Towards Producing Shorter Congruence Closure Proofs in a State-of-the-art SMT Solver (Extended Abstract)

Bruno Andreotti[1], Haniel Barbosa[1] and Oliver Flatt[2]

[1]Universidade Federal de Minas Gerais (UFMG), Belo Horizonte, Brazil
[2]University of Washington, Seattle, USA

Abstract

The satisfiability of constraints in the theory of equality and uninterpreted functions, a core component of SMT solving, can be decided via a congruence closure algorithm. Classical congruence closure algorithms can provide machine checkable proofs when explaining why terms are equal. Recently, Flatt et al. at FMCAD'22 presented a modified congruence closure algorithm producing demonstrably shorter proofs. The algorithm was implemented and evaluated in the context of equality saturation tools. In this extended abstract we present initial work on implementing this new congruence closure algorithm in cvc5, a state-of-the-art SMT solver. We discuss the challenges from adapting a high-performance implementation and how to address them effectively. We evaluate the implementation on a large set of SMT-LIB benchmarks, and demonstrate experimentally how this new algorithm results in smaller proofs in real-world scenarios. Moreover, we evaluate the performance impact of the new algorithm and determine whether the smaller explanations that can be generated from the shorter proofs are worth the performance overhead of the more expensive algorithm.

Keywords

SMT solvers, Congruence closure, SMT proofs

1. Introduction

Proof-producing SMT solvers have a large potential for increasing the trustworthiness of formal methods applications that rely on solvers to discharge proof obligations. SMT solvers however can produce large proof certificates that can be challenging to check quickly, specially in tools with limited performance, such as formally verified checkers or proof assistants. One way to mitigate this issue is to employ solving techniques that can lead to shorter proofs within the same proof calculus, which will therefore be simpler to check. Flatt et al. [1] proposed such a procedure for the classical congruence closure algorithm in the context of equality saturation. In this paper we present an initial implementation of this approach in the congruence closure engine of the state-of-the-art SMT solver cvc5 [2]. Extending the core component of state-of-the-art SMT solvers is notoriously challenging [3], and we discuss the several challenges specific to this setting that we had to face, such as considering backtracking and avoiding cyclic explanations. We also present an evaluation of the current implementation, which indicates a number of future directions of improvements and some encouraging proof compression rates in proofs from the congruence closure algorithm.

2. Context

Congruence is the property of equality that states that if two terms a and b are equal, then, for any function f, $f(a) = f(b)$. For a given set of equalities, we say that the corresponding congruence closure is the minimal equivalence relation that satisfies these equalities as well as the properties of reflexivity, symmetry, transitivity and congruence.

Efficient algorithms for computing congruence closure were first described by Downey et al. [4] and Nelson and Oppen [5], where the main application was determining the equivalence of expressions

PAAR'24: 9th Workshop on Practical Aspects of Automated Reasoning, July 2, 2024, Nancy, France
bandreotti@dcc.ufmg.br (B. Andreotti); hbarbosa@dcc.ufmg.br (H. Barbosa); oflatt@cs.washington.edu (O. Flatt)
0000-0003-0345-6495 (B. Andreotti); 0000-0003-0188-2300 (H. Barbosa); 0000-0002-0656-235X (O. Flatt)

in a program, for the purpose of program verification. This algorithm is based on a *Union-find* data structure [6], where each term is a vertex in a graph (called the *equality graph*, or *e-graph*), and each equivalence class forms a rooted tree, whose root is called the class *representative*. The graph is built by processing equalities sequentially. At the start, each term is part of an equivalence class containing only itself. When the algorithm processes an equality $a = b$, if the terms are not already equivalent, it must *merge* their equivalence classes. First it finds the representatives of a and b and then adds an edge between them, with one of them becoming the representative of the merged class.

Then, the algorithm must merge all classes that have become equivalent due to the congruence property. For example, when the $a = b$ is asserted, the terms $f(a)$ and $f(b)$ are now also equivalent, so the algorithm must also merge their classes. We call edges added in this step *congruence edges*, and the terms that caused them to be added their *justification*. In our example, the congruence edge that merged the classes of $f(a)$ and $f(b)$ will have the terms $\{a, b\}$ as its justification.

Once the equality graph is built, whether two terms are equal is determined by finding their class representatives and checking if they are the same. However, for many applications, responding to these queries with a "yes" or "no" answer is not sufficient. In some cases, like for SMT solvers, an *explanation* of why two terms are equal is also necessary. These explanations take the form of a set of equalities sufficient to ensure that the terms are equivalent. In SMT solvers, producing explanations is crucial since they are used to build conflict clauses in the solver. In the CDCL($\mathcal{T}$) algorithm [7] used by most SMT solvers, the solver will make a series of assumptions and use the theory solvers to determine whether they are consistent. If they are not, the theory solver will provide a conflict clause — a subset of the assumptions that is by itself inconsistent, which is used to prune part of the search space. Note that a smaller clause will increase the pruning, and is thus more useful to the solver. Thus, a congruence closure algorithm that can provide explanations (and ideally, short explanations) is crucial for a high performance SMT solver. However, it is well known that finding the smallest possible explanation for the equivalence of two terms is an NP-complete problem [8].

Congruence closure and proofs. Some congruence closure algorithms are also capable of producing *proofs*. While an explanation is just a set of equalities, a proof is a derivation that uses these equalities and the properties of reflexivity, symmetry, transitivity and congruence to show that the two terms are equivalent.

Proofs from a congruence closure algorithm are also useful for SMT solvers, since they allow the solver to provide a proof for the unsatisfiability of a formula as a whole. Proof-producing SMT solvers have become increasingly important in the last few years, and are already used in many applications [9, 10, 11, 12, 13]. As an example, they are used for proof automation in interactive theorem provers [14]. In this technique, SMT solvers are used to prove a theorem or a lemma in an interactive theorem prover, by showing that its negation is unsatisfiable. Then, they must produce a well-formed proof that will be reconstructed in the langauge of the theorem prover. Since congruence closure reasoning will often be a part of this proof, the congruence closure algorithm used by this SMT solver must be able to produce well formed proofs.

A proof-producing congruence closure algorithm is described by Nieuwenhuis et al. in [15]. When asked to explain two equivalent terms, this algorithm traverses the path between them in the equality graph. When encountering a non-congruence edge, the algorithm simply stores the equality that caused that edge to be merged. When encountering a congruence edge with justification $\{j_1, j_2\}$, the algorithm recursively explains the equivalence of the terms j_1 and j_2, and adds all returned equalities to the explanation. Figure 1 shows the pseudocode for this algorithm.

While the explanations returned by this algorithm are minimal, they are not necessarily *minimum*, meaning a shorter explanation may often exist. A property of these algorithms that allows this to happen is the fact that they ignore equalities between terms that were already determined to be equal (called *redundant equalities*). For example, if an algorithm has already determined that the terms a and b are equivalent due to some long series of equalities, if it then encounters the equality $a = b$, it will discard it. Later, when queried for an explanation of these terms' equivalence, it would provide a long

```
function get_explanation(start, end):
    let explanation = []
    let lca = find_lowest_common_ancestor(start, end)
    explanation += explain_along_path(start, lca)
    explanation += explain_along_path(end, lca)
    return explanation

function explain_along_path(lower, upper):
    let explanation = []
    let current = lower
    while current != upper:
        let edge = current.edge_to_parent()
        if edge.is_congruence_edge():
            let (j1, j2) = edge.justification()
            explanation += get_explanation(j1, j2)
        else:
            explanation += edge
        current = current.parent()
    return explanation
```

Figure 1: Pseudocode for a classical proof-producing explanation algorithm. The function `get_explanation` returns a list with the explanation for the equivalence of two terms. Since the equivalence class is represented by a rooted tree, the function works by explaining the path from each of the nodes to their lowest common ancestor. This pseudocode can be straightforwardly extended to produce structured proofs.

set of equalities, where the singleton explanation $\{a = b\}$ would suffice. Observing this limitation, Flatt et al. [1] developed two new congruence closure algorithms, called TreeOpt and Greedy, that do not discard redundant equalities, and instead use them to provide shorter proofs and potentially smaller explanations.

When encountering a redundant equality $a = b$, these algorithms add the edge (a, b) to the equality graph, not triggering any merges (as the terms were already in the same equivalence class). Now, each equivalence class is not necessarily a tree anymore, so there may be more than one path between two terms in it. Naturally, when queried for an explanation, the goal of these algorithms is to find the path between the terms that represents the shortest proof. This is where the strategies used by the two algorithms diverge. In the following sections, we give a high-level description of the two algorithms introduced by Flatt et al. [1], but a detailed description, as well as pseudocode, can be found in the original paper.

2.1. TreeOpt

As mentioned earlier, finding the path between two equivalent terms that results in the smallest proof is an NP-complete problem [8]. This is in part because the equalities used as premises to explain a congruence edge might also be reused to explain other edges in the path, meaning the best path to take depends on the edges you already took. If we instead assume that equalities in the proof cannot be shared (and instead have to be duplicated), the problem becomes tractable. This new definition of proof size is called the proof's *tree size*, whereas the original definition is called its *DAG size*[1].

The TreeOpt algorithm is a polynomial algorithm for finding the explanation with optimal tree size, hence the name. It does this by first attributing a weight to each edge, defined as the tree size of the proof for that edge. For non-congruence edges this value is always 1, and for congruence edges it is the tree size of the proof of its justification. Since the weight of a congruence edge might depend on the weight of other congruence edges, the algorithm needs to recompute these weights until it reaches a fixed point.

Once the weight of all edges is computed, finding the optimal explanation amounts to simply finding

[1]This refers to the fact that, if we allow this sharing of equalities, the proof is represented by a directed acyclic graph (DAG), whereas if we do not allow sharing, it is represented by a tree.

the shortest path between the two terms, taking care to recursively explain the justification of every congruence edge encountered.

2.2. Greedy

The second algorithm presented by Flatt et al. [1] is a greedy algorithm that attempts to find a small explanation without incurring an increase in complexity. To do this, the algorithm first computes an estimate of the proof size for each edge in the equality graph. This estimate is obtained by computing the proof size of the edge with the traditional congruence closure algorithm, that is, ignoring redundant equalities.

Once the estimates are calculated, the algorithm simply finds the shortest path between the terms, using the estimates as weights for the edges. The algorithm is parameterized by a *fuel* parameter, which, similarly to [1], we set to 10 as a default. When it encounters a congruence edge, if the fuel is greater than 0, the algorithm recurses (and decrements its fuel) to explain the justification of the congruence edge. However, if the fuel is 0, the algorithm explains the justification by simply using the classical congruence closure algorithm, by finding a path which ignores redundant equalities.

3. Implementation

We implemented these new congruence closure algorithms in cvc5 [2], a state-of-the-art SMT solver. The existing theory solver for the theory of equality and uninterpreted functions, called its *equality engine*, implements a version of a proof-producing congruence closure algorithm as seen in Nieuwenhuis et al. [15]. As part of this work, we adapted this existing implementation to not discard redundant equalities, and implemented the TreeOpt and Greedy explanation algorithms. In this section, we highlight the main challenges that needed to be addressed to implement these new algorithms in cvc5.

3.1. Context-dependence

A big challenge in adapting the algorithms in Flatt et al. [1] to work in a modern SMT solver was dealing with the context-dependent nature of the solver. The CDCL($\mathscr{T}$) algorithm, used by most modern SMT solvers, solves problems by testing many possible (partial) solutions and backtracking as soon as a solution is determined to be invalid.

This means that, from the viewpoint of the equality engine, the SMT solver will assert a series of equalities, query the equality engine for explaining the reason why a given equality holds, and, if the asserted equalities are found to be inconsistent, revert to a previous state. This means that the equality engine must be able to efficiently produce explanations for the equivalence of any two terms, while also at any point being able to backtrack to a previous state. To do that efficiently, the existing implementation carefully records the steps taken by the congruence closure algorithm, and structures the equality graph so that it can revert those steps, un-merging any equality classes that were merged since the respective backtracking point.

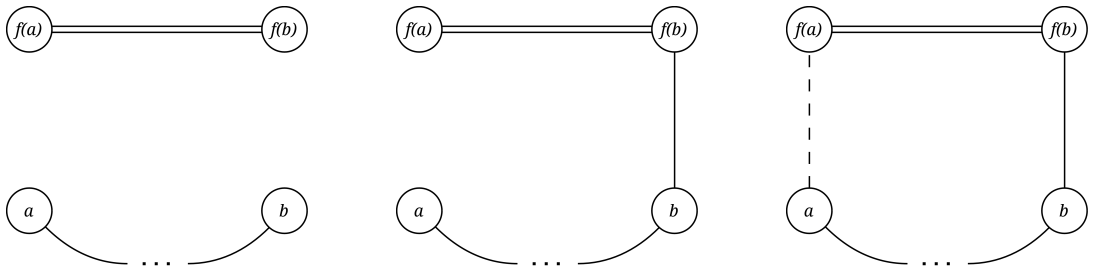
Keeping redundant equalities complicates this process. For example, when backtracking, the equality engine holds an invariant that the number of edges is the same as the number of class merges — since we modified the code to add edges that do not cause merges, we had to add "dummy" entries to the class merges vector in order to keep this invariant. Since some data is cached between calls to `get_explanation` (for example, the edge weights computed for the TreeOpt algorithm), we also had to make sure that no invalid cached data survives after a backtrack. Otherwise, a call to `get_explanation` might use cached data from when the equality graph was in a different state, which might result in an invalid or suboptimal proof.

3.2. Avoiding circular explanations

Since the equality engine in cvc5 is context-dependent, it is useful to think of the state of the equality graph in terms of *levels*. The current level of the equality engine is defined as the number of equalities that were asserted, and for every equality that is backtracked, the level is decreased by 1.

Importantly, when explaining the justification of a congruence edge that was added in level n, we cannot use any redundant edges that were added in levels higher than n, otherwise the explanation might use that congruence edge itself. Consider the following example:

At level n, the term a is in the same equality class as b, separated by a long path of edges. Since a and b are equivalent, there is a congruence edge between $f(a)$ and $f(b)$, whose justification is $\{a, b\}$. In the next level, $n + 1$, the equality $b = f(b)$ is asserted, merging the equivalence classes of these two terms. At level $n + 2$, the redundant equality $a = f(a)$ is asserted, and the corresponding redundant edge is added. Then, the explanation for $f(a) \equiv f(b)$ is queried. When explaining the congruence edge between $f(a)$ and $f(b)$, the algorithm must explain its justification, that is, find a path between a and b. However, the shortest path between these terms will use the $(f(a), f(b))$ edge itself, resulting in a circular explanation.



(a) Example equality graph at level n.

(b) Example equality graph at level $n + 1$.

(c) Example equality graph at level $n + 2$.

Figure 2: An example of an equality graph at different levels, showing how a circular explanation may appear. Simple lines are regular equality edges, double lines are congruence edges, and dashed lines are redundant equality edges.

To prevent this, we store for each edge the level in which it was created. We also added a `max_level` parameter to `get_explanation`, and changed the algorithms to ignore edges whose level is bigger than the maximum level. When explaining a congruence edge with justification $\{j_1, j_2\}$, `max_level` is set to the level in which j_1 and j_2 were determined to be equivalent. In this way, no circular explanations can be constructed.

4. Evaluation

In order to evaluate the effectiveness and performance of our implementation, we tested it against a large set of benchmarks. More specifically, we used the 7502 SMT problems in the QF_UF logic from the SMT-LIB benchmarks library [16], an industry-standard set of benchmarks for SMT solvers. The results were generated with a cluster equipped with 16 x Intel(R) Xeon(R) CPU E5-2637 v4 @ 3.50GHz, 62.79 GiB RAM machines, with one core per solver/benchmark pair, 300s time limit, and 8 GB memory limit.

We compare the two newly implemented algorithms, TreeOpt and Greedy, with the classical congruence closure explanation algorithm, from here on referred to as Vanilla. We ran cvc5 on each benchmark, using each of the three algorithms. We recorded the time taken to solve each problem, as well as the size of the final proof. These results are shown in Section 4.1

The explanations returned by the congruence closure algorithm inform the search of the SMT solver, which in turn changes which equalities are asserted to the equality engine, and the explanations that are requested. Therefore, simply looking at the final proof size is not the most direct way to compare the different algorithms. To get a more direct comparison, we instrumented cvc5 to, on each individual call

to `get_explanation`, run all three algorithms at the same time and record the size of each explanation returned. These results are shown in Section 4.2.

4.1. Global impact results

Figure 3 shows, for the Greedy and TreeOpt algorithms, a boxplot of their overhead when compared to the vanilla algorithm for congruence closure explanations. To facilitate visualization, each boxplot excludes the top and bottom 1% of data, as they are mostly outliers. The overhead for Greedy was 32x in the worst case, 0.16x in the best case, and 1.18x on average. As for TreeOpt, the overhead was 172x in the worst case, 0.46x in the best case, and 2.68x on average.

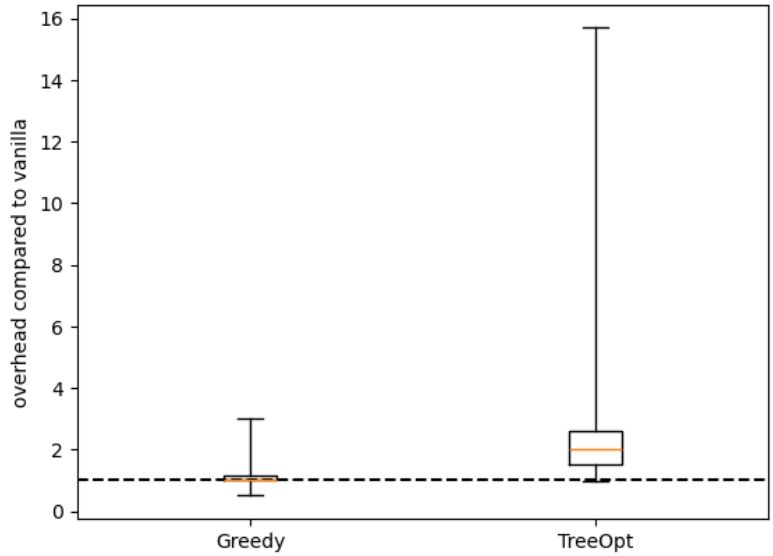


Figure 3: Boxplot of runtime overhead of Greedy and TreeOpt, when compared to Vanilla. The dashed line marks an overhead of 1, that is, the baseline runtime of Vanilla.

This data shows that, as expected, the TreeOpt algorithm is much less efficient than Vanilla. On the other hand, the performance of the Greedy algorithm is not as bad, representing on average an 18% increase in runtime over the Vanilla algorithm, and being in many cases faster than it. We believe theses cases are due to smaller conflict clauses being provided, thus pruning a larger portion of the search space, speeding up the solver.

As for proof size, we did not measure a significant impact of the new algorithms in the final proof size. On average, Greedy and TreeOpt proofs were respectively 0.2% and 0.1% larger than Vanilla proofs. In most cases, proofs from all three algorithms were almost exactly the same size. In the best cases, proofs from Greedy and TreeOpt were respectively 80% and 78% smaller than the corresponding Vanilla proofs. However, in the worst cases, the two new algorithms produced proofs that were up to 70% larger than the corresponding Vanilla proofs.

4.2. Local impact results

We also compared, in each call to `get_explanation`, the sizes of the proofs returned by each algorithm. In particular, for Greedy and TreeOpt, we are interested in the proof size ratio, that is, the size of the proof returned by the algorithm, divided by the size of the proof returned by the Vanilla algorithm.

For both algorithms, 83% of calls to `get_explanation` result in proofs that are the same size as the Vanilla proof (that is, the proof size ratio is 1). For both algorithms, the average proof size ratio was 0.98, and the minimum was 0.065. This means that while most proofs produced by these algorithms are almost the same size as the ones from Vanilla, in some cases they can be up to 93.5% smaller.

Looking at only the proofs whose size actually changed between algorithms (that is, excluding those where the proof size ratio is 1), the average proof size ratio was 0.86 for both algorithms, representing a reduction in proof size of 14%.

In rare occasions, the algorithms produced larger proofs than Vanilla. For Greedy, the worst proof size ratio was 2.2, and for TreeOpt it was 1.75. This is explained by the fact that these algorithms search for proofs with minimal tree size, while the final size of the proof is computed as its DAG size.

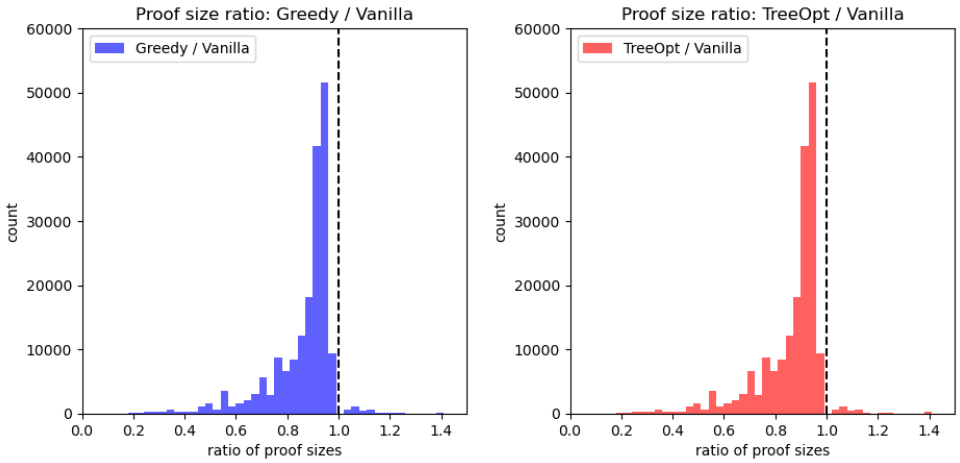


Figure 4: Histograms for the proof size ratios of the algorithms Greedy and TreeOpt, when compared to the proofs produced by Vanilla. To facilitate visualization, this excludes proofs where the proof size ratio was 1, that is, it only considers proofs where the size actually changed. The dashed line marks the excluded data. The graph cuts off at a ratio of 1.5 to improve visualization, but in both cases there was a small number of proofs with a proof size ratio bigger than 1.5.

Figure 4 shows a histogram of the proof size ratios for Greedy and TreeOpt, excluding proofs whose size did not change between algorithms. The graph shows how, when the proof size does change, it almost always decreases, albeit by a generally modest amount. Also, this data shows how the behaviour of the two algorithms is almost identical, indicating that the Greedy algorithm is in general a very good approximation for TreeOpt.

To get an idea of the overhead caused by keeping all redundant equalities, we also recorded, for each call to `get_explanation`, the number of redundant edges stored in the equality graph at that point. On average, 8.7% of the edges in the equality graph were redundant, and in the most extreme case, 44.63%. In around 0.8% of calls, there were 0 redundant edges.

5. Conclusion and future work

In conclusion, our work shows that congruence closure algorithms that keep redundant equalities can be efficiently implemented in a state-of-the-art SMT solver, with reasonable runtime overhead. Furthermore, we show that these techniques can have a measurable, even if modest, impact on proof size. However, there are still many avenues for future work that need to be explored. For one, a more extensive evaluation is necessary, investigating benchmarks from logics beyond QF_UF, and considering other metrics like impact on proof checking time. Also, new algorithms and heuristics that use this technique may be developed that present better performance, or that are better suited for use in an SMT solver. While it is too soon to tell whether this technique will be practical for use in SMT solvers, we hope that we were able to show that is has a lot of potential.

References

[1] O. Flatt, S. Coward, M. Willsey, Z. Tatlock, P. Panchekha, Small proofs from congruence closure, in: 2022 Formal Methods in Computer-Aided Design (FMCAD), 2022, pp. 75–83. doi:10.34727/2022/isbn.978-3-85448-053-2_13.

[2] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, Y. Zohar, cvc5: A versatile and industrial-strength SMT solver, in: D. Fisman, G. Rosu (Eds.), Tools and Algorithms for Construction and Analysis of Systems (TACAS), Part I, volume 13243 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 415–442. URL: https://doi.org/10.1007/978-3-030-99524-9_24. doi:10.1007/978-3-030-99524-9_24.

[3] H. Barbosa, A. Reynolds, D. E. Ouraoui, C. Tinelli, C. W. Barrett, Extending SMT solvers to higher-order logic, in: P. Fontaine (Ed.), Proc. Conference on Automated Deduction (CADE), volume 11716, Springer, 2019, pp. 35–54. URL: https://doi.org/10.1007/978-3-030-29436-6_3. doi:10.1007/978-3-030-29436-6_3.

[4] P. J. Downey, R. Sethi, R. E. Tarjan, Variations on the common subexpression problem, J. ACM 27 (1980) 758–771. URL: https://doi.org/10.1145/322217.322228. doi:10.1145/322217.322228.

[5] G. Nelson, D. C. Oppen, Fast decision procedures based on congruence closure, J. ACM 27 (1980) 356–364. URL: https://doi.org/10.1145/322186.322198. doi:10.1145/322186.322198.

[6] R. E. Tarjan, Efficiency of a good but not linear set union algorithm, J. ACM 22 (1975) 215–225. URL: https://doi.org/10.1145/321879.321884. doi:10.1145/321879.321884.

[7] R. Nieuwenhuis, A. Oliveras, C. Tinelli, Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t), J. ACM 53 (2006) 937–977. URL: http://doi.acm.org/10.1145/1217856.1217859. doi:10.1145/1217856.1217859.

[8] A. Fellner, P. Fontaine, B. W. Paleo, Np-completeness of small conflict set generation for congruence closure, Form. Methods Syst. Des. 51 (2017) 533–544. URL: https://doi.org/10.1007/s10703-017-0283-x. doi:10.1007/s10703-017-0283-x.

[9] H. Barbosa, A. Reynolds, G. Kremer, H. Lachnitt, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Viswanathan, S. Viteri, Y. Zohar, C. Tinelli, C. Barrett, Flexible proof production in an industrial-strength smt solver, in: Automated Reasoning: 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8–10, 2022, Proceedings, Springer-Verlag, Berlin, Heidelberg, 2022, p. 15–35. URL: https://doi.org/10.1007/978-3-031-10769-6_3. doi:10.1007/978-3-031-10769-6_3.

[10] H. Barbosa, C. Barrett, B. Cook, B. Dutertre, G. Kremer, H. Lachnitt, A. Niemetz, A. N ötzli, A. Ozdemir, M. Preiner, A. Reynolds, C. Tinelli, Y. Zohar, Generating and exploiting automated reasoning proof certificates, Commun. ACM 66 (2023) 86–95. URL: https://doi.org/10.1145/3587692. doi:10.1145/3587692.

[11] B. Andreotti, H. Lachnitt, H. Barbosa, Carcara: An efficient proof checker and elaborator for smt proofs in the alethe format, in: Tools and Algorithms for the Construction and Analysis of Systems: 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22–27, 2023, Proceedings, Part I, Springer-Verlag, Berlin, Heidelberg, 2023, p. 367–386. URL: https://doi.org/10.1007/978-3-031-30823-9_19. doi:10.1007/978-3-031-30823-9_19.

[12] J. Hoenicke, T. Schindler, A simple proof format for SMT, volume 3185 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2022, pp. 54–70. URL: http://ceur-ws.org/Vol-3185/paper9527.pdf.

[13] R. Otoni, M. Blicha, P. Eugster, A. E. J. Hyvärinen, N. Sharygina, Theory-specific proof steps witnessing correctness of SMT executions, in: Design Automation Conference (DAC), IEEE, 2021, pp. 541–546. URL: https://doi.org/10.1109/DAC18074.2021.9586272. doi:10.1109/DAC18074.2021.9586272.

[14] H. Schurr, M. Fleury, M. Desharnais, Reliable reconstruction of fine-grained proofs in a proof assistant, in: A. Platzer, G. Sutcliffe (Eds.), Proc. Conference on Automated Deduction (CADE), volume 12699 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 450–467. URL: https://doi.org/10.1007/978-3-030-79876-5_26. doi:10.1007/978-3-030-79876-5_26.

[15] R. Nieuwenhuis, A. Oliveras, Proof-producing congruence closure, in: J. Giesl (Ed.), Term Rewriting and Applications, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 453–468.
[16] C. Barrett, P. Fontaine, C. Tinelli, The Satisfiability Modulo Theories Library (SMT-LIB), www.SMT-LIB.org, 2016.