Solving Constrained Horn Clauses as C Programs with CHC2C

Levente Bajczi and Vince Molnár

Department of Measurement and Information Systems,
Budapest University of Technology and Economics, Hungary
{bajczi,molnarv}@mit.bme.hu

Abstract. Solving Constrained Horn Clauses (CHC) is necessitated by numerous fields in formal methods, from verifying software and smart contracts to modeling systems, yet the competitive scene for academic tools remains fairly sparse, especially compared to more popular fields such as software verification. Comparative evaluation as a competition, such as SV-COMP or CHC-COMP, sparks a more cohesive community around fields in formal methods. Lately, a trend has been emerging with tools such as Btor2C that bridge multiple fields together, thus widening this cohesion. Following that example, we propose and perform an experiment, where we use CHC-to-C transformation to apply software verification tools to linear CHC problems. In the process, we help both fields by diversifying the scene of CHC solvers and providing new and valuable benchmarks to aid the development of software verification tools. Using these benchmarks, we uncovered a previously hidden bug in multiple verification tools that can lead to false positive results. By analysing the results of the experiment, we can confidently make a recommendation for developers of software verifiers to consider supporting CHCs via our pre-verification transformation.

Keywords: CHC · verification · formal methods · software verification

1 Introduction

Formal methods have been gaining significant traction in many new domains in recent years. Facilitating this acceleration of adoption are the many specialized comparative evaluation-based competitions among tools, such as `SV-COMP` for software verification [2], `HWMCC` for hardware model checking [6], `SMT-COMP` for solving queries in satisfiability modulo theories (SMT) [26], or `CHC-COMP` for solving constrained Horn clauses (`CHC`) [10]. These competitions boost both academic interest and visibility towards potential users of the competitors, as well as provide a more-or-less standardized benchmark suite for evaluating the tools. However, with certain tools having been developed for multiple decades to work (and compete) in one of these domains, the price of entry into one of the more established fields can be insurmountable to new tools. Thus, instead of developing a brand new tool with specialized algorithms for a new field of study, a more

established tool can often be used instead, extended with a pre-transformation layer for adapting to one of the tool's supported formats.

Lately, this trend of adapting the problem to suit the verification tool instead of doing it vice versa has not only been used for solving new problems in new domains but also to close the gap between the many existing domains where formal methods are already being used. A success story in this regard is about `Btor2C` [4], which brought hardware verification problems in the language of `Btor2` to software verification tools by adapting it to standard `C`. This means that the exact source of the problem can be opaque to the tools themselves, and just by supporting `C`, they also support `Btor2` by extension. Furthermore, the software verification community gained access to valuable new benchmarks in process, available via the `SV-COMP` benchmark suite[1].

In this paper, we aim to yet again close a gap that exists among use cases of formal methods. Constrained Horn Clauses (`CHC`s) have long been used as a means to verify software [15,23,19], and lately, conventional software verification tools have also been applied to the reverse problem of representing `CHC`s via adapting them to a control-flow based representation, akin to imperative software [16,11,25]. However, by adapting the *format* of (linear) `CHC` problems to a suitable format for software verification, such as plain `C`, any off-the-shelf `C` verification tool becomes capable of solving `CHC` problems. We show that such a transformation is not only possible but beneficial in terms of performance as well.

1.1 Our Motivations and Contributions

The main motivation of our work is the diversification of the field of `CHC` solving, with the secondary motivation being the addition of valuable benchmark tasks to the software verification community. The importance of the former task is demonstrated well by the multitude of types of problems that necessitate `CHC`s: various synthesis problems such as syntax- and semantics guided, or functional synthesis [17,20,12]; systems verification [9]; software verification of conventional [15,19], higher-order [23,21], and specialized (e.g., dataflow-description [7]) programs; program equivalence checking [14]; or smart contract verification [27].

While the problem's importance is well understood, the same diverse and competitive scene that characterizes other formal methods fields (e.g., software verification) is yet to form for `CHC`s. At `CHC-COMP`, the highest number of competitors in any of its tracks remains well under 10 [10], and this number is not changing by much year-to-year [24,13].

By enabling conventional software verification tools to also support `CHC` problems, both fields (`CHC` solving and software verification) will be furthered: `CHC` solvers will be more diverse, and software verification tools will receive valuable, real-life problems as benchmarks to further tweak and tune their algorithms.

In connection with our motivations outlined above, our contributions regarding this paper are the following:

[1] `https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/`

I. We experimentally show that using software verification tools to solve CHC problems can be advantageous
II. We contribute valuable new benchmarks to the software verification community

In furtherance of these contributions, we also *implemented* and *validated* a proof-of-concept transformation tool that takes CHC problems and generates a C-language, error-label reachability-based representation of the problem. Furthermore, we *benchmarked* all ranked participants in this year's SV-COMP'24 on all transformed CHC problems, and *analyzed* their performance.

Note that our contribution is not centered around a fully-fledged and optimized tool for the CHC to C conversion, but rather a demonstration that this theoretical step is *possible*, and even a research prototype is capable of bridging the two domains together in a useful way.

Furthermore, we found and identified some bugs in some of the most widely used software verification tools using these tests, further justifying the point on the value of these new problems.

Novelty Certain approaches that use a *CHC-to-program* pre-transformation step already exist in the lineup of tools for CHC-COMP [16,11,25]. Details on these approaches are available in Sect. 1.3. The novelty of the approach presented in this paper stems from the genericness of the transformation: by using plain C as the target format, we enable *all* software verification tools to participate in CHC solving, while previous attempts focused on transforming the input into a tool-specific internal formalism. This enables seamless support for CHC problems, even for tools where no development towards supporting CHCs could be justified.

Significance The significance of our contributions can be demonstrated by the following points:

1. As shown in Sect. 3, some software verification tools perform on par with dedicated CHC solvers on a portion of problems. There are certain tasks that only software verification tools solved, while dedicated CHC solvers all timed out or threw an exception.
2. We uncovered previously hidden faults in well-known software verification tools by testing them on the newly acquired benchmark set of CHC problems.

We believe that these results significantly further the field of formal methods.

1.2 Background and Example

As a slightly simplified definition, let us define a CHC problem in the following way:

– A CHC problem consists of several *deduction rules*, i.e., implications

- A deduction rule may have *uninterpreted functions* (i.e., relations) in both its premise and its consequence
  - The consequence of a deduction rule may have *exactly one* uninterpreted function (besides the queries, which have zero)
  - The premise of a deduction rule may have zero, one, or more uninterpreted functions
    - If zero uninterpreted functions exist in a rule's premise, we call that rule an *atom*
    - If more than one uninterpreted function exists in a rule's premise, the rule is considered to be *non-linear*
  - If any rule is *non-linear*, the `CHC` is *non-linear*. Otherwise, the `CHC` is *linear*.
- Any number of deduction rules may deduce the literal *false*, these rules are called *queries*

The domain of variables in a `CHC` problem is given as SMT theories. In this paper, we mostly concentrate on the *core* theory, with additional support only for *integer* arithmetic. Note that problems requiring support for more theories exist (such as those using *arrays* or *algebraic data types*), but we discount those in the context of this work.

The goal for any `CHC` problem is to prove whether *false* is deducible (making the system *unsatisfiable*), in which case the query's premise is *true*. In practice, `CHC` problems are often encoded in the format of SMT-LIBv2 [8], ensuring tools' interoperability. Tasks for `CHC-COMP` use a strict subset of SMT-LIBv2 [10].

As an example, we can have three simple deduction rules:

$$\begin{aligned} n = 0 &\implies A(n) \\ A(n-2) &\implies A(n) \\ A(6) &\implies \mathit{false} \end{aligned}$$

The first rule states that $A(0)$ is *true*, and (given $A(0)$) the second rule makes all positive even numbers n to also evaluate $A(n)$ to *true*. The third rule is a *query*, stating that if $A(6)$ is *true*, then *false* is deduced, and hence the system is unsolvable. In this case, this problem is trivially solved as 6 is an even number, and hence, $A(6)$ must be *true*; thus, the system is *unsatisfiable*.

In software, the same process is easy to implement. We can take the query and start from its premise. This approach, demonstrated in Listing 1.1, is called *top-down*, or *backward*; because it starts the program by evaluating the query. While due to the infinite recursion in line 3 this program may not terminate (depending on the parameter to A), tools that can reason about recursion may find that the exit condition $A(6)$ is reachable.

An alternative, without recursion, is to leverage the support for nondeterminism in software verification tools and rewrite the program as seen in Listing 1.2. In this case, the program constructs all facts by following the deduction rules from the atomic facts; thus, this is called *bottom-up* or *forward* transformation, which is only possible for *linear* `CHC`s [25]. In lines $2-3$ the starting $A(0)$ fact is

Listing 1.1. Backward

```
1   int A(int n) {
2       if(n==0) return 1;
3   ↻   else if(A(n-2)) return 1;
4       else return 0;
5   }
6
7   int main() {
8       if(A(6)) return -1;
9       else return 0;
10  }
```

Listing 1.2. Forward

```
1   int main() {
2       int A, n = 0;
3       A = n;
4
5       while(true) {
6           n = nondet();
7           if(A == 6) return -1;
8           else if(A == n - 2) A = n;
9       }
10  }
```

constructed, then lines $5-9$ construct all further facts by assigning a nondeterministic value to n, then testing whether the previous iteration already deduced $A(6)$ (in which case an error code is returned), followed by testing if the last deduced $A(n)$ value is equal to the current n minus two; in which case the value of A (being the last deduced $A(n)$) is updated. This loop is repeated infinitely, only exiting when $A(6)$ is deduced. Verification tools, however, need only discover that the program is *unsafe*, and the original `CHC` problem is *unsatisfiable*. Solvers from both `SV-COMP`'24 and `CHC-COMP`'23 [10] mostly successfully solved the problem; these results are shown in Table 1.

Table 1. ✓: successful solution; ?: no result; ×: wrong verdict

	Software Verification Tool																CHC Solver						
	2ls	bubaak	bubaak-split	cpachecker	cpv	emergentheta	esbmc-kind	goblint	infer	nopsa	symbiotic	theta	uautomizer	ukojak	utaipan	veriabs	veriabsl	Eldarica	Golem	LoAT	Theta	U.TreeAut.	U.Unihorn	spacer-22
Forward	✓	✓	✓	✓	?	?	✓	×	?	?	✓	✓	✓	✓	✓	✓	✓							
Backward	?	✓	✓	✓	?	?	✓	×	?	?	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

1.3 Related Work

The presented approach of transforming `CHC` problems into reachability-based software verification tasks can be found in multiple `CHC` solver tools: `Eldarica` [16], `Ultimate Unihorn` [11], and `Theta` [25]. While the former two are long-time participants in `CHC-COMP` [24,13,10], their exact pre-transformation steps are not well documented (as even mentioned in the `CHC-COMP` competition report [10]). The latter, `Theta`, published an in-depth paper of this process alongside their debut on `CHC-COMP` [25]. Furthermore, `Theta` is the only tool out of the three that supports *both* a backward and a forward transformation option, with the other two – to the best of this paper's authors' knowledge – only supporting backward transformation.

As to not re-invent the transformation process from the ground up, in this paper, we take the approach implemented in `Theta` as the baseline for our experimental evaluation. We build on the existing implementation to create the proof-of-concept tool `CHC2C` and use the method described in the tool paper of `Theta` to reason about the theoretical possibilities of the approach.

2 Experiment Proposal and Methodology

To support the claims and goals of this paper, we devise an experiment to answer the following questions.

RQ1 Is a `CHC`-to-`C` transformation *sound* and *complete* in terms of support for `CHC`s?

RQ2 Is a `CHC`-to-`C` transformation beneficial for verification tools in terms of *performance*?

2.1 Theoretical Analysis

In short, the answer to **RQ1** is *no*. There are several substantial differences between the semantics of `CHC`s and `C` programs that will make this transformation impossible, no matter the encoding.

One of the main differences is in handling data types such as integers. `C` programs are designed to eventually run on actual hardware, which will always have a finite number of bits to represent any value; therefore, the `C` standard defines different lower and upper bounds on the size of variable types. In contrast, `CHC`s use infinite integers, akin to conventional SMT solvers, due to their *logical* nature.

However, one familiar with the inner behavior of software model checkers may see a disconnect here: most model-checking software rely on using `SMT` solvers and will definitely need to pay additional attention to implementing fixed-sized integer or array support when targeting `C`. Therefore, it is easy to see that this limitation on the expressive power of `C` over `CHC` *inside a model checking tool* is entirely artificial and stems from the `C` standard itself. Therefore, we anticipate that tools supporting a `C`-syntax input, albeit with a loose definition of semantics (allowing for infinite integers in the model), will fully mirror the semantics of the original `CHC` problem, even when transformed to `C`.

As most tools will not support redefining the semantics of `C` in an easy way, we also devise a safeguard that limits the types of false verdicts to be *false negatives*, akin to a BMC-like behavior, where a bound exists up to which the problem is deemed safe. However, instead of a bound on the steps from an initial state, we use a metric on the *variables* itself. With such safeguarding, the two verdicts a tool may have are the following:

- *Unsafe* output: Unsafe verdict
- *Safe* output: Safe, given all variables are in bound $[-A; +B]$.

Therefore, such tools may iterate with growing bounds until they run out of supported domain size, or find a counterexample and can return *unsafe*.

The problem safeguarding prevents is reaching the bounds of a variable type defined by the standard. When reaching an unsigned integer's upper- or lower bound, the C standard defines the expected behavior of the value to *wrap around*, thus remaining inside the bounds of the type. Therefore, $15 + 2 == 1$ is satisfiable given a 4-bit unsigned integer (bounds $[0; 16)$), but unsatisfiable given mathematical, infinite integers. The same problem should not exist for *signed* integers, as the standard leaves that to be implementation-dependent, and therefore, model checking tools *should* try and be flexible in handling that case. In practice, however, most tools just assume the signed values also wrap around.

The safeguards introduced above make the bounds of the values sufficiently far from the bounds of the variable type so as not to cause the wraparound problem, which would introduce *false positive* results.

Taking the above two problems into account, let us amend **RQ1** in the following way:

RQ1a Does safeguarding prevent *false positive* results in verification tools utilizing a CHC-to-C transformation?

RQ1b How often do *false negative* results occur on a realistic problem set?

Note that a similar set of problems arise with the use of *arrays*, as C-like arrays will have some size, outside of which addressing the array leads to undefined behavior, while CHCs use an infinite, mathematical definition of arrays, where every value of the address type's domain must have an associated legal value. However, we discount CHC problems with arrays in the scope of this paper so as not to overcrowd the transformation process with mitigations and safeguards.

2.2 Practical Analysis

To answer **RQ1** and **RQ2**, we need to test the proposed approach on a benchmark suite with diverse and numerous problems (for **RQ1**) and representative and challenging problems (for **RQ2**). The GitHub organization associated with CHC-COMP[2] contains a high number of CHCs sourced from a vast selection of sources and each year, a rigorous selection process is applied to the entirety of this collection to select a subset for CHC-COMP [10]. Therefore, we shall use all available problems to answer **RQ1** (more specifically, **RQ1a** and **RQ1b**), and last year's selection for CHC-COMP'23 [10] to answer **RQ2** as to not skew the performance evaluation with the numerous simple tasks some of the sources provide[3].

As mentioned above, we are using the tool Theta to implement the CHC-to-C transformation, as it has a well-documented and versatile pre-transformation step from CHCs to its internal CFA-like representation [25]. We implemented a

[2] https://github.com/chc-comp/

[3] The selection is based on perceived verification difficulty, problem traits, etc.

serialization step, which outputs a C-language program in the format of SV-COMP tasks [2] for compatibility and uses *signed integers* in place of the SMT integers (Boolean values are handled via the _Bool type). We used Theta's default transformation setting, which performs *forward* transformation on linear CHCs. Due to concerns raised about the performance of the *backward* transformation, as well as not to dilute the homogeneous benchmark suite consisting only of linear CHCs with possibly different characteristics over non-linear problems. We also included a new command-line parameter that governs whether to use bounds safeguarding, which uses a limit of $[-1\,000\,000\,000; +1\,000\,000\,000]$ to limit the range of the values. This limit ensures that no single operation may take a value outside the domain's range, which is assumed to be at least 32 bits wide.

The plan for the experimental evaluation is the following:

Getting a Baseline. Having the two benchmark suites (all containing every benchmark, comp containing CHC-COMP'23 benchmarks), we run CHC-COMP'23 participants [10] (and Spacer [22]) on the problems to get the following:

1. a baseline on their performance (in the case of comp), and
2. an expected verdict (for both sets)

The latter is necessary because unlike other benchmark sets such as that of SV-COMP [2], the expected verdicts for CHC-COMP are rarely published together with the input problems. Because verdicts may differ, we take a majority vote among the participants to decide an expected verdict. Note that we use binary classification, and no counterexample or proof validation takes place in these tests.

Note that we ran all participants on the comp benchmark set, but only a subset of the participants on the all benchmark set: Eldarica, Golem, Spacer and Theta.

Generating the Software Benchmark Suits. Using Theta, we generate four benchmark suites: all-range, all-norange, comp-range, comp-norange. Default settings are used for parsing the CHC files. We use the tool indent[4] to pretty-print the results for manual readability, with options -nut -i4.

A step of the transformation process is to generate code for non-deterministic transitions in the control flow automaton (i.e., a location has more than one outgoing edge, and their guards overlap). As in most cases (for CHCs), this means a binary decision, we used the C type _Bool to create a non-deterministic value of either 0 or 1, then used a switch-case statement to choose a transition in the CFA. An example of such a construct can be seen in Figure 1. Here, reach_error should never be reached because all cases of the switch statement return from *main*[5]. Therefore, the program exits. However, some tools fail to interpret this

[4] https://www.gnu.org/software/indent/

[5] Note that the outcomes do not change when an explicit cast is placed inside the switch statement's head, for which the standard definitely states the value should either be 0 or 1 [18]

```
extern void reach_error();
extern _Bool __VERIFIER_nondet_Bool();
int main(){
  switch(__VERIFIER_nondet_Bool()) {
    case 0: return 0;
    case 1: return 0;
  }
  reach_error();
}
```

2ls	bubaak	bubaak-split	cpachecker	cpv	emergentheta	esbmc-kind	goblint	infer	mopsa	symbiotic	theta	uautomizer	ukojak	utaipan	veriabs	veriabsl
✓	✓	✓	×	?	✓	✓	✓	?	?	✓	✓	×	×	×	✓	✓

Fig. 1. Verification test case, and verifier outputs (✓: safe, ×: unsafe, ?: no result)

correctly and allow a non-existent *default* case to be executed, which may lead to an incorrect verdict. Out of the 17 tools used in the context of this paper, 10 solve the example task correctly (yielding a *safe* verdict), and 4 tools solve it incorrectly (yielding an *unsafe* verdict). Therefore, we included an additional *default* case calling `abort()`, so that no trace taking the default case may continue after the statement. This modification made all tools previously giving a wrong output to correct their verdict. Therefore, we used this modification throughout the transformation process to get usable results from these 4 tools. We plan to open issues in the offending tools' repositories, where possible, to help developers correct this behavior.

Executing the Software Benchmarks. We used all non-*hors concours* participants of the ReachSafety category in `SV-COMP`'24. The full list is shown in Table 1. We used the submitted tool archives archived on Zenodo[6] to run the experiments, with the `unreach-call` property as an input specification. We recorded all tools' verdicts and CPU time over all benchmarks.

For the *soundness* and *completeness* check (**RQ1**), we executed all `SV-COMP` tools [3] on `all-range` and `all-norange`. For the *performance* check (**RQ2**), we executed all `SV-COMP` tools on `comp-range` and `comp-norange`.

Expected Outcomes. We expect that – besides some tool-specific failures – all tools will be able to handle at least a subset of the benchmark tasks, given they only use commonly seen elements of the `C` language. We expect that tools will report false negative verdicts for tasks where a counterexample cannot be found within the range of the signed integer domain. We also expect that in the case of the `*-norange` sets, tools that handle signed integer overflow will report false positives where wraparound behavior leads to an infeasible (in terms of feasibility over logical integers) counterexample to be found.

We can discount false positive verdicts, as they can easily be eliminated with a simple feasibility check in an SMT solver after the verification run, which could mark the verdict as *unknown*. We omitted such a check to not group false verdicts together with actual *unknown* verdicts in the experiment's evaluation phase.

[6] https://zenodo.org/

For *false* results, we matched up different tools' outputs and classified the benchmark as *commonly wrong* if multiple tools gave a wrong result for the same task, or *tool-specific fault* if at least one other tool solved the task correctly.

3 Results

We ran the experiments as discussed in Sect. 2.2 [1]. We collected 23958 `CHC` problems from the `CHC-COMP` GitHub organization[7]. We removed 8644 tasks containing algebraic data types and a further 8892 tasks containing arrays. Out of the remaining 6422 tasks, `Theta` could parse (in 60 seconds) 3076 tasks, out of which 1914 tasks were *linear*. The selected linear `CHC` problems were then transformed to `C`, both *with* and *without* the safeguarding technique presented in Sect. 2.1. We also filtered and transformed the `CHC-COMP`'23 benchmark set[8], yielding us 405 succesfully transformed `CHC` tasks.

Using the top performing `CHC` tools from `CHC-COMP`, we attempted to solve the 1914 tasks and got 1207 *true* (i.e., *safe*); 475 *false* (i.e., *unsafe*); and 232 *unknown* results. For the 405 tasks in `CHC-COMP`'23, 269 *true*, 71 *false* and 65 *unknown* verdicts were given.

Results of the first experiment regarding **RQ1** can be seen in Table 2 and Table 3. Each row corresponds to an `SV-COMP` participant, and the columns represent the classification of their outputs. The first six number columns show the number of wrong verdicts (*false*) the tool produced. Within the *false* results, there are *common* wrong results (where every software verification tool that solved the task produced the wrong verdict), and *tool-specific* wrong results (where at least one other tool succeeded with the verification). *True* results are correctly classified tasks. In all three groups of columns, *All* denotes the total number of results (*common false*, *tool-specific false*, *true*); and+ or - denote the output of the tool (i.e., *False*/*Tool*/+ means *tool-specific false positive* results, which are said to be *safe* by the tool yet the expected verdict is *unsafe*, and there is at least one other tool that returned *unsafe* correctly). The last but one column shows the *unconfirmed* verdicts, which are tasks the `CHC` solvers could not solve. The last column shows the *points* a tool would receive should the scoring system of `SV-COMP` be applied (1 point per good verdict, -32 points per false negative, -16 points per false positive). Note that false positives are discarded from point calculation due to their inherent easyness in checking: as opposed to software verification, a counterexample to a `CHC` problem is trivially checked by substituting the values in the SMT formula, and having it solved with a dedicated SMT solver. The table is sorted in descending order by the *points* column.

The number of tasks where a common fault caused all tools to report a wrong verdict upon success is not directly readable from the table. For the safeguarded transformation, 84 tasks were found to cause common wrong verdicts, while for the non-safeguarded transformation, 68 tasks. 16 tasks were found to cause

[7] `https://github.com/chc-comp/`
[8] `https://github.com/chc-comp/chc-comp23-benchmarks`

Table 2. Safeguarded transformation (all tasks)

	False						True			Unconf.	Points
	Common			Tool							
	All	+	-	All	+	-	All	+	-		
uautomizer	25	0	25	38	38	0	1054	686	368	4	1054
ukojak	24	0	24	38	38	0	1036	680	356	3	1036
utaipan	25	0	25	35	35	0	952	621	331	4	952
cpachecker	23	0	23	39	39	0	765	444	321	6	765
esbmc-kind	23	0	23	4	1	3	805	427	378	8	709
mopsa	20	0	20	1	0	1	210	210	0	0	178
2ls	0	0	0	0	0	0	101	84	17	0	101
infer	0	0	0	0	0	0	0	0	0	0	0
cpv	0	0	0	0	0	0	0	0	0	0	0
emergentheta	82	0	82	98	0	98	911	832	79	59	-2225
theta	81	0	81	191	0	191	1354	1157	197	55	-4758
bubaak	84	0	84	379	0	379	1183	1171	12	59	-10945
symbiotic	84	0	84	379	0	379	1183	1171	12	59	-10945
bubaak-split	84	0	84	379	0	379	1182	1170	12	59	-10946
veriabs	70	0	70	391	0	391	1117	1117	0	59	-11395
veriabsl	70	0	70	391	0	391	1113	1113	0	59	-11399
goblint	71	0	71	391	0	391	1101	1101	0	59	-11411

Table 3. Non-safeguarded transformation (all tasks)

	False						True			Unconf.	Points
	Common			Tool							
	All	+	-	All	+	-	All	+	-		
uautomizer	24	0	24	39	38	1	1142	769	373	5	1110
ukojak	23	0	23	39	38	1	1097	739	358	3	1065
utaipan	24	0	24	36	36	0	1028	690	338	5	1028
cpachecker	22	0	22	78	78	0	792	451	341	5	792
esbmc-kind	22	0	22	60	57	3	827	444	383	8	731
mopsa	22	0	22	1	0	1	212	212	0	0	180
2ls	0	0	0	3	3	0	98	81	17	0	98
infer	0	0	0	0	0	0	0	0	0	0	0
cpv	0	0	0	0	0	0	0	0	0	0	0
theta	67	0	67	208	0	208	1355	1158	197	66	-5301
emergentheta	68	0	68	210	0	210	1344	1147	197	59	-5376
symbiotic	65	0	65	352	1	351	1190	1139	51	173	-10042
bubaak	67	0	67	355	1	354	1229	1176	53	236	-10099
bubaak-split	68	0	68	375	0	375	1201	1170	31	59	-10799
goblint	66	0	66	396	0	396	1120	1120	0	59	-11552
veriabs	65	0	65	407	0	407	1169	1169	0	59	-11855
veriabsl	65	0	65	407	0	407	1169	1169	0	59	-11855

Table 4. Safeguarded transformation (CHC-COMP'23)

	False						True			Unconf.	Points
	Common			Tool							
	All	+	-	All	+	-	All	+	-		
utaipan	8	0	8	4	4	0	103	82	21	9	103
uautomizer	8	0	8	6	6	0	94	73	21	7	94
cpachecker	8	0	8	6	6	0	91	71	20	11	91
ukojak	7	0	7	6	6	0	87	68	19	9	87
mopsa	10	0	10	0	0	0	56	56	0	9	56
esbmc-kind	9	0	9	1	0	1	86	67	19	14	54
2ls	0	0	0	0	0	0	17	15	2	0	17
infer	0	0	0	0	0	0	0	0	0	0	0
cpv	0	0	0	0	0	0	0	0	0	0	0
emergentheta	32	0	32	13	0	13	168	163	5	48	-248
theta	37	0	37	14	0	14	198	190	8	48	-250
bubaak	39	0	39	23	0	23	234	234	0	52	-502
symbiotic	37	0	37	23	0	23	208	208	0	51	-528
bubaak-split	34	0	34	23	0	23	198	198	0	50	-538
veriabsl	30	0	30	23	0	23	190	190	0	50	-546
veriabs	31	0	31	23	0	23	189	189	0	50	-547
goblint	28	0	28	23	0	23	182	182	0	50	-554

Table 5. Non-safeguarded transformation (CHC-COMP'23)

	False						True			Unconf.	Points
	Common			Tool							
	All	+	-	All	+	-	All	+	-		
utaipan	8	0	8	4	4	0	133	108	25	10	133
cpachecker	12	0	12	15	15	0	117	90	27	10	117
uautomizer	8	0	8	7	6	1	116	93	23	10	84
ukojak	7	0	7	7	6	1	114	91	23	9	82
mopsa	12	0	12	0	0	0	76	76	0	9	76
esbmc-kind	9	0	9	15	14	1	93	72	21	14	61
2ls	0	0	0	0	0	0	17	15	2	0	17
infer	0	0	0	0	0	0	0	0	0	0	0
cpv	0	0	0	0	0	0	0	0	0	0	0
bubaak	34	0	34	20	0	20	258	247	11	70	-382
symbiotic	29	0	29	20	0	20	219	209	10	63	-421
emergentheta	31	0	31	23	0	23	226	218	8	51	-510
theta	29	0	29	23	0	23	212	204	8	50	-524
goblint	28	0	28	25	0	25	190	190	0	50	-610
bubaak-split	28	0	28	28	0	28	207	204	3	50	-689
veriabs	29	0	29	31	0	31	203	203	0	51	-789
veriabsl	29	0	29	31	0	31	202	202	0	51	-790

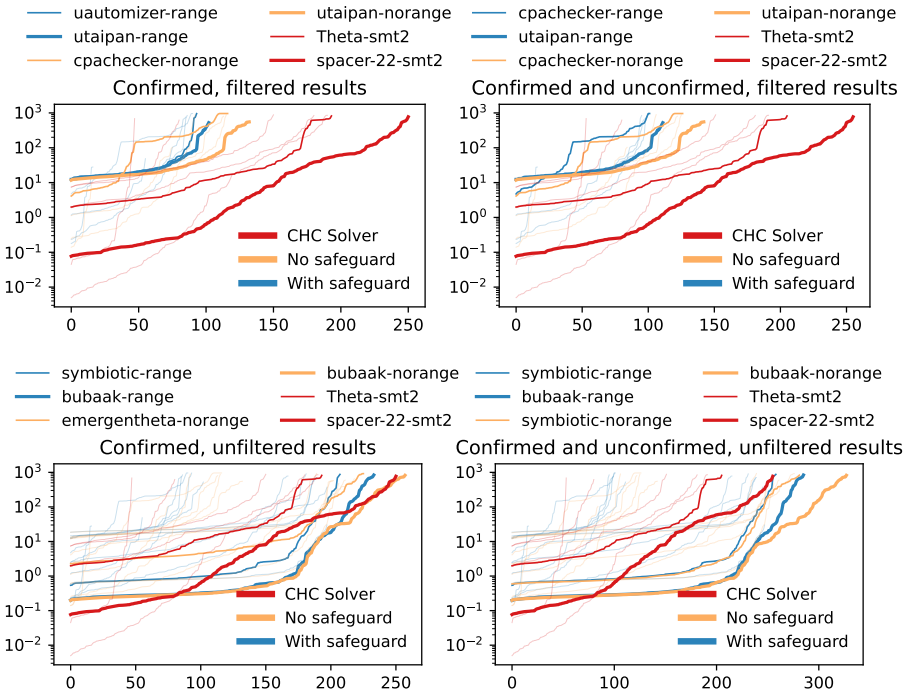

Fig. 2. Software verifiers vs. `CHC` solvers. Best two tools per category highlighted.

common wrong verdicts only in the safeguarded transformation but not in the non-safeguarded transformation, meaning all 68 tasks from the latter remain commonly wrong for the safeguarded version as well.

Table 4 and Table 5 show the results for the tasks in `CHC-COMP`'23 in the same format as described above. Here, 39 and 34 tasks were commonly wrong among the tools for the safeguarded and non-safeguarded transformations, respectively.

The performance of the tools is shown on the quantile plots in Figure 2. Horizontal axes show the number of solved tasks (out of 405 possible), and vertical axes show verification time, with data points in the series sorted by ascending order. Given any line $y(x) = N$, we can easily determine the tool solving the most tasks under N timelimit by taking the series containing the rightmost point under the said line.

Each quantile plot shows the performance of the native `CHC` solvers, the software verification tools using safeguarded transformation, and software verification tools with non-safeguarded transformation. The top two participants are highlighted and shown on the legend above each plot in each of the three groups. The top participant is shown in the thickest line.

Tools with negative points in Table 4 and Table 5 are not shown on the two plots in the first row.

Verdicts with unconfirmed results (i.e., tasks solved only by software verification tools) are not shown on the two plots in the first column. Wrong verdicts are not shown in either column.

4 Discussion

In this paper, we claim to have made two contributions: we have experimentally shown that solving `CHC` problems is possible – and sometimes even advantageous – using software verification tools, and we contributed a new set of valuable benchmarks to the software verification community.

The value of these benchmarks lies in the diversity and complexity of the benchmarks. As shown later in the discussion of the results, these tasks differentiated the verification tools, meaning they provide a good balance of difficulty, diversity, and complexity: if all tools were able to solve the tasks easily or no tool could solve any at all, this value would significantly decrease. Furthermore, by uncovering a latent fault in some of the tools (seen on the problem in Figure 1), we hope to have helped their development towards a more sound verification workflow. With these new benchmarks, we opened a pull request in the SV-Benchmarks repository[9].

As for the main contribution (i.e., analysis of the `CHC`-to-`C` transformation in verification), we show the following using the results in Sect. 3:

Ans1a The transformation can be free from introducing *false positive* results (answering **RQ1a**)

Ans1b The *false negative* results are significantly less frequent than correct results (answering **RQ1b**)

Ans2 The performance characteristics of software verification tools on `CHC` problems make them *competitive* (answering **RQ2**)

4.1 Analysis of the Results

For **Ans1a** and **Ans1b**, see Table 2 and Table 3: no common *false positive* results exist, meaning no *safe* task was deemed *unsafe* by all tools; but 84 and 68 common *false negative* results exist for the safeguarded and non-safeguarded transformations, respectively. 84 *false negative* results make up around 4% of all tasks (as to answer **Ans1b**). Tool-specific false positive results exist; for most tools, their number is significantly lower in the case of the safeguarded transformation.

Almost half of the tools received negative points (8), and 2 tools produced no results, leaving 7 tools to receive positive points. Out of these 7, the tool `2ls` is noteworthy: in the case of the safeguarded transformation, it produced *only correct results*, while in the non-safeguarded transformation, it only produced

[9] `gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/merge_requests/1467`.

3 *false positive* results. These are likely cases where `2ls` handled signed integer overflow using wraparound. This fairly clean result also provides further evidence for **Ans1a** and **RQ1a**, showing that safeguarding can prevent false positive results.

The top scoring tool in both cases, `Ultimate Automizer`, experienced 25 and 24 common faults out of the 84 and 68 commonly wrong tasks for the safeguarded and non-safeguarded transformation, respectively. A further 38 and 39 tasks were given a wrong verdict by the tool, where at least one further tool was successful, therefore, we classify these as tool-specific faults. Almost all of these were false *positive* results.

Note that these wrong verdicts might not mean actual faults in the former set of tools, as the handling of signed integer overflow is undefined, and if the 3 tools having more false positive results handle overflow as wraparound, while some other tools just allow values outside of the domain to persist; then tools using wraparound will have false positive results while the others will solve the tasks correctly. This explanation is more likely for such a small number of false positive results. However, for tools on the other end of the scores, having almost a quarter of the results be wrong is more likely to hide a fault in the tool rather than some quirk of such edge cases.

As for the performance of the software verification tools concerning **Ans2**, results are shown in Figure 2. The best performing native `CHC` solver throughout the experiments was `Spacer`, with second best being either `Golem` or `Theta` depending on the addition of unconfirmed results[10].

The overwhelming amount of false results in the case of some of the tools makes reasoning about performance difficult. We cannot distinguish between a faulty tool coupled with a lucky guess and a correctly reasoning tool. Therefore, we present multiple views about the participants' performance and the unique flaws these representations may carry.

Filtering. If we exclude all negative scoring tools (as per Table 4 and Table 5) from appearing on the plots, the native `CHC` solvers cannot be outperformed by software verification tools in terms of number of solved tasks. However, including these tools makes software verification seem on-par (in the case of confirmed results only) or even better (in the case of confirmed and unconfirmed results) than native `CHC` solvers. Their exclusion favors `CHC` solvers, while their inclusion favors software verification tools – these tools *did* provide a good verdict for a number of the tasks, but overwhelming false verdicts allow for speculation on hidden faults. A faulty tool may still produce a good result, but it may or may not be correct in its reasoning.

Including unconfirmed results. Excluding unconfirmed verdicts skews the results towards `CHC` solvers, because their majority vote *is* the confirmation –

[10] Note that these results show performance on only a subset of the real problems in `CHC-COMP`'23, which `Theta` could parse.

meaning if only a single `CHC` solver solves a task (e.g., `Spacer`), it is automatically confirmed. On the other hand, including unconfirmed verdicts may skew the results towards software verification tools because their verdict is accepted without validation from a dedicated (and thoroughly tested) `CHC` solver.

To summarize, the following problems exist with the four quantile plots representing performance characteristics in Figure 2:

Confirmed, filtered results Favors `CHC` solvers because only confirmed results are shown. Favors `CHC` solvers because the results are filtered only to show positive scoring tools.

Confirmed and unconfirmed, filtered results Favors software verification tools because unconfirmed results are treated as good verdicts. Favors `CHC` solvers because the results are filtered only to show positive scoring tools.

Confirmed, unfiltered results Favors `CHC` solvers because only confirmed results are shown. Favors software verification tools because the results are not filtered only to show positive scoring tools, potentially including tools incorrectly deducing the expected output

Confirmed and unconfirmed, unfiltered results Favors software verification tools because unconfirmed results are treated as good verdicts. Favors software verification tools because the results are not filtered only to show positive scoring tools, potentially including tools incorrectly deducing the expected output

Even if we ignore the two extremes (*confirmed, filtered*; and *confirmed and unconfirmed, unfiltered*) due to them skewing to one side only, the remaining two plots still do not show a congruent picture. What conclusion we *can* draw is that using software verification tools with the presented approach may be better than *some* native `CHC` solver tools and worse than or on-par with the leading `CHC` solvers. Note that these results are produced by a *research prototype* as the transformation step, meaning a theoretical, optimal transformation step may outperform these results. Our contribution concludes by showing that using this transformation, on-par performance is *possible* to native `CHC` solvers; and should be considered a viable solution in the future.

4.2 Threats to Validity

As the main contribution of this paper is the experiment design and its analysis, the factors that threaten the validity of this experiment are presented in this section.

Internal Validity. Consistency and accuracy of the experiments were ensured by using the BenchExec framework [5]. Memory consumption statistics may deviate between executions due to the managed nature of some languages used in developing the tested tools, therefore, such metrics are not used. CPU time and, therefore, solved tasks may be influenced by external factors such as other processes or environmental temperature fluctuations, therefore, minute differences are disregarded.

External Validity. The results of the experiments are at risk of not being generalizable due to the relatively low number of benchmarks used throughout the experiments. The experiment in this paper is designed to show the characteristics of *one possible* CHC-to-C transformation, and therefore, we can only state observations about the feasibility of the concept, not about the actual performance impact of an optimal transformation. This may skew the results to the detriment of software verification tools because there may be a better, optimized version of the CHC2C tool, which would make them outperform their current behavior.

Construct Validity. To justify the type of metrics used in the evaluation of the experiments, we considered the main use cases these tools would face should they be used in the approach described in this paper. Academic competitions such as CHC-COMP [10] or SV-COMP [2] reflect the performance of tools after careful tuning of them while constantly re-testing on the same benchmark set. Therefore, tools that may produce wrong results when applied first to a problem, are generally fixed before the actual competition. This means that directly comparing the number of successfully solved tasks of dedicated CHC solvers, which have been developed relying on the benchmark set we tested on, to software verification tools that see these problems the first time may skew the results to the benefit of CHC solvers. Small problems and imperfections (such as the one seen in Figure 1) may introduce false results.

Notice that we have designed our experiment and its analysis to consistently skew *against* our main hypothesis – which is that *software verification tools may be beneficial in solving CHCs* – rather than being *in favor of* it. Therefore, the presented results show a pessimistic view about the value of our contributions.

4.3 Conclusion

We have shown that software verification tools can be useful tools for CHC solving using a CHC2C pre-transformation step before verification. While dedicated CHC solvers produced a better ratio of good results to bad results, just the number of new tools that became capable of solving CHC problems results in a useful diversification of the state-of-the-art.

We plan to use a version of our presented approach in the next CHC-COMP to create a participant that uses a portfolio-based approach and tries to run the optimal software verification tool on any given CHC problem. We hope to show the developers and researchers in both domains, CHC solving and software verification, that it is worth collaborating on tool development and that the exact domain might only matter regarding a pre-verification transformation step, not at the algorithmic level. Thus, we hope to help integrate the knowledge of both fields, resulting in advantages for all parties involved.

References

1. Bajczi, L., Molnár, V.: Solving Constrained Horn Clauses as C Programs with CHC2C (Jan 2024). https://doi.org/10.5281/zenodo.10529452

2. Beyer, D.: Competition on Software Verification and Witness Validation: SV-COMP 2023. In: Sankaranarayanan, S., Sharygina, N. (eds.) 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 2022, Paris, France, April 22-27, 2023, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13994, pp. 495–522. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_{2}{9}
3. Beyer, D.: Verifiers and Validators of the 12th Intl. Competition on Software Verification (SV-COMP 2023) (2023). https://doi.org/10.5281/ZENODO.7627829
4. Beyer, D., Chien, P., Lee, N.: Bridging Hardware and Software Analysis with Btor2C: A Word-Level-Circuit-to-C Translator. In: Sankaranarayanan, S., Sharygina, N. (eds.) 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 2022, Paris, France, April 22-27, 2023, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13994, pp. 152–172. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_{1}{2}
5. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. Int. J. Softw. Tools Technol. Transf. **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
6. Biere, A., van Dijk, T., Heljanko, K.: Hardware model checking competition 2017. In: Stewart, D., Weissenbacher, G. (eds.) 2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017. p. 9. IEEE (2017). https://doi.org/10.23919/FMCAD.2017.8102233
7. Champion, A., Mebsout, A., Sticksel, C., Tinelli, C.: The Kind 2 Model Checker. Lecture Notes in Computer Science, vol. 9780, pp. 510–517. Springer (2016). https://doi.org/10.1007/978-3-319-41540-6_{2}{9}
8. Cok, D.R.: The SMT-LIBv2 Language and Tools: A Tutorial (2012), https://api.semanticscholar.org/CorpusID:63272811
9. Daniel, J., Cimatti, A., Griggio, A., Tonetta, S., Mover, S.: Infinite-State Liveness-to-Safety via Implicit Abstraction and Well-Founded Relations. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9779, pp. 271–291. Springer (2016). https://doi.org/10.1007/978-3-319-41528-4_{1}{5}
10. De Angelis, E., K., H.G.V.: CHC-COMP 2022: Competition Report. In: Hamilton, G.W., Kahsai, T., Proietti, M. (eds.) Proceedings 9th Workshop on Horn Clauses for Verification and Synthesis and 10th International Workshop on Verification and Program Transformation, HCVS/VPT@ETAPS 2022, and 10th International Workshop on Verification and Program TransformationMunich, Germany, 3rd April 2022. EPTCS, vol. 373, pp. 44–62 (2022). https://doi.org/10.4204/EPTCS.373.5
11. Dietsch, D., Heizmann, M., Hoenicke, J., Nutz, A., Podelski, A.: Ultimate TreeAutomizer (CHC-COMP Tool Description). In: Angelis, E.D., Fedyukovich, G., Tzevelekos, N., Ulbrich, M. (eds.) Proceedings of the Sixth Workshop on Horn Clauses for Verification and Synthesis and Third Workshop on Program Equivalence and Relational Reasoning, HCVS/PERR@ETAPS 2019, Prague, Czech Republic, 6-7th April 2019. EPTCS, vol. 296, pp. 42–47 (2019). https://doi.org/10.4204/EPTCS.296.7

12. Fedyukovich, G., Gurfinkel, A., Gupta, A.: Lazy but Effective Functional Synthesis. In: Enea, C., Piskac, R. (eds.) Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11388, pp. 92–113. Springer (2019). https://doi.org/10.1007/978-3-030-11245-5_5
13. Fedyukovich, G., Rümmer, P.: Competition Report: CHC-COMP-21. In: Hojjat, H., Kafle, B. (eds.) Proceedings 8th Workshop on Horn Clauses for Verification and Synthesis, HCVS@ETAPS 2021, Virtual, 28th March 2021. EPTCS, vol. 344, pp. 91–108 (2021). https://doi.org/10.4204/EPTCS.344.7
14. Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification. In: Crnkovic, I., Chechik, M., Grünbacher, P. (eds.) ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014. pp. 349–360. ACM (2014). https://doi.org/10.1145/2642937.2642987
15. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn Verification Framework. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9206, pp. 343–361. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_{2}{0}
16. Hojjat, H., Rümmer, P.: The ELDARICA Horn Solver. In: Bjørner, N.S., Gurfinkel, A. (eds.) 2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018. pp. 1–7. IEEE (2018). https://doi.org/10.23919/FMCAD.2018.8603013
17. Hu, Q., Cyphert, J., D'Antoni, L., Reps, T.W.: Exact and approximate methods for proving unrealizability of syntax-guided synthesis problems. pp. 1128–1142. ACM (2020). https://doi.org/10.1145/3385412.3385979
18. Information technology — Programming languages — C. Standard, International Organization for Standardization (Apr 2011)
19. Kahsai, T., Rümmer, P., Sanchez, H., Schäf, M.: JayHorn: A Framework for Verifying Java programs. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9779, pp. 352–358. Springer (2016). https://doi.org/10.1007/978-3-319-41528-4_{1}{9}
20. Kim, J., Hu, Q., D'Antoni, L., Reps, T.W.: Semantics-guided synthesis. Proc. ACM Program. Lang. **5**(POPL), 1–32 (2021). https://doi.org/10.1145/3434311
21. Kobayashi, N., Sato, R., Unno, H.: Predicate abstraction and CEGAR for higher-order model checking. In: Hall, M.W., Padua, D.A. (eds.) Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011. pp. 222–233. ACM (2011). https://doi.org/10.1145/1993498.1993525
22. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. Formal Methods in System Design **48**(3), 175–205 (Jun 2016). https://doi.org/10.1007/s10703-016-0249-4
23. Matsushita, Y., Tsukada, T., Kobayashi, N.: RustHorn: CHC-based Verification for Rust Programs. ACM Trans. Program. Lang. Syst. **43**(4), 15:1–15:54 (2021). https://doi.org/10.1145/3462205
24. Rümmer, P.: Competition Report: CHC-COMP-20. In: Fribourg, L., Heizmann, M. (eds.) Proceedings 8th International Workshop on Verification and Program Transformation and 7th Workshop on Horn Clauses for Verification and Synthesis, VPT/HCVS@ETAPS 2020, Dublin, Ireland, 25-26th April 2020. EPTCS, vol. 320, pp. 197–219 (2020). https://doi.org/10.4204/EPTCS.320.15

25. Somorjai, M., Dobos-Kovács, M., Ádám, Z., Bajczi, L., Vörös, A.: Bottoms Up for CHCs: Novel Transformation of Linear Constrained Horn Clauses to Software Verification. In: 10th Workshop on Horn Clauses for Verification and Synthesis (2023), https://ftsrg.mit.bme.hu/paper-hcvs23-chc/paper.pdf
26. Weber, T., Conchon, S., Déharbe, D., Heizmann, M., Niemetz, A., Reger, G.: The SMT Competition 2015-2018. J. Satisf. Boolean Model. Comput. **11**(1), 221–259 (2019). https://doi.org/10.3233/SAT190123
27. Wesley, S., Christakis, M., Navas, J.A., Trefler, R.J., Wüstholz, V., Gurfinkel, A.: Verifying Solidity Smart Contracts via Communication Abstraction in SmartACE. In: Finkbeiner, B., Wies, T. (eds.) Verification, Model Checking, and Abstract Interpretation - 23rd International Conference, VMCAI 2022, Philadelphia, PA, USA, January 16-18, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13182, pp. 425–449. Springer (2022). https://doi.org/10.1007/978-3-030-94583-1_{2}{1}