A Pythonic examination of "To Ackermann-ize or not to Ackermann-ize?"

Anish Bhobe

January 29, 2023

Abstract

Satisfiability Modulo Theories (SMT($\mathcal{T}$)) is the problem of deciding the satisfiability of a formula with respect to a given background theory $\mathcal{T}$. In practical use, such theories are combination of multiple simpler theories. There are primarily two approaches to solving SMT($\mathcal{EUF} \cup \mathcal{T}$). First is to use Delayed Theory Combination (DTC) schema and the other is using Ackermann Expansion. In this report we re-examine the paper by Bruttomesso et al.[3] and attempt to implement the algorithms in Python as well as reproducing the results.

1 Introduction

SMT($\mathcal{T}$) solvers check the satisfiability of a First-Order Logic (FOL) formula with respect to some given FOL theory $\mathcal{T}$. Some commonly found FOL theories are Equality and Uninterpreted Functions ($\mathcal{EUF}$), Linear Arithmentic over Rational or Integer numbers (LRA and LIA respectively) etc. However, in general use cases, we find theories that are unions of two or more such theories such as Quantifier Free Equality, Uninterpreted Functions and Linear Integer and Real Arithmetic (QF_UFLIRA) which is the quantifier free union of $\mathcal{EUF}$, LRA and LIA. While there are many SMT($\mathcal{T}$) solvers such as Z3[5], MATHSAT[4] etc, they generally use similar generalized state of the art approaches such as Delayed Theory Combination (DTC) to solve such theory combination problems. However, in even of a problem of SMT($\mathcal{EUF} \cup \mathcal{T}$), we can use Ackermann expansion[2] (hereby ACK) to eliminate all instances of the $\mathcal{EUF}$ theory, simplifying the problem to an SMT($\mathcal{T}$) problem.

Bruttomesso et al.[3] detail the DTC approach as well as the Ackermann approach to solve the SMT($\mathcal{EUF} \cup \mathcal{T}$) problem and analyse the performance over QF_UFLRA problems. The performance over the two methods varies drastically depending on the specific problem, however - the Bruttomesso et al. propose two heuristics to offline pre-process the formula for better total performance.

2 Existing Approaches

2.1 Delayed Theory Combination

Delayed Theory Combination solves the SMT($\mathcal{T}_1 \cup \mathcal{T}_2$) problem by separating the atoms in the terms based on the respective theories. For example for an atom $f(x) \leq y$ belonging to $\mathcal{EUF} \cup LRA$, the purified formula is of the form $f(x) = v_{f(x)} \wedge v_{f(x)} \leq y$ such that $f(x) = v_{f(x)}$ belongs to $\mathcal{EUF}$ and $v_{f(x)} \leq y$ belongs to LRA. This process is known as purification.

For each such pure formula, the variables that exist in both the theories ($v_{f(x)}$) in the example above are known as interface variables. We split the formula into two sets - $\mu_{\mathcal{T}_1}$ and $\mu_{\mathcal{T}_2}$ which contains the terms belonging to $\mathcal{T}_1$ and $\mathcal{T}_2$ respectively. We also add μ_{ie} as the interface equalities formed by pair-wise equalities between interface variables.

They also define a function $\mathcal{T}2\mathcal{B}$ such that $\mathcal{T}2\mathcal{B}(\varphi) \mapsto \varphi^p$ where φ^p is the propositional equivalent of φ and the reverse function $\mathcal{B}2\mathcal{T}(\varphi^p) \mapsto \varphi$.

Thus, we can use boolean-satisfiability on φ^p in order to check unsatisfiability as $UNSAT(\varphi^p) \implies UNSAT(\varphi)$. While φ^p is boolean satisfiable, we apply CDCL to find truth values for $\mu_{\mathcal{T}_1}$, $\mu_{\mathcal{T}_2}$ and μ_{ie} and check consistency of each part in their own theory along with the interface equalities.

$$SAT(\varphi) \iff \mathcal{T}_1 - consistent(\mu_{\mathcal{T}_1} \cup \mu_{ie}) \wedge \mathcal{T}_2 - consistent(\mu_{\mathcal{T}_2} \cup \mu_{ie})$$

In case of inconsistencies, the negation of the counter-example is added to φ^p. This continues until φ^p is UNSAT or we find a consistent φ.

The intuition here is that if the same set of truth value assignments is consistent in both theories, then the truth assignment is consistent in the conjunction of the theories, and thus can satisfy the problem.

2.2 Ackermann Expansion

Ackermann Expansion (ACK) removes $\mathcal{EUF}$ functions from the formula by replacing it with the equality constraint. From $\mathcal{EUF}$ we know the following.

$$x = y \implies f(x) = f(y)$$

$$\bigwedge_{i=0}^{arity} x_i = y_i \implies f(x_0, ..., x_{arity}) = f(y_0, ..., y_{arity})$$

For example in case of a simple formula below

$$f(x) = a \wedge f(y) = b \wedge f(z) = c$$

we replace all function applications with their variables and add the conjunction of ackermann equalities produced for each unique pair of function applications.

$$v_x = a \ \wedge \ v_y = b \ \wedge \ v_z = c \ \wedge \ (x = y \implies v_x = v_y) \ \wedge \ (x = z \implies v_x = v_z) \ \wedge \ (y = z \implies v_y = v_z)$$

By using the Ackermann Expansion method, we can eliminate the $\mathcal{EUF}$ terms and replace them with variables and equalities. Thus converting the SMT($\mathcal{EUF} \cup \mathcal{T}$) problem to an SMT($\mathcal{T}$) problem.

3 Decision Strategies to choose between DTC and ACK

Bruttomesso et al. analysed the performance of MathSAT[4] using DTC, which is the default implementation of SMT($\mathcal{EUF} \cup \mathcal{T}$) solver on MathSAT in comparison to a pre-processed Ackermann expansion. The execution time of the solver correlated with the boolean search space in the propositional formula φ^p.

Thus the heuristic used to decide the method to be used was based upon the minimization of the additional boolean search space that is created by each of the methods.

The boolean search space expansion caused by DTC depend on the number of atoms generated by the interface equalities. For $|V|$ interface variables, the upper bound on the number of atoms generated is $\frac{|V|\times(|V|-1)}{2}$

Similarly, in Ackermann expansion, the search space expansion depends on the number of equalities generated by the implications. Thus, for each function f, the number of equalities generated is $\frac{|N|\times(|N|-1)}{2} \times (arity(f) + 1)$. Here, the arity of f denotes the equalities on the LHS and the 1 denotes the equalities between the introduced variables on the RHS of the implication.

3.1 Decide

The first procedure introduced was a simple selection criteria between the two methods. The DECIDE algorithm simply compares the number of introduced equalities and choses the one that is minimum.

$$Decide(\varphi) = \begin{cases} ack(\varphi) & equalities(ack(\varphi)) < equalities(purify(\varphi)) \\ purify(\varphi) & otherwise \end{cases}$$

3.2 Partial

An all or nothing effort such as Decide would not lead to the most effective minimization. Instead Bruttomesso et al. propose the Ackermann expansion of some terms such that the number of equalities in minimized. However, finding such global minima is slow and thus they instead use a greedy procedure to find a local minima called Partial.

The method generally follows the pseudocode in 1 and 2. Partial takes a purified function and then finds a group of functions such that ackermannizing them provides largest reduction in equalities (gain).

For every interface variable, a set of functions is found which cause the variable to be an interface variable. Then, for each such group of functions G, the gain is defined as

$$gain(G, \varphi) = ackermannEqualitiesIntroduced(G, \varphi) - interfaceEqualitiesReducedByAckermannization(G, \varphi)$$

Thus by finding the groups with max gain (if positive) and ackermannizing them, we reduce the number of equalities at every step until no such group can be found (gain ≤ 0)

Algorithm 1 Partial

```
1: function PARTIAL(φ)
2:     ψ ← purify(φ)
3:     setA ← ∅
4:     do
5:         setB ← FindFunctionsToAck(ψ)
6:         ψ ← ackermannizeFunctions(ψ, setB)
7:         setA ← setA ∪ setB
8:     while setB ≠ ∅
9:     φ' ← ackermannizeFunctions(φ, setA)
10:    return φ'
11: end function
```

Algorithm 2 FindFunctionsToAck

```
1: function FINDFUNCTIONSTOACK(ψ)
2:     V ← interfaceVariables(ψ)
3:     G ← dict()
4:     for all v ∈ V do
5:         G ← getFunctionsContaining(ψ, v)
6:         G[G] ← G[G] ∪ {v}
7:     end for
8:     for all g, v ∈ G do
9:         ie ← countInterfaceEqualities(ψ, v)
10:        ack ← countAckermannEqualities(ψ, g)
11:        gain[g] ← ie − ack
12:    end for
13:    group, gain ← findGroupWithMaxGain(gain)
14:    if gain > 0 then
15:        return group
16:    else
17:        return ∅
18:    end if
19: end function
```

4 Implementation and Experimentation

The pre-processing steps as well as the decision strategies in the paper were re-implemented in *Python* using *PySMT*[6] library. The internally provided *Walker*s were extended used to purify and ackermannize the formulae. During this implementation however, certain simplifications were made which impact the efficiency of the system.

- Ackermannization does not remove redundant terms and implications.
- The gain is calculated on the upper bounds instead of the actual counts to improve performance.

Type	aggregation	ACK / DTC	DECIDE / DTC	PARTIAL / DTC
Coupled	mean	136.7102642	121.0243507	1.019782695
Coupled	median	11.79818787	11.41459875	0.9919459046
Deoupled	mean	59.99361813	33.92411728	0.9726368695
Decoupled	median	3.809595068	3.86830752	0.9763517788

Table 1: Benchmark times of all the MathSAT problems w.r.t MathSAT Solver

The implementation was tested on the QF_UFLRA MathSAT benchmarks in SMT-LIB-benchmarks[1] repository. And the preprocessing steps as well as the assertion additions into the SMT solver were completed outside the benchmark timings in order to minimize the impact of the speed of Python on the actual execution.

4.1 Results

The data recorded from the benchmark cases on the PySMT implementation is shown in 1. We see that the *Partial* strategy is faster (median) than the default DTC method in MathSAT solver. For RandomCoupled benchmarks, the median has a minor 0.81% improvment, while for RandomDecoupled benchmark, the *Partial* is faster than DTC on median by 2.4%. While these tests are not comprehensive or fair (due to the use of different languages), they show a general trend that even a relatively simple/simplified *Partial* strategy heuristic provides generally faster results.

4.2 Threats to Validity

The re-implementation of the pre-processors is done in Python instead of a faster, low level language such as C++ leads to slowdowns and constraints over what parts can be timed. As the DTC preprocess takes place in C++ for all the four methods while Ackermannization uses python for the ACK and PARTIAL schemas leads to difference in performance. As such all the boilerplate such as decision process between Purify and ACK has been completed ahead of time in order to ensure that the measured elapsed time contains as little of Python lines as possible.

However the serialization of the PySMT Formula DAG to the input formats for the solvers is slightly slower and memory intensive for larger formulae. During exceptional consumption of the memory on the test device, the MathSAT DTC scheme solver consistently outperformed the *Partial* scheme.

5 Conclusion

Through this report we look at the different methods use to solve an SMT$\mathcal{EUF} \cup \mathcal{T}$ such as the more generic DTC schema and the Ackermann Expansion to eliminate the $\mathcal{EUF}$ terms. We also look at the strategies such as *Decide* to decide between usage of DTC and ACK, and the *Partial* strategy to selectively Ackermannize certain functions in order to minimize the state space expansion and improve performance.

We also re-implement and benchmark the methods using the MathSAT solver as a back-end and show that for the most cases, *Partial* outperforms *DTC*. Through understanding of the PySMT code, the possible source of the performance issue comes python's slower conversion from the *PySMT.FNode* to the requires SMT2 string for the actual solver.

References

[1] Smt-lib-benchmarks · gitlab.

[2] Ackermann, W. Solvable cases of the decision problem. *The Mathematical Gazette 39* (01 1962).

[3] Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Santuari, A., and Sebastiani, R. To ackermann-ize or not to ackermann-ize? on efficiently handling uninterpreted function. pp. 557–571.

[4] Cimatti, A., Griggio, A., Schaafsma, B. J., and Sebastiani, R. The mathsat 5 smt solver .

[5] De Moura, L., and Bjørner, N. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 2008), TACAS'08/ETAPS'08, Springer-Verlag, p. 337–340.

[6] Micheli, A., and Gario, M. Pysmt/pysmt: Pysmt: A library for smt formulae manipulation and solving.