WhyUnsat: A Practical Explanation Tool

Robert Nieuwenhuis
Technical University of Catalonia, Barcelona, Spain

Albert Oliveras
Technical University of Catalonia, Barcelona, Spain

Enric Rodríguez-Carbonell
Technical University of Catalonia, Barcelona, Spain

Abstract

Hard industrial planning, timetabling or scheduling instances for SAT typically have many high-level constraints, each expressed by a possibly large number of clauses. When a given instance is reported unsatisfiable by the SAT solver, the user normally needs an *explanation* why: a (hopefully small) subset of the constraints causing it. Our industrial applications require *fast* explanations, preferably faster than the original SAT run. For this, we leverage the original solver's work through its unsatisfiability proof.

Here we introduce WhyUnsat[1], and explain why it is fast and robust. In WhyUnsat one can always plug in the best current SAT solver and proof trimmer, without any modification, by simply indicating the path to their executables. WhyUnsat is also fast because it exploits, via MPI, the -progressively cheaper- shared-memory and distributed computing resources. Another requirement we had is that the tool should be anytime and user-friendly; indeed, it quickly shows a human-readable presentation of (an over-approximation of) the explanation, which is then progressively reduced until minimality (unless interrupted by the user).

Finally, and not less importantly, here we explain how and why the WhyUnsat approach is now also directly applicable, at no implementation cost, to IPASIR-UP-based constraint programming by Lazy Clause Generation (LCG) as well as to SAT Modulo Theories (SMT).

2012 ACM Subject Classification Theory of computation

Keywords and phrases SAT, SMT, Constraint Programming, Lazy Clause Generation

Digital Object Identifier 10.4230/LIPIcs.SAT.2026.39

Category Tool Paper

Supplementary Material *Software (Source Code)*: `https://github.com/enric-rodriguez/WhyUnsat` [18]; archived at `swh:1:dir:68a66c6e39049a7035e472378224d089a41143f4`

Funding All authors are supported by grant PID2024-157044OB-C32, funded by MICIU /AEI /10.13039/501100011033 / FEDER, UE.

1 Introduction

Despite the fact that SAT solvers ultimately process problems in CNF, it is a widely accepted fact that *"There are no CNF problems"* [21]. Additionally, users of SAT technology require more than a simple yes/no answer: models, proofs, unsatisfiable cores, incremental usage or its integration into more complex systems via well-defined APIs are nowadays standard demands.

In our industrial experiences, e.g., with planning, timetabling or scheduling, SAT solvers are being used for handling instances with many high-level constraints, where each constraint can contribute a large number of clauses to the final CNF. Even though these problems can

[1] Available at `https://github.com/enric-rodriguez/WhyUnsat`

be large and hard, in case of unsatisfiability the user frequently still wishes to be able to quickly understand why, i.e., to quickly obtain an *explanation*, a (hopefully small) subset of the high-level constraints that is already unsatisfiable by itself.

▶ **Example 1.** Here we consider a resource-constrained project scheduling problem (RCPSP), where each task has its time window, and a duration that depends on which (single) machine it uses, exclusively and uninterruptedly. There are also precedence constraints of the form: "task i must end before task j starts". A typical RCPSP instance with 200 tasks (the first benchmark of our RCPSP set of Section 6) looks as follows (in Prolog-style). Here task 1 is in window 25-82, and takes 13h if on machine 1, 10h if on machine 4, and 9h on machine 5. Similarly, precedences state that task 60 must end before task 104 starts, 112 before 92, etc.

```
numMachines(6).
task(   1, 25, 82, [1-13, 4-10, 5-9] ).
...
before(  60, 104 ).
before( 112,  92 ).
...
```

This non-trivial 200-task RCPSP instance generates a CNF with (approx.) 22,000 variables and 1,5 million clauses. It takes KISSAT [9] 6 seconds (approx.) to return "unsatisfiable", even though there is a relatively simple explanation. This explanation is reported by WHYUNSAT in 6.2 seconds as follows (and later on proved minimal in 2.3 more seconds):

```
constraint:  numclauses:   meaning of the constraint:
          0        295414   basic RCPSP constraint.
        112           682   Task 112 has window 20-50 and machine-runtimes 3-10 6-12
        137           713   Task 137 has window 20-50 and machine-runtimes 3-11 6-12
        191           483   Task 191 has window 30-50 and machine-runtimes 3-11 6-12
        199           483   Task 199 has window 30-50 and machine-runtimes 3-11 6-12
        203        132803   No two tasks can run simultaneously on machine 3.
        206        160691   No two tasks can run simultaneously on machine 6.
        214           496   Task 112 ends before task 137 starts.
```

Constraint 0 here is typical in many industrial applications: a constraint describing the general problem. Such a "basic" constraint usually belongs to the final explanation, and, in fact, for efficiency reasons, a tool like WhyUnsat could directly assume that. However WhyUnsat (wlog.) does not do so; it makes no distinctions between constraints (since, efficiency-wise, it would essentially save just one easy(!) parallel SAT call, see below).

In principle, this problem can be solved by other existing tools, such as MUSER2 [7] or HAIFAHLMUC [20]. However, for the previous example, they do not generate any (intermediate or final) result in an hour runtime even when using the latest version of CADICAL [8] for MUSER2, which makes it much faster than its original version. These tools excel on other, quite different, intended uses, such as gate-level circuit analysis, as in the instances used at the SAT competition for group-oriented MUS (2011, the only one held, and won by HAIFAHLMUC), where each constraint represents a gate with typically only three clauses. However, in spite of applying several sophisticated techniques, these tools are not adequate for our industrial applications, with much harder and larger initial SAT instances and where a single constraint usually has many clauses. Indeed, they are not robust for our instances, frequently requiring extremely long runtimes, even for, say, 1-minute original SAT runs. To our understanding, this is because they rely on internal modifications of (nowadays somewhat outdated) SAT solvers (e.g., Haifa-HLMUC) and/or on the use of *indicator variables* that are well known to have a severe negative impact on solvers' performance (see our experiments on all this, below).

WhyUnsat is a different, conceptually simpler and rather pragmatic tool, with the aim to be robust for modern industrial applications. It requires no wrappers, indicator variables, or modifications of the underlying SAT solver, which only has to be able to do proof logging; this is the case for essentially *all* modern solvers. One can always plug in the best current SAT solver and proof trimmer, without any modification, by simply indicating the path to their executables. WhyUnsat is fast and robust as it leverages the original solver's work through its unsatisfiability proof and it exploits, via MPI, the -progressively cheaper- shared-memory and distributed computing resources. It is also anytime and user-friendly; it quickly shows a human-readable presentation of (an over-approximation of) the explanation, which is then progressively reduced until minimality (unless interrupted by the user).

This paper is organized as follows. After some short preliminaries in Section 2, we describe the architecture of WhyUnsat in Section 3. Section 4 discusses its application to Lazy Clause Generation and Satisfiability Modulo Theories. After that, in Section 5 we report on related work and in Section 6 we present some experimental evaluation. Finally, we conclude in Section 7, where we also outline future research directions.

2 Preliminaries

We assume the reader to be familiar with standard SAT terminology. In this paper, we will sometimes consider CNFs of the form $C_0 \wedge C_1 \wedge \cdots \wedge C_{k-1}$, where each C_i is a set of clauses. Intuitively, we consider problems with k high-level constraints, and each C_i is the CNF encoding of one such constraint. Throughout the paper we will use (indexed) upper-case C to denote such high-level constraints, and (indexed) lower-case c to refer to clauses.

Given an unsatisfiable CNF F, a *Minimally Unsatisfiable Subformula of* F (MUS), also known as a *minimal unsatisfiable core*, is an unsatisfiable set of clauses $\mathcal{M} \subseteq F$ such that for every clause $c \in \mathcal{M}$, the formula $\mathcal{M} \setminus \{c\}$ is satisfiable.

Given an unsatisfiable CNF $F = C_0 \wedge C_1 \wedge \cdots \wedge C_{k-1}$, a *group Minimally Unsatisfiable Subformula of* F [15] (g-MUS), also called a *high-level minimal unsatisfiable core* [17], is a set $S \subseteq \{0, 1, \cdots, k-1\}$ (or, abusing the notation, the set $\wedge_{i \in S} C_i$), such that $\wedge_{i \in S} C_i$ is unsatisfiable and for each $j \in S$, $\wedge_{i \in S \setminus \{j\}} C_i$ is satisfiable.

▶ **Example 2.** Let us consider $F = C_0 \wedge C_1 \wedge C_2$, with $C_0 = \{x,\ \neg x \vee y,\ z \vee y\}$, $C_1 = \{\neg x,\ \neg x \vee z\}$ and $C_2 = \{\neg x \vee \neg z\}$. Both $\{x,\ \neg x\}$ and $\{x,\ \neg x \vee z,\ \neg x \vee \neg z\}$ are MUSes of F. However, the only g-MUS is $\{C_0,\ C_1\}$.

3 Architecture

WhyUnsat is a conceptually simple and pragmatic tool, with no deep algorithmic ideas. It requires no wrappers or modifications of the underlying SAT solver, which only has to be able to do proof logging; this is the case for essentially *all* modern solvers. Hence WhyUnsat is *timeless* in this sense: the best new SAT solver can always be plugged in by simply indicating the path to its executable. WhyUnsat performs the following three steps:

First step. This step of WhyUnsat processes the input file in the well-known `gcnf` format. It is a slight extension of the DIMACS CNF format. It starts with a header

```
p gcnf <numVars> <numClauses> <numConstraints>
```

and each subsequent line contains a clause expressed as

```
{<highLevelConstraintNum>} literal_1 ...  literal_M 0.
```

where `highLevelConstraintNum` is a number in $\{0, 1, \cdots, k-1\}$ expressing the high-level constraint associated with this clause.

This initial phase ignores the high-level constraints and only works at the clause level. The CNF formula is processed by a SAT solver (currently KISSAT), which is asked to log an unsatisfiability proof into a file. Note that proof logging incurs a negligible overhead on the solver (less than 2% on average). An off-the-shelf tool such as DPR-TRIM[2] [14] is then used to extract an unsatisfiable *core* of input clauses from the unsatisfiability proof produced by the SAT solver. Again, the best new trimmer tool can always be plugged in into WHYUNSAT by simply indicating the path to its executable. Since a core is just another CNF, it can hence be fed again into the SAT solver, generating another proof that usually uses fewer input clauses. This well-known process [22] can be iterated, and is controlled by WHYUNSAT under three possible strategies (also used in [7]):

a) until fixpoint (argument `-fp`)

b) repeat a given number $N \geq 0$ of iterations (`-times N`)

c) iterate until we get less than P% reduction (`-percent P` ; default, with 20%)

If the user proved the unsatisfiability of the original problem via an external proof-logging SAT solver, WHYUNSAT can leverage the proof generated in the original call to the SAT solver, which saves significant time and work. To do this, WHYUNSAT admits the optional argument `-proof <name>` , and hence it does not even have to re-inspect the full initial CNF or re-prove its unsatisfiability.

Usually, the core obtained from trimming this initial proof is much smaller –and hence faster to handle– than the initial CNF. Subsequent iterations are usually faster but give more modest reductions, although this is highly dependent on the industrial application domain. But note that trimming until fixpoint is usually too expensive for this kind of instances (and, of course, even reaching a fixpoint does not imply that a MUS has been obtained). Altogether, this is why it is useful for the user to be able to tune by means of the three aforementioned core reduction strategies.

Second step. This second step starts from the set of core clauses resulting from the first step. The aim here is to collect a small (hitting) set S of high-level constraints such that each core clause belongs to at least one high-level constraint in S. This handles the fact that the same clause (with its literals possibly ordered differently) can belong to multiple constraints.

To do this efficiently, it first uses a map, built from the input gcnf, indexed by clauses (seen as an *ordered* sequence of literals). The map is used to recover, for each core clause, the set of (one or more) constraints it belongs to. This results in a collection CS of high-level constraint sets, one set for each core clause. Then the following greedy algorithm is used to find a hitting set S: (1) for all singletons $\{C\}$ in CS, add C to S; (2) remove from CS the sets A with $A \cap S \neq \emptyset$; (3) while $CS \neq \emptyset$, add to S a maximally frequent high-level constraint C in CS, and remove from CS all sets containing C.

At this point, WHYUNSAT already displays S to the user as a first over-approximation of the g-MUS, i.e., of the explanation. When given an (optional) file (argument `-exp <explanationFile>`), where each line has the format `<constraintNum> "text describing this constraint"`, S is listed in this user-friendly way, as in Example 1. In our industrial experience, this initial explanation is frequently minimal or very close to minimal (see the experiments below), and many times it suffices for a knowledgeable user to understand the cause of infeasibility.

[2] `https://github.com/marijnheule/dpr-trim`

▶ **Example 3.** It is interesting to understand that, *even if* the clause-level core produced by step 1 were minimal, and *even if* the hitting set algorithm of step 2 applied to this core always returned a minimal hitting set S (both are not the case in general), still some constraint of S could be redundant. Consider the three constraints:

$C_1 = \{\ x \vee y,\ \ x \vee z\}, \quad C_2 = \{\ \neg x,\ \ \neg z\ \}, \quad C_3 = \{\ \neg y\ \}.$

Then a clausal core returned by step 1 can be $\{\ x \vee y,\ \ \neg x,\ \ \neg y\ \}$. It involves all three constraints, so the unique minimal hitting set is $\{C_1, C_2, C_3\}$. However, at the constraint level, once we pull all clauses of C_1 and C_2 back in, we see that constraint C_3 is not needed, since $\{\ x \vee z,\ \ \neg x,\ \ \neg z\ \}$ is already unsatisfiable.

Third step. In this final step, the aim is to reduce the unsatisfiable set of high-level constraints $S = \{C_1, \cdots, C_n\}$ by removing C_i's while preserving unsatisfiability (i.e., we use a simple deletion-based algorithm [11, 2, 16, 10]). To do this, WhyUnsat exploits modern shared-memory and distributed computing resources, via MPI. The master process first broadcasts to all workers the set $S = \{C_1, \cdots, C_n\}$ and, for each C_i, the set of its clauses. Then the master sends out to the workers (at most) n jobs, identified by job index i in $1 \ldots n$. A worker taking job i will call a SAT solver with the input CNF consisting of all clauses generated by constraints of $S \setminus (\{C_i\} \cup U)$, where U is the set of constraints already marked "unnecessary" (initially empty). If the worker for job i finishes and returns "satisfiable" to the master, then the master marks C_i as "necessary", and sends this worker a new job (if any). If the worker with job i returns "unsatisfiable" then the master marks C_i as "unnecessary", interrupts all workers, notifies them that C_i has to be added to U, and sends out new (yet unmarked) jobs. This continues until all constraints are marked. Then, the final result is displayed and also written to file if option `-output <outputFile>` is given.

▶ **Example 4.** Assume we start the third step with the initial set S with three constraints

$\{\ C_1 = \{\neg x \vee z,\ \neg y \vee z,\ \neg z\},\ C_2 = \{x\},\ C_3 = \{y\}\ \}$

that is broadcast to all workers. Then the master sends out jobs 1, 2 and 3 to three different workers. Assume the worker with job 1 finishes first, returning "satisfiable" to the master for $S \setminus \{C_1\}$, so the master marks C_1 as "necessary". Here $S \setminus \{C_2\}$ and $S \setminus \{C_3\}$ are both unsatisfiable, but of course, as the example illustrates, this does not mean that C_2 and C_3 can *both* be removed, as C_1 alone is satisfiable. Assume job 2 finishes before job 3, returning "unsatisfiable". Then the master marks C_2 as "unnecessary" and interrupts all workers indicating this fact, so they can add C_2 to U. Then some worker gets the new job 3 (the only job pending), which (since now $U = \{C_2\}$) is a SAT call with $\{C_1\}$, returning "satisfiable" to the master, which then concludes that the final minimal explanation is $\{C_1, C_3\}$.

4 Application to Lazy Clause Generation / SAT Modulo Theories

SAT solvers are also being used in the context of Constraint Programming by Lazy Clause Generation (LCG), as well as in SAT Modulo Theories (SMT), where each constraint T, or –in essence equivalently– each SMT *theory* T, has its own algorithm (a *T-solver*) for T-propagating literals given the current SAT trail, and for detecting T-inconsistencies. Such propagations or inconsistencies are then communicated to –and handled by– the SAT solver, and justified during the SAT solver's conflict analysis through *reason clauses* provided on demand to the SAT solver by the T-solver. For the purposes of this paper, it suffices to know that, in case of unsatisfiability, the SAT solver's proof will of course make use of these reason clauses, each one of them contributed by a given constraint/theory T. In fact, the latest versions of SMT solvers such as CVC5 [4] or LCG solvers [19] leverage the

(proof-logging) CaDiCaL SAT solver through its IPASIR-UP interface [13], designed for this purpose. Indeed, as it is the case for WhyUnsat, this allows these SMT/LCG solvers to always plug in the newest (in this case, IPASIR-UP-compatible) SAT solver. So it is clear how WhyUnsat is also directly applicable here, at no implementation cost: the LCG/SMT solver only has to write a line to the gcnf file each time it generates a reason clause.

5 Related Work

We already mentioned HaifaHLMUC and MUSer2 in the introduction. As mentioned, these tools excel on instances such as the ones of the 2011 SAT competition for group-oriented MUS, which were *all* from gate-level circuit analysis: thousands of constraints/gates with typically three clauses each, and g-MUSes of thousands of constraints as well. In contrast, WhyUnsat is intended for completely different applications: hard industrial planning/timetabling/scheduling, with typically many clauses per constraint, and finding explanations, i.e., small g-MUSes. It is not intended, and inadequate, for gate-level circuit analysis (for huge g-MUSes step 3 would require another massively parallel setup, see "Future Work").

HaifaHLMUC uses a number of sophisticated techniques, requiring a significant implementation effort to incorporate them into some existing SAT solver. It is therefore not very surprising that it has not been updated for about a decade by adapting the newest (and progressively more complex) SAT solvers. In contrast, through MUSer2's IPASIR [3] interface, one can easily make new wrappers for it to use other SAT solvers. For our experiments, we did this for the latest CaDiCaL SAT solver (Kissat does not support IPASIR). MUSer2 relies on the ability of SAT solvers to handle *assumptions* [12] and *indicator* variables (harming performance, see below). But the emergence of efficient proof trimmers such as DPR-trim, after MUSer2 and HaifaHLMUC were developed, has helped to establish proof tracing as the dominant method for core extraction (over in-memory or assumption-based schemes).

In contrast to MUSer2 and HaifaHLMUC, WhyUnsat does not exploit *model rotation*: if F is a set of clauses and a model is found for $F \setminus \{c_i\}$ (showing the clause c_i is necessary in the MUS), model rotation attempts to flip one literal of c_i, thus getting a new model where c_i is true; if then one single other c_j of F becomes false, then c_j is necessary too. In our applications, with many clauses per constraint, model rotation can hardly be exploited.

A parallel approach extending MUSer2 is [6], with involved techniques for information exchange between workers. Unfortunately, according to the tool documentation, "the multi-threaded code [...] is probably broken and inconsistent with the changes that have been made since then". WhyUnsat aims to avoid this type of situations, where sophisticated techniques are incorporated into a system that are later difficult to maintain.

Proofs are also exploited in [5], for MUS (not g-MUS) extraction, through (assumption-based) interleaving of trimming and MUS extraction (hence no timelessness).

6 Experimental Evaluation

Here we illustrate some of the situations encountered while solving combinatorial optimization problems in an industrial context, on three different problems, with 20 instances each: RCPSP, Timetabling and Event planning problems. Our system and the problems with a more precise description can be found at `https://github.com/enric-rodriguez/WhyUnsat`; cf. "cadical-aux" for reproducing the results of the first table (with cadical-3.0.0-86899b4). All experiments were done on a cluster of 3.3Ghz 16GB Intel Xeon E-2124 machines. Times are in seconds.

The first table below shows the average time (over the 20 instances) needed to prove the unsatisfiability of the initial problem. We compare: Kissat 4.0.4 and CaDiCaL 3.0.0 in an offline command-line run version; "CaDiCaL-IPASIR" runs a wrapper that uses IPASIR [3] to add the clauses and solve; the third version "assump" uses assumptions [12] in IPASIR by adding one indicator variable per high-level constraint. In the latter two, we exclude parsing time from the timecount. Results speak for themselves. In some instances the command-line versions perform much better (analyzing precisely why is beyond the scope of this paper). In any case, using the fastest available SAT technology has an important impact on the overall solving time:

Benchmark	Kissat	CaDiCaL	CaDiCaL-IPASIR	CaDiCaL-IPASIR-assump
RCPSP	9.6	35	84	172
Timetabling	1.6	8.4	43	225
Events	0.4	0.5	0.5	1.4

The next three tables are for RCPSP, Timetabling and Events, and give results by three tools: WhyUnsat on 16 cores using Kissat as an offline SAT solver, MUSer2 using the latest CaDiCaL via IPASIR, and HaifaHLMUC using Glucose [1]. For WhyUnsat we set `-percent 2` for its first step to keep reducing the core while reductions are of at least 2%. For MUSer2, we did this only when better than the default option without initial trimming. HaifaHLMUC was used with the default options.

For each instance we give **size** and **time** for the first over-**approx**imated g-MUS (the result of step 2) and for the **final** g-MUS, where "(k)" in **approx** indicates that the **final** size is smaller (by k). The best sizes are marked in **bold**. A dash **-** indicates that the tool could provide no information and **TO** that the final result could be not be proved minimal, both within a time limit of 10 minutes. HaifaHLMUC is not included for RCPSP as it could not provide any informative result.

	RCPSP Benchmarks							
	WhyUnsat				MUSer2			
	Size		Time		Size		Time	
#	Approx	Final	Approx	Final	Approx	Final	Approx	Final
1	**8**	**8**	6.2	8.5	–	–	–	–
2	**27**(3)	**24**	48.8	75.9	31(5)	26	25.4	39.5
3	**19**(2)	**17**	18.4	30.0	37(3)	34	49.8	101.2
4	**37**(1)	**36**	44.8	85.2	–	–	–	–
5	5	5	4.7	5.2	**4**	**4**	2.1	2.2
6	**32**(5)	**27**	24.4	68.9	39(3)	36	46.9	372.1
7	**23**(1)	**22**	12.7	21.0	45(19)	26	23.4	34.1
8	37(5)	32	20.5	64.7	**24**(1)	**23**	11.1	17.1
9	**4**	**4**	4.9	5.3	31(5)	26	13.3	30.9
10	32(4)	28	58.5	99.3	**26**(4)	**22**	86.0	387.3
11	**21**	**21**	7.9	14.8	39(8)	31	19.6	43.0
12	6	6	4.7	5.2	**5**	**5**	2.0	2.3
13	6	6	5.6	6.1	**5**	**5**	1.9	2.0
14	**26**	**26**	24.5	35.0	38	38	261.4	TO
15	**29**(1)	**28**	25.0	43.0	38(9)	29	15.5	76.3
16	**23**(2)	**21**	23.1	35.5	29(1)	28	47.3	51.5
17	**31**(7)	**24**	24.7	70.8	–	–	–	–
18	**33**(1)	**32**	57.7	83.0	37(2)	35	22.6	33.2
19	**24**(3)	**21**	30.1	46.0	30(3)	27	19.7	31.4
20	**4**	**4**	4.4	4.9	34(2)	32	61.5	122.8
		Average	22.6	40.4		Average	125.5	187.3

	Timetabling Benchmarks											
	WhyUnsat				MUSer2				HaifaHLMUC			
	Size		Time		Size		Time		Size		Time	
#	Approx	Final	Approx	Final	Approx	Final	Approx	Final	Approx	Final	Approx	Final
1	**61**(2)	**59**	49.0	52.4	**59**(1)	**58**	193.4	194.3	66	66	175.3	TO
2	**41**(2)	**39**	13.7	18.3	51(1)	50	183.6	183.9	50	50	39.3	47.9
3	**45**	**45**	2.2	2.7	57(1)	56	116.9	117.6	62	62	88.0	140.6
4	**37**(2)	**35**	1.4	2.3	61(1)	60	270.4	271.0	60	60	6.6	9.5
5	**27**	**27**	1.4	1.8	–	–	–	–	62	62	1.1	1.6
6	37	37	1.3	1.9	51(1)	50	49.6	49.8	**36**	**36**	0.1	0.1
7	**43**	**43**	6.4	7.0	51(1)	50	467.5	468.0	50	50	15.2	19.3
8	**27**	**27**	1.4	1.9	57(1)	56	188.7	189.0	44	44	0.2	0.3
9	15	15	0.8	1.0	57(1)	56	405.7	405.8	**14**	**14**	0.1	0.1
10	61(10)	**51**	4.8	9.2	–	–	–	–	**56**	56	0.8	1.2
11	**33**	**33**	2.3	2.9	57(1)	56	102.3	102.7	56	56	8.9	12.8
12	53	53	1.1	1.8	63(1)	62	340.0	340.3	**48**	**48**	0.1	0.1
13	**39**	**39**	1.2	1.9	55(1)	54	176.6	177.6	56	56	2.1	3.4
14	**47**	**47**	1.8	2.6	57(1)	56	335.7	336.3	50	50	0.5	0.9
15	**45**	**45**	18.1	18.8	55(1)	54	203.8	204.2	66(12)	54	193.9	500.4
16	49(4)	**45**	9.9	13.8	**47**(1)	46	17.4	17.5	56	56	85.1	131.2
17	**41**(2)	**39**	10.5	15.6	–	–	–	–	–	–	–	–
18	**51**	**51**	3.2	4.1	–	–	–	–	60	60	2.7	4.0
19	53(6)	**47**	9.1	13.0	51(1)	50	85.4	86.0	**50**	50	12.9	16.7
20	**37**	**37**	9.5	10.1	–	46	–	TO	–	–	–	–
		Average	7.5	9.2		Average	209.1	307		Average	91.6	134.5

Results on RCPSP and Timetabling show that WhyUnsat in general provides better quality first approximations and final g-MUSes and is able to do so more efficiently. Both MUSer2 and HaifaHLMUC sometimes do not provide any information at all.

	Event Benchmarks											
	WhyUnsat				MUSer2				HaifaHLMUC			
	Size		Time		Size		Time		Size		Time	
#	Approx	Final	Approx	Final	Approx	Final	Approx	Final	Approx	Final	Approx	Final
1	**20**(17)	**3**	2.0	44.6	26(9)	17	0.2	29.1	27(19)	8	0.3	TO
2	**19**(16)	**3**	3.1	28.8	27(11)	16	0.9	13.4	28(22)	6	1.7	TO
3	**17**(14)	**3**	1.9	13.4	23(7)	16	0.3	14.2	24(16)	8	0.3	TO
4	**20**(17)	**3**	2.7	48.4	24(5)	19	0.3	13.2	25(18)	7	0.4	TO
5	**16**(13)	**3**	4.2	109.5	27(13)	14	1.8	14.0	28(21)	7	5.1	TO
6	**21**(18)	**3**	1.0	55.0	25(11)	14	0.1	0.5	29(24)	5	0.1	TO
7	**26**(19)	**7**	1.8	13.2	**26**(10)	16	0.4	20.5	27(15)	12	0.2	TO
8	**19**(16)	**3**	1.7	20.6	29(9)	20	0.2	1.2	30(23)	7	0.2	TO
9	**18**(15)	**3**	2.4	52.6	26(4)	22	0.4	3.3	27(20)	7	0.6	TO
10	**17**(14)	**3**	2.9	19.9	23(4)	19	0.4	12.4	24(16)	8	0.4	TO
11	**16**(13)	**3**	2.5	53.3	20(6)	14	0.7	3.9	21(15)	6	0.7	TO
12	**20**(17)	**3**	3.6	66.7	26(5)	21	0.7	11.4	27(19)	8	0.9	TO
13	**16**(13)	**3**	4.3	58.0	28(12)	16	0.6	2.4	29(23)	6	0.4	TO
14	**19**(16)	**3**	10.0	33.3	21(4)	17	2.6	94.9	22(15)	7	3.6	TO
15	**21**(18)	**3**	2.0	58.0	27(7)	20	0.4	2.7	28(21)	7	0.3	TO
16	**18**(15)	**3**	12.9	80.9	19(8)	11	2.2	28.8	22(12)	10	4.5	TO
17	**17**(14)	**3**	2.8	61.0	**17**(6)	11	0.3	2.8	20(12)	8	1.1	TO
18	**15**(12)	**3**	3.2	48.0	21(10)	11	0.5	3.6	22(16)	6	0.5	TO
19	**22**(16)	**6**	3.4	19.2	28(4)	24	0.3	2.2	29(18)	11	0.3	TO
20	**18**(15)	**3**	1.7	49.8	30(7)	23	0.5	7.7	31(24)	7	0.5	TO
		Average	3.5	46.7		Average	0.7	14.1		Average	1.1	600

The Event results are surprising (motivating us to independently check all results in several ways), as there are at least two independent unsatisfiability reasons for them: a small one, offering the user a good and clear explanation, but also a large one. On all 20 instances, MUSer2 systematically finds a large one, whereas WhyUnsat and HaifaHLMUC appear to be directed towards a small one (although HaifaHLMUC ends up timing out). For the large ones, proving their unsatisfiability is easy (hence the shorter runtimes for MUSer2) while for the small ones, SAT calls get harder and harder as the set gets reduced. We conjecture that WhyUnsat gets directed to the better small g-MUSes thanks to its hitting set algorithm for handling clauses belonging to several constraints.

7 Conclusions and Future Work

We have shown that WhyUnsat is a useful tool for explaining unsatisfiabilities in a practical context, in terms of efficiency, robustness, usability and user-friendliness, and moreover it is timeless, as it costs no work to always plug in the best current SAT solver and proof trimmer. Regarding future work, as computation resources become cheaper, efficiency can be further improved by using different strategies for step 3; for instance, on machines (or clusters) with hundreds of nodes, it pays off to speculatively run certain calls, or not to interrupt all workers after the first "unsatisfiable" answer (as the answers of the ones returning "satisfiable" will still be valid).

References

1 Gilles Audemard and Laurent Simon. On the glucose SAT solver. *Int. J. Artif. Intell. Tools*, 27(1):1840001:1–1840001:25, 2018. doi:10.1142/S0218213018400018.

2 R. R. Bakker, F. Dikker, F. Tempelman, and P. M. Wognum. Diagnosing and solving over-determined constraint satisfaction problems. In Ruzena Bajcsy, editor, *Procs 13th IJCAI*, pages 276–281, 1993. URL: http://ijcai.org/Proceedings/93-1/Papers/039.pdf.

3 Tomás Balyo, Armin Biere, Ashlin Iser, and Carsten Sinz. SAT race 2015. *Artif. Intell.*, 241:45–65, 2016. doi:10.1016/J.ARTINT.2016.08.007.

4 Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems – 28th International Conference, TACAS 2022*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022. doi:10.1007/978-3-030-99524-9_24.

5 Anton Belov, Marijn JH Heule, and Joao Marques-Silva. Mus extraction using clausal proofs. In *Theory and Applications of Satisfiability Testing–SAT 2014: 17th International Conference, Vienna, Austria, July 14-17, 2014. Proceedings*, pages 48–57. Springer, 2014.

6 Anton Belov, Norbert Manthey, and Joao Marques-Silva. Parallel mus extraction. In Matti Järvisalo and Allen Van Gelder, editors, *Theory and Applications of Satisfiability Testing – SAT 2013*, pages 133–149, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

7 Anton Belov and João Marques-Silva. Muser2: An efficient MUS extractor. *J. Satisf. Boolean Model. Comput.*, 8(3/4):123–128, 2012. doi:10.3233/SAT190094.

8 Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL 2.0. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification – 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*, volume 14681 of *Lecture Notes in Computer Science*, pages 133–152. Springer, 2024. doi:10.1007/978-3-031-65627-9_7.

9 Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL, Gimsatul, IsaSAT and Kissat entering the SAT Competition 2024. In Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2024 – Solver, Benchmark and Proof Checker Descriptions*, volume B-2024-1 of *Department of Computer Science Report Series B*, pages 8–10. University of Helsinki, 2024.

10 John W. Chinneck and Erik W. Dravnieks. Locating minimal infeasible constraint sets in linear programs. *INFORMS J. Comput.*, 3(2):157–168, 1991. doi:10.1287/IJOC.3.2.157.

11 Nachum Dershowitz, Ziyad Hanna, and Alexander Nadel. A scalable algorithm for minimal unsatisfiable core extraction. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing – SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, volume 4121 of *Lecture Notes in Computer Science*, pages 36–41. Springer, 2006. doi:10.1007/11814948_5.

12 Niklas Eén and Niklas Sörensson. Temporal induction by incremental sat solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003. BMC'2003, First International Workshop on Bounded Model Checking. doi:10.1016/S1571-0661(05)82542-3.

13 Katalin Fazekas, Uwe Egly, Marijn JH Heule, Stefan Szeider, and Armin Biere. Satisfiability modulo user propagators. *Journal of Artificial Intelligence Research*, 81:989–1017, 2024. doi:10.1613/jair.1.16163.

14 Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Strong extension-free proof systems. *J. Autom. Reason.*, 64(3):533–554, 2020. doi:10.1007/S10817-019-09516-0.

15 Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reason.*, 40(1):1–33, 2008. doi:10.1007/S10817-007-9084-Z.

16 João Marques-Silva and Inês Lynce. On improving MUS extraction algorithms. In Karem A. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing – SAT 2011 – 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*, volume 6695 of *Lecture Notes in Computer Science*, pages 159–173. Springer, 2011. doi:10.1007/978-3-642-21581-0_14.

17 Alexander Nadel. Boosting minimal unsatisfiable core extraction. In Roderick Bloem and Natasha Sharygina, editors, *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*, pages 221–229. IEEE, 2010. URL: https://ieeexplore.ieee.org/document/5770953/.

18 Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. enric-rodriguez/WhyUnsat. Software, swhId: swh:1:dir:68a66c6e39049a7035e472378224d089a41143f4 (visited on 2026-05-26). URL: https://github.com/enric-rodriguez/WhyUnsat, doi:10.4230/artifacts.26035.

19 Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation = lazy clause generation. In Christian Bessiere, editor, *CP*, volume 4741 of *Lecture Notes in Computer Science*, pages 544–558. Springer, 2007. doi:10.1007/978-3-540-74970-7_39.

20 Vadim Ryvchin and Ofer Strichman. Faster extraction of high-level minimal unsatisfiable cores. In Karem A. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing – SAT 2011 – 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*, volume 6695 of *Lecture Notes in Computer Science*, pages 174–187. Springer, 2011. doi:10.1007/978-3-642-21581-0_15.

21 Peter J. Stuckey. There are no CNF problems. In Matti Järvisalo and Allen Van Gelder, editors, *Theory and Applications of Satisfiability Testing – SAT*, volume 7962 of *LNCS*, pages 19–21. Springer, 2013. doi:10.1007/978-3-642-39071-5_3.

22 L. Zhang and S. Malik. Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications. In *2003 Conference on Design, Automation and Test in Europe Conference, DATE '03*, pages 10880–10885. IEEE Computer Society, 2003. doi:10.1109/DATE.2003.10014.