ($\approx$ 60Mb) scripts in which 1/3 to 1/2 is unused declarations. Removing these will at least speed up the parsing phase.

4.4 Simplifications in QF_LRA

We elected all formulas from QF_LRA and solved them using CVC4 and Z3, both with and without applying the simplifications described in Section 3. All solvers have a timeout of 90 seconds. The preprocessing helped both solvers solve more formulas. Whereas CVC4 solved 45 more instances, Z3 solved 27 more. In addition both solvers have improved the cumulative time used for solving the instances to the LassoRanker family, where the work finds its motivation. However, we also observed some slight performance improvement on the whole QF_LRA family. These results, shown in Figure 6, confirm that it might be worth investing efforts on identifying generic preprocessing steps in order to boost solvers performance.

In the actual state of SMTpp, preprocessing times is less than 20% of time spent for more than half of solved problems, and less than 40% for 75% of solved instances. On average, the time taken on common solved instances is similar, even though there are outliers on both sides.

As shown in Figure 6, a time limit around 90 seconds exhibits more unique successes for Z3 than for SMTpp + Z3. This might be because of SMTpp's unstable state but further investigations might also uncover a more profound reason. As the time limit increases, the number of solved instances show a distinct advantage in using a preprocessed problems.

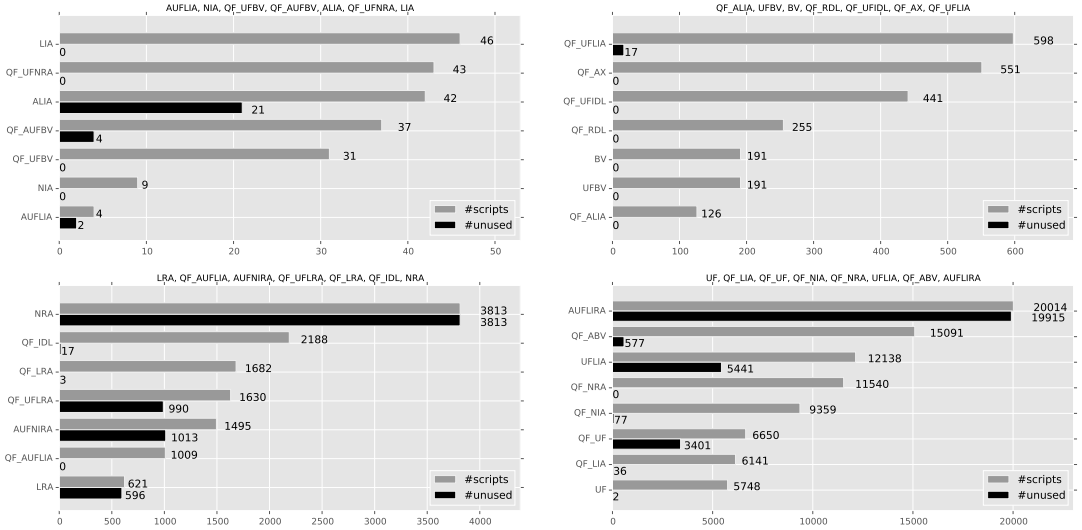

Figure 5: Scripts with unused symbols detected in SMT-LIB

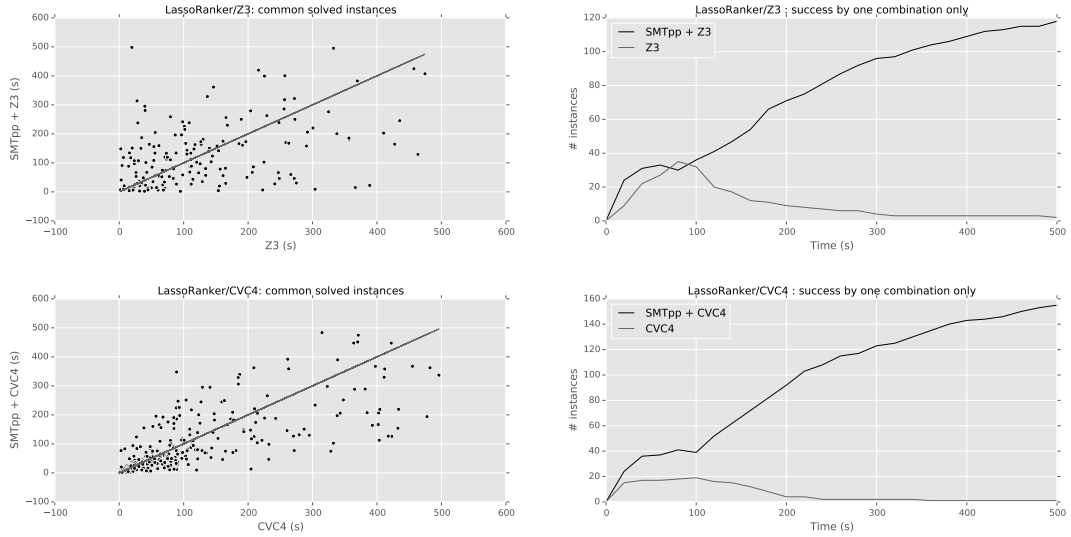

Figure 6: CVC4 and Z3 with or without SMTpp in LassoRanker

5 Conclusion

We have presented SMTpp, a preprocessor and analyzer for SMT-LIB, detailing its current features and performances. We think it is already usable by third-party, though with caution. Benchmarks processed with other tools seem to confirm the current features. The analyses which are currently working could already be used as post-processor for tools that automatically generate SMT-LIB scripts to remove unnecessary definitions or check that the logic declared is not too over-inflated.

Further developments on our roadmap include, but are not limited to:

Simplifications Other simplifications are already known to enhance the performances of SMT-solver, such as symmetry-related simplifications [14] or if-then-else conversions [17]. The latter is an especially interesting goal.

Script transformations Delta-debuggers [8, 18] and scramblers [2] are in the toolbox of every solver developer. They involve source-to-source tranformations that could be implemented as modules of SMTpp. We have for example started to work on an obfuscator and an incremental to non-incremental mode transformer.

Types and polymorphism The current SMTpp lacks a typer: its implementation is one of our priorities in order to enhance the precision of analyses and normalizations performed. The polymorphic notations and monomorphization process presented in [6] are a good candidate to be included in SMTpp. This would avoid the need for SMT-solvers to implement support for this feature.

veriT support Some preliminary but very alpha work has been done on embedding SMTpp into veriT. The crucial forthcoming steps are linking SMTpp's AST to veriT's internal DAG representation and evaluating the performance cost of such a deeper embedding.

Towards a better framework The current version of SMTpp is simple but could already support some refactoring towards making a platform for preprocessors and analyzers of SMT-LIB. We would like to explore having dynamic plug-ins, as permitted for example by the C analyzers platform Frama-C [10].

Tactic language A more distant, and harder, but exciting challenge to tackle is to include some means to guide SMT-solvers from SMTpp, as presented by Mendonça de Moura and Passmore [13].

Acknowledgments. We would like to thank the anonymous reviewers, as well as Pascal Fontaine for stimulating this project and for many fruitful discussions.

References

[1] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd Int'l Conf. Computer Aided Verification (CAV)*, volume 6806 of *LNCS*, pages 171–177, 2011.

[2] C. Barrett, M. Deters, L. de Moura, A. Oliveras, and A. Stump. 6 years of SMT-COMP. *Journal of Automated Reasoning*, 50(3):243–277, 2013.

[3] C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[4] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

[5] F. Bobot, S. Conchon, E. Contejean, M. Iguernelala, S. Lescuyer, and A. Mebsout. The Alt-Ergo Automated Theorem Prover, 2008. `http://alt-ergo.lri.fr/`.

[6] R. Bonichon, D. Déharbe, and C. Tavares. Extending SMT-LIB v2 with λ-terms and polymorphism. In P. Rümmer and C. M. Wintersteiger, editors, *Proc. 12th Int'l Workshop on Satisfiability Modulo Theories (SMT)*, volume 1163 of *CEUR Workshop Proceedings*, pages 53–62. CEUR-WS.org, 2014.

[7] T. Bouton, D. C. B. de Oliveira, D. Déharbe, and P. Fontaine. veriT: An open, trustable and efficient SMT-solver. In R. A. Schmidt, editor, *Proc. 22nd Int'l Conf Automated Deduction - CADE-22*, volume 5663 of *LNCS*, pages 151–156, 2009.

[8] R. Brummeyer and A. Biere. Fuzzing and Delta-Debugging SMT Solvers. In *Proc. 7th Int'l Workshop on Satisfiability Modulo Theories (SMT)*, page 5, 2009.

[9] D. R. Cok. jSMTLIB: Tutorial, Validation and Adapter Tools for SMT-LIBv2. In *Proc. 3rd Int'l Conf. on NASA Formal Methods*.

[10] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C - A software analysis perspective. In G. Eleftherakis, M. Hinchey, and M. Holcombe, editors, *Proc 10th Int'l Conf. Software Engineering and Formal Methods (SEFM)*, volume 7504 of *LNCS*, pages 233–247, 2012.

[11] P. Cuoq, J. Signoles, P. Baudin, R. Bonichon, G. Canet, L. Correnson, B. Monate, V. Prevosto, and A. Puccetti. Experience report: OCaml for an industrial-strength static analysis framework. In G. Hutton and A. P. Tolmach, editors, *Proc. 14th ACM SIGPLAN Int'l Conf on Functional programming (ICFP)*, pages 281–286, 2009.

[12] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Proc 14th Int'l Conf on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *LNCS*, pages 337–340, 2008.

[13] L. M. de Moura and G. O. Passmore. The strategy challenge in SMT solving. In M. P. Bonacina and M. E. Stickel, editors, *Automated Reasoning and Mathematics - Essays in Memory of William W. McCune*, volume 7788 of *LNCS*, pages 15–44, 2013.

[14] D. Déharbe, P. Fontaine, S. Merz, and B. W. Paleo. Exploiting symmetry in SMT problems. In N. Bjørner and V. Sofronie-Stokkermans, editors, *Proc. 23rd Int'l Conf Automated Deduction (CADE-23)*, volume 6803 of *LNCS*, pages 222–236, 2011.

[15] D. Delahaye, C. Dubois, C. Marché, and D. Mentré. The BWare project: Building a proof platform for the automated verification of B proof obligations. In Y. A. Ameur and K. Schewe, editors, *Proc 4th Int'l Conf. Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ)*, volume 8477 of *LNCS*, pages 290–293, 2014.

[16] N. Eén and A. Biere. Effective preprocessing in sat through variable and clause elimination. In *Proc. 8th Int'l Conf. on Theory and Applications of Satisfiability Testing*, SAT'05, pages 61–75, Berlin, Heidelberg, 2005.

[17] H. Kim, F. Somenzi, and H. Jin. Efficient Term-ITE Conversion for Satisfiability Modulo Theories. In O. Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009*, volume 5584 of *Lecture Notes in Computer Science*, pages 195–208. Springer Berlin Heidelberg, 2009.

[18] A. Niemetz and A. Biere. ddSMT: A Delta Debugger for the SMT-LIB v2 Format. In *Proc. 11th Int'l Workshop on Satisfiability Modulo Theories (SMT)*, pages 36–45, 2013.

[19] F. Pottier and Y. Régis-Gianas. Menhir, Dec. 2005.