CHCVERIF: A Portfolio-Based Solver for Constrained Horn Clauses

Mihály Dobos-Kovács Levente Bajczi András Vörös
Department of Artificial Intelligence and Systems Engineering
Budapest University of Technology and Economics, Hungary
{mdobosko,bajczi,vori}@mit.bme.hu

Constrained Horn Clauses (CHCs) are widely adopted as intermediate representations for a variety of verification tasks, including safety checking, invariant synthesis, and interprocedural analysis. This paper introduces CHCVERIF, a portfolio-based CHC solver that adopts a software verification approach for solving CHCs. This approach enables us to reuse mature software verification tools to tackle CHC benchmarks, particularly those involving bitvectors and low-level semantics. Our evaluation shows that while the method enjoys only moderate success with linear integer arithmetic, it achieves modest success on bitvector benchmarks. Moreover, our results demonstrate the viability and potential of using software verification tools as backends for CHC solving, particularly when supported by a carefully constructed portfolio.

Funding. This research was partially funded by the 2024-2.1.1-EKOP-2024-00003 University Research Scholarship Programme under project numbers EKOP-24-3-BME-213, and the Doctoral Excellence Fellowship Programme under project numbers DKÖP-23-1-BME-5 and DKÖP-23-1-BME-15; with the support provided by the Ministry of Culture and Innovation of Hungary from the NRDI Fund.

1 Introduction

Constrained Horn Clauses (CHCs) form a widely adopted intermediate representation for a variety of verification tasks, including safety checking [29, 16, 18], invariant synthesis [21, 23, 17], and systems verification [12]. Consequently, CHC solving has become a fundamental component in many formal verification pipelines. Despite the maturity of existing solvers such as Eldarica [20] and Spacer [24], there are still unsolved challenges, such as supporting CHCs with bitvector theories.

We present CHCVERIF, a portfolio-based CHC solver that adopts a software verification perspective. Rather than solving CHCs directly at the logical level, CHCVERIF translates CHC problems into semantically equivalent C programs and applies a portfolio of C verifiers to determine their correctness. The satisfiability of the original CHC system corresponds to the correctness of the generated C program. This approach enables CHCVERIF to take advantage of mature software verification backends while diversifying its solving strategy through portfolio composition.

CHCVERIF builds on COVERITEAM [4], a recently proposed framework for constructing compositional verification portfolios. COVERITEAM allows CHCVERIF to orchestrate multiple C verifiers within a configurable and extensible architecture. By expressing CHC solving as a software verification task and managing diverse tools through COVERITEAM, CHCVERIF offers a novel and modular pathway to Horn clause verification.

The remainder of the paper is structured as follows. Section 2 provides background on portfolio-based verification. Section 3 describes our overall approach to verification. Section 4 presents our process of constructing the portfolio. In Section 5, we reflect on our results.

Emanuele De Angelis, Florian Frohn (Eds.): 12th Workshop on Horn Clauses for Verification and Synthesis (HCVS 2025)
EPTCS ??, 2025, pp. 1–12, doi:10.4204/EPTCS.??.??

2 Background

2.1 CHC to C transformation

Let us define a CHC problem as a set of *deduction rules* in the context of this work. A deduction rule consists of a premise and a consequence. The premise can have zero or more *uninterpreted functions*, while the consequence can have zero or exactly one.

If there are no uninterpreted functions in the premise, we call that rule an *atom*. If there is more than one uninterpreted function in the premise of a rule, we call that rule *non-linear*. If a deduction rule deduces the literal *false*, we call that rule a *query*.

If any rule is non-linear in a CHC problem, we call that problem *non-linear*. Otherwise, we say that the CHC is *linear*.

In practical applications, CHC problems are frequently represented using the SMT-LIBv2 format [11, 13]. This format depicts deduction rules as *imply* expressions over a specific SMT theory. Moreover, every variable involved in the deduction rules is subject to universal quantification across the entire domain of variables, as required by the respective SMT theories.

In this paper, we focus on the SMT theories *core*, *linear integer arithmetic*, and *fixed-size bitvectors*. 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.

In the following, we present a simple example to showcase the basic idea behind the CHC to C transformation from [26, 2].

$$\begin{aligned} A(x) &\leftarrow x = 1 \\ A(x) &\leftarrow A(x-1) \\ \mathit{false} &\leftarrow A(11) \end{aligned}$$

The first rule declares that $A(1)$ is *true*. The second rule propagates this fact forward, stating that if $A(x)$ holds, then $A(x+1)$ must also hold. Finally, the third rule is a *query* that leads to a contradiction if $A(11)$ holds. Intuitively, this system is unsatisfiable: by repeated application of the second rule, starting from $A(1)$, we deduce $A(2)$, $A(3)$, and so on up to $A(11)$. Thus, $A(11)$ is deduced, which triggers the contradiction, making the system unsatisfiable.

The CHC system can be encoded in software in a top-down or bottom-up manner [26]. In the backward (top-down) encoding in Listing 1, the verifier starts with the query $A(11)$ and recursively unfolds the rules until it reaches the exit condition. Note that this program might fall into an infinite recursion based on the value of the parameter of the function A. However, tools that can reason about recursion may find that the exit condition of $A(11)$ is reachable.

In contrast, the forward (bottom-up) version shown in Listing 2 constructs the program starting from the facts. It iteratively applies the rules using a loop and a nondeterministic choice of the current state. By explicitly modeling the inference steps, this version avoids recursion and can often be more amenable to automated reasoning by tools that support nondeterminism and state exploration, but in return, it only works with linear CHCs.

2.2 Algorithm selection and portfolios

Formal verification, in general, is a resource-intensive task that involves a variety of analysis techniques, such as symbolic execution, bounded model checking, abstract interpretation, and others, each with

Listing 1: Backward transformation

```
1  int A(int x) {
2      if(x == 1) return 1;
3      else if(A(x - 1)) return 1;
4      else return 0;
5  }
6
7  int main() {
8      if(A(11)) return -1;
9      else return 0;
10 }
```

Listing 2: Forward transformation

```
1  int main() {
2      int A = 1, x;
3
4      while(true) {
5          x = nondet();
6          if(A == 11) return -1;
7          else if(A == x - 1) A = x;
8      }
9  }
```

its own strengths and limitations. Given the inherent variability in how different tools perform across verification problems, using portfolio-based and algorithm selection strategies has received increasing attention in recent years [7].

In the context of formal verification, the task of algorithm selection involves determining the most suitable algorithm, configuration, or tool from a set of options to solve a verification problem. The choice is typically based on characteristics derived from the problem, such as its structural details, control flow, data types, or recognizable patterns.

In contrast, a portfolio involves the execution of multiple verification tools or configurations on the same verification problem, either in parallel or in a coordinated sequence. The main idea is to make use of the complementary strengths of different verifiers, as no single tool consistently outperforms all others across the diverse landscape of verification tasks. While portfolios can combine configurations of a single tool, usually they are used to integrate tools using fundamentally different verification approaches (e.g., bounded model checking, abstract interpretation, or interpolation), improving overall robustness and increasing the likelihood of verification success within given time or resource bounds.

In recent years, the use of portfolio-based algorithm selection and techniques has grown significantly in the field of software verification [7]. Clear evidence of this fact is that approximately a third of the entries in SV-COMP 2025 [7] used some form of algorithm selection or portfolio in their verification approach. Entries include THETA [28] or CPACHECKER [5], for example, that use select different algorithms based on the input problem, or CPV [10] that uses a portfolio of other tools for verification.

2.3 COVERITEAM

COVERITEAM [4] is a framework for constructing and executing modular verification workflows by composing existing verification tools into customizable pipelines. It builds on three basic concepts: verification artifacts, actors, and compositional operators.

1. Verification artifacts depict the data in the workflow, such as the input files, the property to verify, the results, and witnesses.
2. Actors, on the other hand, act on the data and execute tools such as verifiers, validators, or test generators. They take verification artifacts as inputs and produce verification artifacts as outputs.
3. Compositional operators provide a way to tie multiple actors together by taking care of scheduling tasks and directing the verification artifacts between the actors.

COVERITEAM is tightly integrated with the toolchain of SV-COMP. It executes the actors by using BENCHEXEC [6] under the hood for reliable containerization and measurement. Moreover, as part of

the submission process to SV-COMP, tool authors must submit a tool entry file to the FM Tools repository[1] that COVERITEAM can consume as an actor definition. The tool entry file contains, among other information:

- contains details on where to download the tool from. It supports DOIs for Zenodo archives or URLs for direct repository artifacts;
- describes which command-line parameters to pass to the tool;
- determines which BENCHEXEC tool-info module to use that integrates the tool with the benchmark environment;
- lists dependencies that the system must have for the tool to run.

Leveraging this tight integration, COVERITEAM is able to execute any tool from the last couple of editions of SV-COMP out-of-the-box, and makes it easy and well documented to integrate additional tools with it.

3 Verification Approach

In this section, we present our approach to the verification of CHCs. Our main idea is based on our previous work in [26, 2] in which we presented a novel way to verify CHCs by transforming them into a C program (①). This transformation is done in such a way that the reachability of an assertion failure in the C program corresponds to the satisfiability of the CHC. The C program is then verified using a software verification tool (②) that produces a correctness witness if the program is safe (assertion failure is not reachable), or a violation witness if the program is unsafe (assertion failure is reachable). Finally, the verdicts of the software verification tool are transformed into the CHC domain (③): a safe verdict implies a satisfiable CHC, and the model can be extracted from the correctness witness, while an unsafe verdict means an unsatisfiable CHC, and the refutation can be derived from the violation witness. The approach is summarized in Figure 1.

In this paper, we focus only on ②. More specifically, we aim to determine an optimal portfolio of software verification tools for C programs produced from CHCs. With this approach, we aim to diversify the field of CHC solving further, especially for theories where tool support is sparse (e.g., bitvectors).

4 Portfolio construction

This section elaborates on the main ideas and the experiments conducted to determine the optimal portfolio. First, we discuss CHCs containing linear integer arithmetic and elaborate on how we overcame the challenges posed by the C standard for this category. Then, we explain our portfolio approach and present the results supporting our portfolios for CHCs containing bitvercors.

4.1 Linear integer arithmetic

The theory of linear integer arithmetic (LIA) deals with formulas over integer variables using linear expressions and supporting operations such as addition, subtraction, and comparisons, but excludes the multiplication of variables or non-linear terms.

[1] `https://gitlab.com/sosy-lab/benchmarking/fm-tools`

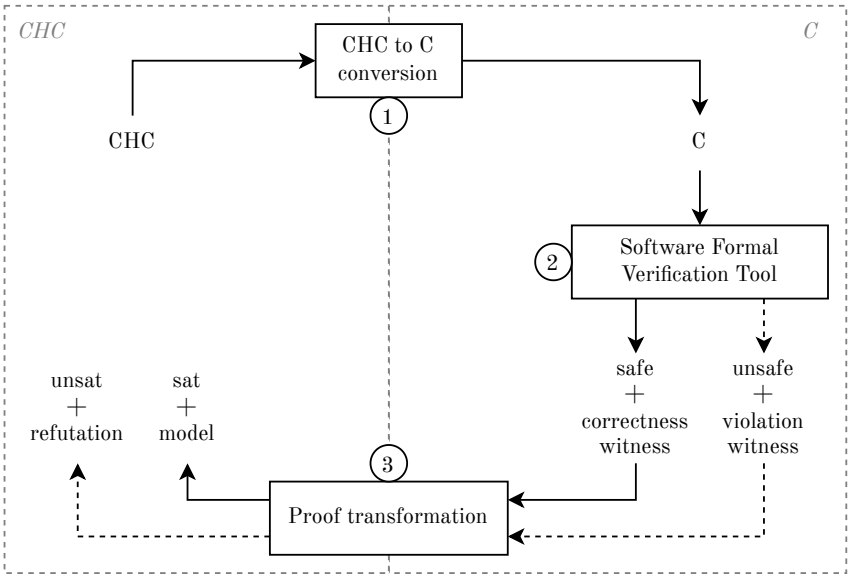

Figure 1: Overview of the proposed approach

One of the key differences between CHCs and C programs in handling integers is the way in which they are interpreted. CHCs use unbounded integers as they reason over the mathematical set of integers. C programs, on the other hand, are expected to run on hardware with finite resources. Thus, every value (in memory) is expressed over a finite number of bits, leading to a bounded domain.

This difference in semantics means that not all CHCs containing linear integer arithmetic can be mapped to a C program with equivalent behavior. Some CHCs might require the corresponding C program to evaluate expressions to values outside the possible domain; in other words, overflow. If the overflowing value is of an unsigned type, the value wraps around the domain, which does not match the behavior of such values in CHCs. On the other hand, if the overflowing value is of a signed type, the resulting behavior is undefined according to the C standard. Due to overflow, our approach can unfortunately be both unsound (with integer wraparounds causing infeasible traces) and incomplete (with a constrained integer domain causing incomplete models).

Fortunately, there are software verification tools that can determine whether a C program contains an overflow or not. Based on this observation, we propose the following structure for the portfolio (Figure 2). We first transform the CHC into a C program via the transformation introduced in [2]. Then, we run the reachability analysis on the C program that can produce three different verdicts:

1. If the verdict of the reachability analysis is safe, we check via an overflow analysis if the C program contains an overflow. If it does, the result is unsound, and the final verdict must be unknown. If there is no overflow in the problem, then we return a safe (satisfiable) result.
2. If the verdict of the reachability analysis is unsafe, we generate a test case from the violation witness. If the execution of the witness leads to an overflow, the result is unsound, and the final verdict is unknown. If there is no overflow, the witness corresponds to a sound refutation, and we return an unsafe (unsatisfiable) result.
3. If the verdict of the reachability analysis is unknown, we return that unknown verdict.

To determine which tools to use, we opted to run experiments. We transformed approximately 600

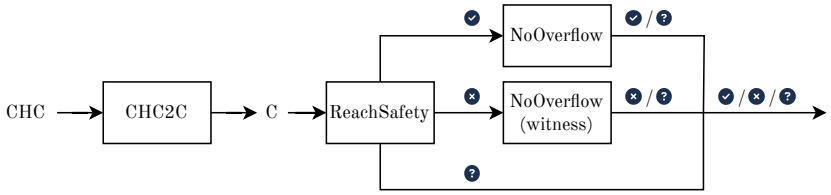

Figure 2: The proposed portfolio for the LIA theory

CHCs from the benchmark set[2] of CHC-COMP 2025 into C programs with both the recursive backward and the nonrecursive forward transformation. In the end, we ended up with 557 recursive C programs from the backward transformation of all CHCs and 195 nonrecursive C programs from the forward transformation of linear CHCs. We conducted our experiments using BENCHEXEC [6].

We selected 16 different tools based on the results of the NoOverflow category of SV-COMP 2025 [7]. We ran each benchmark with a time limit of 15 minutes, on 4 CPU cores and a memory limit of 15 GB of RAM. The results can be seen in Table 1. The first column denotes whether the C program resulted from the nonrecursive forward transformation or the recursive backward transformation. The verdict column describes the verdict given by the overflow checker, while the verdict type describes whether the given verdict was correct or not. The rest of the columns contain the number of overflow tasks solved with the given resources.

The only tool that produced wrong results was SVF-SVC, but it produced more incorrect verdicts than correct ones, so it is less than an ideal candidate tool for verification. There were also tools (e.g., GOBLINT, THETA) that were able to prove if a program did not contain overflow, but failed to find an instance of overflow during the experiment, also rendering them unsuitable candidates for this task. Taking also into account the tool diversity as well (UKOJAK, UTAIPAN, and UAUTOMIZER are the same tool family), we opted to use BUBAAK, SYMBIOTIC, UAUTOMIZER, and ESBMC-KIND for overflow analysis.

Table 1: The number of solved overflow tasks with LIA

Recursive	**Verdict**	**Verdict type**	goblint	mopsa	emergentheta	sv-sanitizers	thorn	theta	svf-svc	bubaak-split	cpachecker	2ls	ukojak	bubaak	utaipan	symbiotic	uautomizer	esbmc-kind	**Out of**
False	No overflow	confirmed	26	24	22	9	22	22	81	21	27	28	32	21	33	21	36	0	195
	Overflow	confirmed	0	0	0	0	0	0	0	61	65	75	86	92	93	93	96	105	
		wrong	0	0	0	0	0	0	105	0	0	0	0	0	0	0	0	0	
True	No overflow	confirmed	45	28	43	43	43	43	224	28	65	72	79	29	76	66	93	88	557
	Overflow	confirmed	0	0	0	0	0	0	0	9	68	177	217	263	276	280	288	289	
		wrong	0	0	0	0	0	0	304	0	0	0	0	0	0	0	0	0	

For the reachability analysis, we selected 20 tools based on the results of the ReachSafety category of SV-COMP 2025. We ran each benchmark with a time limit of 15 minutes, on 4 CPU cores and a memory limit of 15 GB of RAM, similarly to the overflow experiments. The results can be seen in Table 2. The first column denotes whether the C program resulted from the forward or backward transformation. The

[2] `https://github.com/chc-comp/chc-comp24-benchmarks`

Table 2: The number of solved reachability tasks with LIA

Recursive	Verdict	Verdict type	2ls	aise	brick	bubaak	bubaak-split	cpachecker	cpv	emergentheta	esbmc-kind	goblint	hornix	korn	mopsa	svf-svc	symbiotic	theta	thorn	uautomizer	ukojak	utaipan	Out of
False	safe	confirmed	14	3	0	115	118	43	70	43	51	11	12	0	0	9	127	2	73	111	85	85	195
		unconfirmed	0	0	0	20	20	0	0	1	1	0	0	0	0	0	22	0	0	2	1	1	
		wrong	0	0	0	18	22	0	0	0	1	0	0	0	0	0	37	0	4	5	4	2	
	unsafe	confirmed	15	24	0	19	7	25	26	13	21	25	0	0	0	0	0	20	24	28	27	27	
		unconfirmed	1	2	0	2	2	4	1	2	0	2	0	0	0	0	0	2	1	0	0	0	
		wrong	13	14	0	12	9	14	6	13	0	18	0	0	0	0	0	14	0	0	0	0	
True	safe	confirmed	21	6	0	24	25	60	96	19	18	21	0	13	13	12	336	23	24	19	132	132	557
		unconfirmed	0	0	0	0	0	0	0	0	0	0	0	0	0	0	50	0	0	0	0	0	
		wrong	0	0	0	0	0	2	3	0	0	0	0	0	0	0	142	0	1	0	0	0	
	unsafe	confirmed	13	2	0	80	41	61	69	8	11	110	0	7	1	0	0	84	24	12	107	101	
		unconfirmed	0	0	0	2	2	1	0	0	0	2	0	0	0	0	0	2	1	0	0	0	
		wrong	3	0	0	20	16	16	1	3	0	24	0	0	0	0	0	20	1	0	0	0	

Table 3: The number of solved CHC problems by CHC solvers for the LIA theory

Verdict	eldarica	golem	ultimateunihorn	spacer	Out of
sat	286	256	149	337	557
unsat	151	164	125	160	

verdict column describes the verdict given by the reachability checker, while the verdict type describes whether the given verdict was correct or not. An unconfirmed verdict means that the correctness of the verdict is unknown, as it was neither confirmed nor refuted by any of the CHC solvers. The remaining columns contain the number of reachability tasks solved with the given resources.

The table highlights a wide variance in tool performance. For nonrecursive tasks, tools like BUBAAK [9] and BUBAAK-SPLIT [8] show strong results with many confirmed safe verdicts (115 and 118 tasks, respectively). Taking unsafe verdicts into account as well, THORN [27] and UAUTOMIZER [19] rise as suitable candidates. Conversely, for recursive tasks, tools CPV [10], UKOJAK [25], and UTAIPAN [14] lead the charge. The number of unconfirmed and wrong verdicts varies across tools and verdict types, indicating differences in precision and reliability. In the end, we opted to use THORN, BUBAAK, and UTAIPAN for the nonrecursive programs, while CPV and UKOJAK for the recursive programs.

Finally, we used CPA-WITNESS2TEST [3] to validate the unsafe result. Since converting violation witnesses to test cases and running them is not a resource-intensive task, and CPA-WITNESS2TEST managed to handle every violation witness our tools produced, we opted not to explore other tools.

In comparison, we included the result of some CHC solvers on the same benchmark set in Table 3. It can be seen that while software verification tools tend to solve around 100 CHCs, the CHC solvers are in the magnitude of 300-400.

4.2 Bitprecise arithmetic

The verification of CHCs with linear integer arithmetic had to use a portfolio of both reachability and overflow analysis tools to bridge the gap between the semantics of mathematical integers and values in a C program. While CHCs with mathematical integers are prevalent in practice in the CHC-COMP benchmark suite, some CHCs use other theories, most notably bitvectors.

Bitvectors represent the values on a finite number of bits and, in 2's complement, capture the semantics of C values better than mathematical integers can. Unlike mathematical integers, bitvectors provide bit-level access to the value and wrap around when a value is out of their bounds.

While bitvectors are widely used, tool support in CHC solvers for bitvectors is sparse. Of the tools participating in CHC-COMP 2024, only THETA [26], and ELDARICA [20] were able to verify CHC problems with bitvectors. On the other hand, more than 25 tools participated in the ReachSafety-BitVectors category on SV-COMP 2024. Making use of these tools for CHC solving would significantly diversify solving capabilities.

As bitvectors capture the semantics of C values, there is no need to check for overflow or validate the result afterwards: the safety of the program implies the satisfiability of the CHC problem directly. Therefore, it is sufficient to perform a reachability analysis only for CHC problems with bitvectors.

We transformed the CHCs in the bitvectors category of the CHC-COMP 2024 benchmark suite[3], and ended up with 213 nonrecursive C programs and 396 recursive ones. We selected seven different tools based on the results of the ReachSafety-Bitvector category of SV-COMP 2025 [7]. We ran each benchmark with a time limit of 15 minutes, on 2 CPU cores and a memory limit of 15 GB of RAM. The results of the experiments are in Table 4.

Notably, in the nonrecursive setting, most tools achieve a relatively high number of confirmed unsafe verdicts, indicating strong effectiveness in proving unsatisfiability. Tools such as CPACHECKER [1], ESBMC-KIND [30], and SYMBIOTIC [22] show consistently high confirmed unsat results. The number of confirmed safe verdicts is more modest and varies across tools (especially taking wrong verdicts into account), with CPACHECKER achieving relatively higher counts. For recursive tasks, the landscape is similar, with the notable exception that only CPACHECKER was able to produce more than a couple of safe verdicts.

We also included the performance of two CHC solvers on the same benchmark set in Table 5 for comparison. It can be seen that while the CHC solvers still perform better, the gap is far less than previously with linear integer arithmetic. Moreover, the relatively high ratio of unconfirmed to confirmed verdicts shows that the software tools solved numerous tasks that none of the CHC solvers were able to. In the end, we opted to use a portfolio of CPACHECKER, ESBMC-KIND, and SYMBIOTIC.

4.3 Final portfolio

Based on the previous findings, we opted to use the following portfolio at the end. We start by first transforming the CHC into a C program nonrecursively. After that, based on the theory used, the workflow diverges. If the CHC uses LIA, first, a reachability analysis is conducted with THORN, BUBAAK, and UTAIPAN in a parallel configuration (meaning the tools are running until one of them produces a result). Then, based on the verdict, the verdict is either validated via CPA-WITNESS2TEST or by an overflow check with BUBAAK, SYMBIOTIC, UAUTOMIZER, and ESBMC-KIND in a parallel configuration. If the CHC uses bitvectors, only a single reachability analysis is conducted by CPACHECKER, ESBMC-

[3] https://github.com/chc-comp/chc-comp24-benchmarks

Table 4: The number of solved reachability tasks for bitvectors

Recursive	Verdict	Verdict type	2ls	bubaak-split	cpachecker	emergentheta	esbmc-kind	symbiotic	theta	Out of
False	safe	confirmed	9	39	27	16	5	0	38	213
		unconfirmed	4	28	6	21	4	0	32	
		wrong	0	16	0	4	0	0	8	
	unsafe	confirmed	31	62	81	69	79	75	68	
		unconfirmed	5	8	12	3	10	10	3	
True	safe	confirmed	0	0	24	0	0	0	0	396
		unconfirmed	1	1	13	5	1	1	6	
		wrong	0	0	0	2	0	0	4	
	unsafe	confirmed	69	88	91	67	101	99	73	
		unconfirmed	4	9	10	0	18	16	4	

Table 5: The number of solved CHC problems by CHC solvers for the bitvector theory

Verdict	eldarica	theta	Out of
sat	103	44	396
unsat	160	161	

KIND, and SYMBIOTIC in a parallel configuration. If the analysis produces a result, the portfolio ends its run.

If the nonrecursive reachability or overflow analysis does not yield a result under a given time limit, the nonrecursive analysis is terminated. Next, the recursive backward transformation transforms the CHC into a C program. After that, the portfolio diverges again based on the theory used by the CHC. In the case of LIA, a reachability analysis by CPV and UKOJAK is followed by an overflow analysis with BUBAAK, SYMBIOTIC, UAUTOMIZER, and ESBMC-KIND in a parallel configuration. If the CHC uses bitvectors, only a single reachability analysis is conducted by CPACHECKER, ESBMC-KIND, and SYMBIOTIC.

We implemented the proposed portfolio in the open-source CHCVERIF [15] tool using COVERITEAM and using the archives of the aforementioned tools from SV-COMP 2024.

5 Discussion

Our results demonstrate the viability and potential of using software verification tools as a backend for CHC solving, particularly when supported by a carefully constructed portfolio. This strategy opens up new directions for verifying CHCs over theories with limited or no dedicated solver support, such as bitvectors.

Bitvectors, in particular, represent a challenging theory for traditional CHC solvers due to their low-level, word-based semantics and their close correspondence with hardware behavior. However, bitvectors are widely supported by state-of-the-art software verifiers. By translating CHCs into C code, our approach effectively leverages the mature tool ecosystem of software verification to handle such cases,

allowing us to shift the burden of reasoning about bit-level behavior onto tools that are already optimized for such tasks, such as CPACHECKER, ESBMC, and SYMBIOTIC.

In the case of the more established linear integer arithmetics, the overall picture is much more nuanced. First, the software verification tools performed better on the nonrecursive forward transformation that only supports linear CHCs, but are far from the performance of CHC solvers. The fact that the backward transformation performed worse is not surprising, as it introduces recursive functions into the verification process, which are considered a more difficult verification problem and are supported by fewer tools. However, the main limitation is the poor performance of the overflow tools. As these results need to be verified by an overflow check, the poor overflow analysis performance will impact the whole portfolio.

6 Conclusion

We presented CHCVERIF, a tool for solving Constrained Horn Clause (CHC) problems by translating them to C programs and leveraging existing software verification tools to check safety properties. This approach enables the reuse of mature verification infrastructures to tackle CHC benchmarks, particularly those involving bitvectors and low-level semantics. Our evaluation shows that while the method enjoys only moderate success with linear integer arithmetic due to semantic mismatches, it achieves modest success on bitvector benchmarks, demonstrating the potential of this translation-based approach for certain classes of CHC problems. In the future, we plan to extend CHCVERIF to support more theories, such as floating-point numbers that no CHC solvers support yet.

References

[1] D. Baier, D. Beyer, P.-C. Chien, M. Jankola, M. Kettl, N.-Z. Lee, T. Lemberger, M. Lingsch-Rosenfeld, M. Spiessl, H. Wachowitz & P. Wendler (2024): CPACHECKER *2.3 with Strategy Selection (Competition Contribution)*. In: *Proc. TACAS (3)*, LNCS 14572, Springer, pp. 359–364, doi:10.1007/978-3-031-57256-2_21.

[2] Levente Bajczi & Vince Molnár (2025): *Solving Constrained Horn Clauses as C Programs with CHC2C*. In Thomas Neele & Anton Wijs, editors: *Model Checking Software*, Springer Nature Switzerland, Cham, pp. 146–163, doi:10.1007/978-3-031-66149-5_8.

[3] Dirk Beyer, Matthias Dangl, Thomas Lemberger & Michael Tautschnig (2018): *Tests from Witnesses*. In Catherine Dubois & Burkhart Wolff, editors: *Tests and Proofs*, Springer International Publishing, Cham, pp. 3–23, doi:10.1007/978-3-319-92994-1_1.

[4] Dirk Beyer & Sudeep Kanav (2022): *CoVeriTeam: On-demand composition of cooperative verification systems*. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, pp. 561–579, doi:10.1007/978-3-030-99524-9_31.

[5] Dirk Beyer & M Erkan Keremoglu (2011): *CPAchecker: A tool for configurable software verification*. In: *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, Springer, pp. 184–190, doi:10.1007/978-3-642-22110-1_16.

[6] Dirk Beyer, Stefan Löwe & Philipp Wendler (2019): *Reliable benchmarking: requirements and solutions*. *International Journal on Software Tools for Technology Transfer* 21(1), pp. 1–29, doi:10.1007/s10009-017-0469-y.

[7] Dirk Beyer & Jan Strejček (2025): *Improvements in software verification and witness validation: SV-COMP 2025*. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, pp. 151–186, doi:10.1007/978-3-031-90660-2_9.

[8] M. Chalupa & C. Richter (2024): BUBAAK-SPLIT*: Split What You Cannot Verify (Competition Contribution)*. In: *Proc. TACAS (3)*, LNCS 14572, Springer, pp. 353–358, doi:10.1007/978-3-031-57256-2_20.

[9] M. Chalupa & C. Richter (2025): BUBAAK*: Dynamic Cooperative Verification (Competition Contribution)*. In: *Proc. TACAS (3)*, LNCS 15698, Springer, pp. 212–216, doi:10.1007/978-3-031-90660-2_14.

[10] P.-C. Chien & N.-Z. Lee (2024): CPV*: A Circuit-Based Program Verifier (Competition Contribution)*. In: *Proc. TACAS (3)*, LNCS 14572, Springer, pp. 365–370, doi:10.1007/978-3-031-57256-2_22.

[11] David R. Cok (2012): *The SMT-LIBv2 Language and Tools: A Tutorial*. Available at `https://smtlib.github.io/jSMTLIB/SMTLIBTutorial.pdf`.

[12] Jakub Daniel, Alessandro Cimatti, Alberto Griggio, Stefano Tonetta & Sergio Mover (2016): *Infinite-State Liveness-to-Safety via Implicit Abstraction and Well-Founded Relations*. In Swarat Chaudhuri & Azadeh Farzan, editors: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, *Lecture Notes in Computer Science* 9779, Springer, pp. 271–291, doi:10.1007/978-3-319-41528-4_15.

[13] Emanuele De Angelis & Hari Govind V. K. (2022): *CHC-COMP 2022: Competition Report*. In Geoffrey William Hamilton, Temesghen Kahsai & Maurizio Proietti, editors: *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* 373, pp. 44–62, doi:10.4204/EPTCS.373.5.

[14] D. Dietsch, M. Heizmann, D. Klumpp, F. Schüssele & A. Podelski (2023): ULTIMATE TAIPAN *2023 (Competition Contribution)*. In: *Proc. TACAS (2)*, LNCS 13994, Springer, pp. 582–587, doi:10.1007/978-3-031-30820-8_40.

[15] Mihály Dobos-Kovács & Levente Bajczi (2025): *chc2c-svcomp: Portfolio of software verification tools for solving systems of Horn clauses.*, doi:10.5281/zenodo.15283157.

[16] Zafer Esen & Philipp Rümmer (2022): *Tricera: Verifying C Programs Using the Theory of Heaps*. In: *2022 Formal Methods in Computer-Aided Design (FMCAD)*, pp. 380–391, doi:10.34727/2022/isbn.978-3-85448-053-2_45.

[17] Grigory Fedyukovich, Arie Gurfinkel & Aarti Gupta (2019): *Lazy but Effective Functional Synthesis*. In Constantin Enea & Ruzica Piskac, editors: *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings*, *Lecture Notes in Computer Science* 11388, Springer, pp. 92–113, doi:10.1007/978-3-030-11245-5_5.

[18] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli & Jorge A Navas (2015): *The SeaHorn verification framework*. In: *International Conference on Computer Aided Verification*, Springer, pp. 343–361, doi:10.1007/978-3-319-21690-4_20.

[19] M. Heizmann, J. Christ, D. Dietsch, E. Ermis, J. Hoenicke, M. Lindenmann, A. Nutz, C. Schilling & A. Podelski (2013): ULTIMATE AUTOMIZER *with SMTInterpol (Competition Contribution)*. In: *Proc. TACAS*, LNCS 7795, Springer, pp. 641–643, doi:10.1007/978-3-642-36742-7_53.

[20] Hossein Hojjat & Philipp Rümmer (2018): *The ELDARICA Horn Solver*. In: *2018 Formal Methods in Computer Aided Design (FMCAD)*, pp. 1–7, doi:10.23919/FMCAD.2018.8603013.

[21] Qinheping Hu, John Cyphert, Loris D'Antoni & Thomas W. Reps (2020): *Exact and approximate methods for proving unrealizability of syntax-guided synthesis problems*. In Alastair F. Donaldson & Emina Torlak, editors: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, ACM, pp. 1128–1142, doi:10.1145/3385412.3385979.

[22] M. Jonáš, K. Kumor, J. Novák, J. Sedláček, M. Trtík, L. Zaoral, P. Ayaziová & J. Strejček (2024): SYMBIOTIC 10*: Lazy Memory Initialization and Compact Symbolic Execution (Competition Contribution)*. In: *Proc. TACAS (3)*, LNCS 14572, Springer, pp. 406–411, doi:10.1007/978-3-031-57256-2_29.

[23] Jinwoo Kim, Qinheping Hu, Loris D'Antoni & Thomas W. Reps (2021): *Semantics-guided synthesis*. *Proc. ACM Program. Lang.* 5(POPL), pp. 1–32, doi:10.1145/3434311.

[24] Anvesh Komuravelli, Arie Gurfinkel & Sagar Chaki (2016): *SMT-based model checking for recursive programs*. *Form. Methods Syst. Des.* 48(3), p. 175–205, doi:10.1007/s10703-016-0249-4.

[25] A. Nutz, D. Dietsch, M. M. Mohamed & A. Podelski (2015): ULTIMATE KOJAK *with Memory Safety Checks (Competition Contribution)*. In: *Proc. TACAS*, LNCS 9035, Springer, pp. 458–460, doi:10.1007/978-3-662-46681-0_44.

[26] Márk Somorjai, Mihály Dobos-Kovács, Zsófia Ádám, Levente Bajczi & András Vörös (2024): *Bottoms up for CHCs: novel transformation of linear constrained horn clauses to software verification*. *arXiv preprint arXiv:2404.15215*, doi:10.4204/eptcs.402.11.

[27] C. Telbisz, L. Bajczi, D. Szekeres & A. Vörös (2025): THETA*: Various Approaches for Concurrent Program Verification (Competition Contribution)*. In: *Proc. TACAS (3)*, LNCS 15698, Springer, pp. 260–265, doi:10.1007/978-3-031-90660-2_22.

[28] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei & István Majzik (2017): *Theta: a Framework for Abstraction Refinement-Based Model Checking*. In Daryl Stewart & Georg Weissenbacher, editors: *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, pp. 176–179, doi:10.23919/FMCAD.2017.8102257.

[29] Scott Wesley, Maria Christakis, Jorge A. Navas, Richard J. Trefler, Valentin Wüstholz & Arie Gurfinkel (2022): *Verifying Solidity Smart Contracts via Communication Abstraction in SmartACE*. In Bernd Finkbeiner & Thomas Wies, editors: *Verification, Model Checking, and Abstract Interpretation - 23rd International Conference, VMCAI 2022, Philadelphia, PA, USA, January 16-18, 2022, Proceedings*, *Lecture Notes in Computer Science* 13182, Springer, pp. 425–449, doi:10.1007/978-3-030-94583-1_21.

[30] T. Wu, X. Li, E. Manino, R. Menezes, M. Gadelha, S. Xiong, N. Tihanyi, P. Petoumenos & L. Cordeiro (2025): ESBMC V7.7*: Efficient Concurrent Software Verification with Scheduling, Incremental SMT and Partial Order Reduction (Competition Contribution)*. In: *Proc. TACAS (3)*, LNCS 15698, Springer, pp. 223–228, doi:10.1007/978-3-031-90660-2_16.