Bottoms Up for CHCs: Novel Transformation of Linear Constrained Horn Clauses to Software Verification

Márk Somorjai Mihály Dobos-Kovács Zsófia Ádám
Levente Bajczi András Vörös
vori@mit.bme.hu
Department of Measurement and Information Systems
Budapest University of Technology and Economics

Constrained Horn Clauses (CHCs) have conventionally been used as a low-level representation in formal verification. Most existing solvers use a diverse set of specialized techniques, including direct state space traversal or under-approximating abstraction, necessitating purpose-built complex algorithms. Other solvers successfully simplified the verification workflow by translating the problem to inputs for other verification tasks, leveraging the strengths of existing algorithms. One such approach transforms the CHC problem into a recursive program roughly emulating a *top-down* solver for the deduction task; and verifying the reachability of a safety violation specified as a control location. We propose an alternative *bottom-up* approach for linear CHCs, and evaluate the two options in the open-source model checking framework THETA on both synthetic and industrial examples. We find that there is a more than twofold increase in the number of solved tasks when the novel *bottom-up* approach is used in the verification workflow, in contrast with the *top-down* technique.

1 Introduction

Constraint Horn Clauses (CHCs) are widely used in the field of formal verification both as a means for an intermediate representation [6, 10, 14] and as a specification language [1]. Conventionally, CHCs allow the specification of *deduction* problems using implication, allowing the formalization of rules that govern how atomic facts lead to more complex (*deduced*) information.

A CHC problem can be solved in many different ways. SPACER [9] in Z3 [15] uses a solver based on automatic under-approximating abstraction; ELDARICA [13] uses a direct abstract state space traversal over the CHC formulae; and UNIHORN [1] uses a translation to recursive Boogie [3] code before applying a conventional software verification workflow to achieve a result. While the former approaches in SPACER and ELDARICA work well as demonstrated by their performance in previous years' CHC-COMP [1], a competition for CHC solvers, they require purpose-built solvers, thus incurring additional effort when developing new algorithms.

In contrast, the approach utilized by UNIHORN relies on existing algorithms, taking advantage of the tool being part of the ULTIMATE framework with proven and efficient algorithms for tackling software verification tasks [12]. By complementing the framework with a new front-end for parsing and transforming CHC formulae, the same verification workflows can be applied to the CHC-based problems as well, enabling their efficient verification.

The transformation step used by UNIHORN creates Boogie code that roughly emulates a program capable of deducing the existence of facts necessary to reach some end goal (e.g., a safety violation). We refer to this approach as *top-down* [18] or *backward*.

In this paper, we introduce an alternative to the *backward* method, which creates a program that emulates a *bottom-up* solver [18] (i.e., starting from nondeterministic facts and trying to deduce a safety

Temur Kutsia, Daniel Ventura, David Monniaux &
José F. Morales (Eds.): LSFA 2023 + HCVS 2023
EPTCS 402, 2024, pp. 105–117, doi:10.4204/EPTCS.402.11

violation using the formulae). We implement this *forward* transformation to another formal representation of programs, the Control Flow Automaton (CFA), alongside with a *backward* transformation alternative, in THETA [11]. Our benchmarks show that using the proposed approach increased the number of successfully solved CHCs more than twofold on linear CHC verification tasks from the CHC-COMP benchmark suite [1].

This paper is structured as follows. In Section 2, we introduce the necessary background concepts. Then, in Section 3, we present our proposed *forward* transformation and the accompanying verification workflow, as well as the theory behind proof- and counterexample-generation. Finally, in Section 4, we present our experimental results comparing the effect of using the existing *backward* transformation versus the novel *forward* transformation on the performance of the verification workflow.

2 Background

In this section, we introduce the theoretical background for the paper, including *software verification*, *control flow automata* (CFAs), and *Counterexample-Guided Abstraction Refinement* (CEGAR).

2.1 Formal Software Verification

The goal of software verification is to mathematically prove certain properties of a program. One such property is the reachability of labelled control locations. A program is *unsafe* if such a location can be reached from the initial location of the program using a finite number of transitions; otherwise, it is *safe*. Due to the uncertainties and complexity of dealing with high-level programming languages, the input is first transformed into a formal representation [2]. *Model checking* is then often employed [8], which explores the state space of the program, thus verifying the reachability of the error states. While generally this problem is undecidable [17], and enumerating the state space naively is infeasible in practice [5], there exist efficient algorithms for solving a subset of the input tasks, such as the Counterexample-Guided Abstraction Refinement (CEGAR) technique [4].

2.1.1 Control Flow Automata

A *Control Flow Automaton* represents a program as a directed graph. Formally, a control flow automaton is a tuple $CFA = (V, L, l_0, E)$, where:

- V: A set of *variables*, where each $v \in V$ can have values from its domain D_v.
- L: A set of *locations*, where each *location* can be interpreted as a possible value of the program counter.
- $l_0 \in L$: The *initial location*, that is active at the start of the program.
- $E \subseteq L \times Ops \times L$: A set of transitions, where a transition is a directed edge going from one location in L to another, with a label $op \in Ops$, where Ops is a set of operations that can be executed as the program advances from one location to another. An $op \in Ops$ can be one of the following:
  - $v = expr$: An assignment of a variable, where the value of $v \in V$ becomes the evaluation of the right-hand side $expr$.
  - *havoc* v: A non-deterministic assignment of a variable, after which the value of $v \in V$ can be anything from its domain D_v.
  - $[cond]$: A *guard* operation, where *cond* is an expression that evaluates to a boolean value. The transition can only be executed if the *cond* in the *guard* evaluates to $true$.

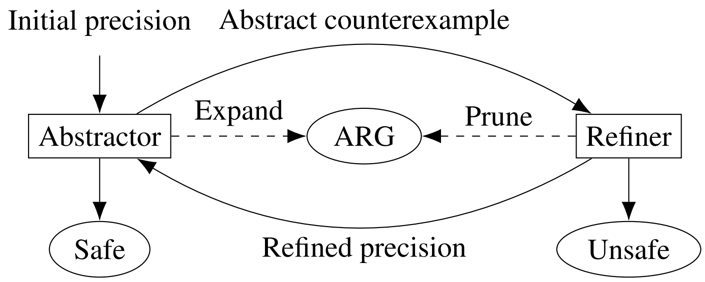

Figure 1: The CEGAR loop

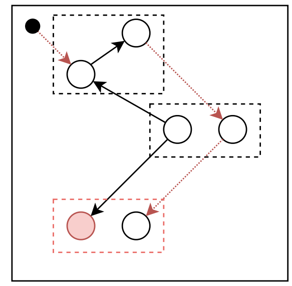

Figure 2: Abstract state space

In formal software verification, it is also useful to distinguish *error locations*, which are locations where the program would behave in an undesirable way, as well as *final locations*, which have no *outgoing transitions*.

The representation of program execution on the CFA consists of an alternating sequence of locations and operations, where at each location, the *state* of the CFA can be described as $S = (l_S, d_1, d_2, ..., d_n)$, where:

- $l \in L$ is the current location of the program,
- $d_1, d_2, ..., d_n$ are the values of all variables, that is $v_i = d_i, v_i \in V, d_i \in D_{v_i}$, for every $1 \leq i \leq |V|$.

All possible states of the CFA make up the *state space* of the program. The operations in an alternating sequence (representing an execution of the program) can then be interpreted as *transitions* in the state-space of the program.

2.2 Counterexample-Guided Abstraction Refinement

Counterexample-Guided Abstraction Refinement (CEGAR) [4] is an abstraction-based model checking algorithm.

The core of the algorithm is the CEGAR-loop (Figure 1), made up of two main parts: the *abstractor* and the *refiner*. The abstractor builds the *Abstract Reachability Graph* (ARG, a directed and acyclic graph containing abstract states and interconnecting transitions) using the *expand* operation and *covering* relation on abstract states. A parameter of abstraction is precision, which describes how much information about a concrete state is abstracted in the abstract state. An abstract state is an overapproximation of the possible concrete states (as seen in Figure 2), consequently, if no abstract error-state is reachable, then no concrete error-state is reachable, meaning the program is *safe*.

On the other hand, if an abstract error-state is reachable, the abstractor produces an *abstract counterexample*, starting at the initial abstract state and ending in an abstract error state. The refiner then decides whether a concrete error state is reachable in the abstract error state. If it can be reached, then the program is *unsafe*, and the path from the initial location of the CFA to a concrete error state is presented as a counterexample.

However, if a concrete error-state is not reachable, then the reachability of the abstract error state is a result of the overapproximation of abstraction, as demonstrated in Figure 2. Thus, the abstraction needs to be *refined* so that the abstract error state does not contain the unreachable concrete error state. This results in a refined precision, which is passed back to the abstractor after all unreachable abstract states are removed (*pruned*) from the abstract state-space.

The CEGAR loop is repeated until it either finds a concrete counterexample to the safety of the program or proves that no abstract error-state is reachable, that is, all nodes in the ARG are either expanded or covered. In the first case, the program is *unsafe*, while in the latter, it is *safe*.

3 Transforming Constrained Horn Clauses to Control Flow Automata

In this section, we present a novel approach of CHC to CFA transformation. The goal of this transformation is to create a CFA from a linear CHC in a way that turns the SMT problem of satisfiability in a CHC into a software verification question of erroneous state reachability in the CFA, so that model checking techniques can be used to decide both. More specifically, an erroneous state in a CFA should be reachable if, and only if the CHC is unsatisfiable. In this case, a refutation of the satisfiability should be given; otherwise a satisfying model ought to be generated. The approach is summarized in Figure 3.

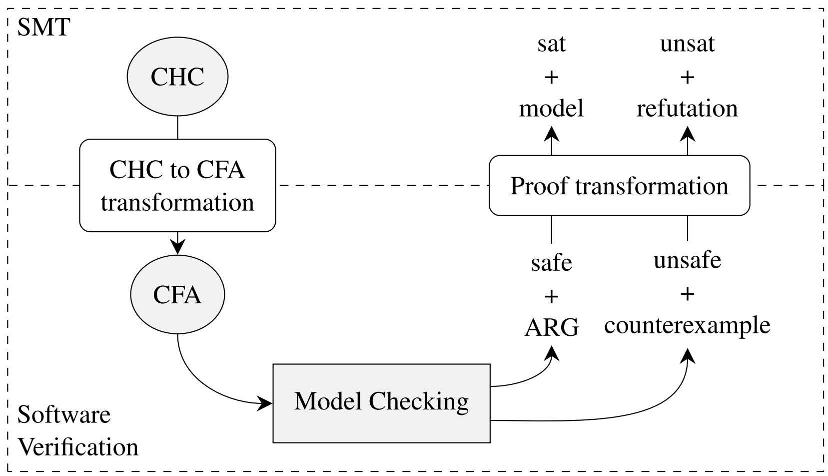

Figure 3: Overview of the presented work.

The transformation consists of two parts: the mapping of CHCs to CFAs, and the generation of a model/refutation from the output of model checking. These are represented in Figure 3 by the boxes *CHC to CFA transformation* and *Proof transformation*, respectively, and are not to be confused with *forward* and *backward* transformations described later on. As seen in the figure, proof transformation requires the utilized model checking algorithm to provide a counterexample when the CFA is deemed unsafe, and to produce an ARG when the CFA is safe.

The main idea behind the CHC to CFA transformation is to represent the uninterpreted functions as locations in the CFA, map CHCs to edges guarded by the conditions in the CHC, and use local variables to model the implications of deductions. The deducibility of a predicate with certain parameters can then be represented by the corresponding location's reachability during verification, with the given parameters as the variables' values. The source of the edges of fact CHCs can be the initial location of a CFA, since these do not have any preconditional predicates in their bodies. The target of the edge of a query CHC can then be an error location, which can only be reached if the conditions on an incoming edge are satisfied, similarly to how $\bot$ is deduced. If the error location can be reached from the initial location, then the counterexample contains the path of edges to it, which can then be mapped to their CHCs to show a sequence of CHCs that deduce $\bot$ from facts. On the other hand, if the error location is unreachable, then the explored abstract states can be used to define the uninterpreted functions to provide a satisfying model.

One way of approaching the problem of CHC satisfiability is to start with the facts, and try to apply the induction and query CHCs to deduce $\bot$. This is called the *forward* or *bottom-up* approach, which is what our main contribution, the *forward transformation* employs. Another approach is to recursively check what would be required to satisfy the body of the query CHC, stopping only when all requirements are satisfied by facts. We refer to this as a *backward* or *top-down* approach, which is used by ULTIMATE UNIHORN [1] to transform CHCs into program code.

An example CHC problem will be used throughout the chapter to demonstrate the transformations.

Example 1 *Consider the following CHC problem within integer arithmetic:*

$$A(n) \leftarrow n > 0 \wedge n < 100 \tag{1}$$
$$B(n,x) \leftarrow A(n) \wedge x > 0 \tag{2}$$
$$C(y,x) \leftarrow B(n,x) \wedge y = n - x \wedge y > 0 \tag{3}$$
$$A(n) \leftarrow C(y,x) \wedge n = y + (y \bmod x) \tag{4}$$
$$\bot \leftarrow A(n) \wedge n \geq 100 \tag{5}$$

The fact states that $A(n)$ needs to evaluate to true for $0 < n < 100$, while the satisfiability of the query depends on $A(n)$ being false for $n \geq 100$ and $n \leq 0$. What makes this problem non-trivial is the cyclic deductions between the predicates A, B and C: B can be deduced from A, C can be deduced from B, and A can be deduced from C under certain conditions. Trying a naive, manual deduction approach becomes a bit cumbersome here, due to the possibility of an infinite deduction cycle and the high number of combinations possible between the variables' values.

One may notice that n can not increase in the cycle since no matter what the subtracted x is, it will always be larger than the $y \bmod x$ that is added to n in a cycle. In the following, it will be shown that the problem is indeed satisfiable, by transforming it into a software verification problem and synthesizing a satisfying model from its proof. The CFA resulting from the transformation can be seen on Figure 4. The effect of each step on the CFA is explained as the steps are introduced.

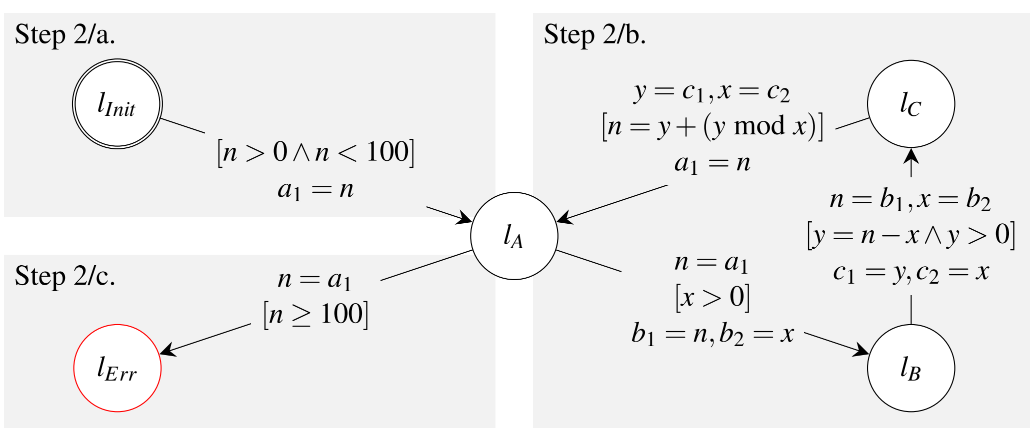

Figure 4: CFA of Example 1 after forward transformation.

3.1 Constrained Horn Clause Transformation

The transformation first creates the locations and variables of the CFA, then maps the CHCs to edges in different ways for fact, induction and query CHCs.

Consider the linear CHC problem with CHC set $\{C_1, C_2, \ldots, C_k\}$ over the following uninterpreted functions:

$$B_1(b_1^1, b_2^1, \ldots, b_{m_1}^1), B_2(b_1^2, b_2^2, \ldots, b_{m_2}^2), \ldots, B_n(b_1^n, b_2^n, \ldots, b_{m_n}^n)$$

That is, each CHC $C_l, \forall l \in \{1,2,\ldots,k\}$ takes one of the following three forms for some $i, j \in \{1,2,\ldots,k\}$:

$$\begin{aligned} B_i(x_1, x_2, \ldots, x_{m_i}) &\leftarrow \varphi_l, \\ B_i(x_1, x_2, \ldots, x_{m_i}) &\leftarrow B_j(y_1, y_2, \ldots, y_{m_j}) \wedge \varphi_l, \\ \bot &\leftarrow B_j(y_1, y_2, \ldots, y_{m_j}) \wedge \varphi_l, \end{aligned}$$

where φ_l is the interpreted formula in the body of C_l. As before, CHCs in these forms are referred to as facts, inductions and queries, respectively.

Step 1. Create CFA locations and variables

The uninterpreted functions $B_1(b_1^1, b_2^1, \ldots, b_{m_1}^1)$, $\ldots$, $B_n(b_1^n, b_2^n, \ldots, b_{m_n}^n)$ are mapped to the $CFA = (V, L, l_{Init}, E)$, where:

- $V = \{b_j^i \,|\, \forall i \in \{1,2,\ldots,n\} : \forall j \in \{1,2,\ldots,m_i\}\}$,
- $L = \{l_{Init}, l_{Err}, l_1, l_2, \ldots, l_n\}$,
- l_{Init},
- $E = \varnothing$.

Semantically, a new location is created for each uninterpreted function, along with an initial location l_{Init} and a distinguished error location l_{Err}. In addition, a unique variable is created for each parameter in every predicate. It is worth noting that the edge set is empty at this point, because edges are added in the next step of the transformation.

The motivation behind creating a location and variables for every uninterpreted function is that this way, a location's reachability with certain variable values can be directly mapped to the predicate's evaluation with said variable values as parameters: if a location l_i representing C_i is reachable with some values for variables $b_1^i, b_2^i, \ldots, b_{m_i}^i$, then $C_i(b_1^i, b_2^i, \ldots, b_{m_i}^i)$ should evaluate to true. On the other hand, if l_i can not be reached with variables $b_1^i, b_2^i, \ldots, b_{m_i}^i$, then $C_i(b_1^i, b_2^i, \ldots, b_{m_i}^i)$ ought to evaluate to false.

Example 2 *From Example 1, the first step of the forward transformation would create the* $CFA = (V, L, l_{Init}, \varnothing)$*, with locations* $L = \{l_{Init}, l_{Err}, l_A, l_B, l_C\}$ *and variables* $V = \{a_1, b_1, b_2, c_1, c_2\}$*. The created locations can be seen in white on the CFA in Figure 4.*

Step 2. Create CFA edges

In this step, each CHC is transformed into an edge in the CFA created in Step 1. Each kind of CHC (fact, induction, query) is treated differently, as described in the following subsections. The goal of this mapping is for the transition on the edge to only be possible, when the head of the CHC is deducible from the body of it.

Step 2/a. Create fact edges

For each fact CHC $C_l : B_i(x_1, x_2, \ldots, x_{m_i}) \leftarrow \varphi_l$ where $i \in \{1,2,\ldots,n\}$, an edge is created from the initial location l_{Init} to l_i, the location representing B_i. The labels on the created edge consist of the following, in the specified order:

- φ_l, the interpreted formula in the CHC's body as a guard,
- $b^i_1 = x_1, b^i_2 = x_2, \dots, b^i_{m_i} = x_{m_i}$, assignment of the passed values to the variables corresponding to the input parameters.

Fact CHCs are named facts because they can be deduced just from the background theory $\top$, when the interpreted formula φ_l is true. The created edge from the initial location mimics this, since the target of an edge will be reachable from the initial location when the guard φ is true.

To put it more formally, the head of a fact CHC $B_i(x_1,x_2,\dots,x_{m_i})$ is only deducible when its body, the interpreted formula φ_l is true. Similarly, the location l_i is only reachable from the initial location l_{Init} of the CFA using the created edge, when its guard φ_l evaluates to true. Furthermore, the parameters $x_1,x_2,\dots,x_{m_i}$ are assigned to $b^i_1,b^i_2,\dots,b^i_{m_i}$, meaning that the constraints of φ_l on the parameters are applied to the variables related to the location, just as they are applied when deducing $B_i(x_1,x_2,\dots,x_{m_i})$. Thus, we can conclude that l_i is only reachable using the created edge with variables $b^i_1,b^i_2,\dots,b^i_{m_i}$ valued $x_1,x_2,\dots,x_{m_i}$, when $B_i(x_1,x_2,\dots,x_{m_i})$ is deducible using C_l.

Example 3 *In Example 1, the second step of the forward transformation for fact CHCs would create the edge $e = (l_{Init}, op, l_A)$ from Equation 1, where the guard of op would be $n > 0 \land n < 100$, and the assignments would consist of $a_1 = n$, since a_1 is the variable corresponding to the first (and only) parameter of the predicate A. The created edge can be seen in the top-left gray rectangle on the CFA in Figure 4.*

Step 2/b. Create induction edges

For each induction CHC $C_l : B_i(x_1,x_2,\dots,x_{m_i}) \leftarrow B_j(y_1,y_2,\dots,y_{m_j}) \land \varphi_l$ where $i, j \in \{1,2,\dots,n\}$, an edge is created from l_j (the location representing B_j) to l_i (the location representing B_i). The labels on the created edge consist of the following, in the specified order:

- $y_1 = b^j_1, y_2 = b^j_2, \dots, y_{m_j} = b^j_{m_j}$, assignment of the variables corresponding to the input parameters of B_j to the passed values,
- φ_l, the interpreted formula in the CHC's body as a guard,
- $b^i_1 = x_1, b^i_2 = x_2, \dots, b^i_{m_i} = x_{m_i}$, assignment of the passed values to the variables corresponding to the input parameters of B_i.

In addition to the first assignments, $x_1,x_2,\dots,x_{m_i}$ and all variables in φ_l need to be uninitialized with a *havoc* statement to ensure that the semantics of $\forall$ in the CHCs are kept. However, the *havoc* statements are omitted from the examples for ease of readability. The order of instructions is also important: the assignments from the source location's variables need to happen before φ_l is evaluated.

Induction CHCs embody deductions from their bodies to their heads with some conditions φ_l. Assuming that l_j could have only been reached if it is deducible with some parameters, then this edge resembles the same: one can only go to l_i from l_j when φ_l is true.

More formally, the head of an induction CHC $B_i(x_1,x_2,\dots,x_{m_i})$ is only deducible, when φ_l is true and $B_j(y_1,y_2,\dots,y_{m_j})$ is deducible. Similarly, the location l_i can only be reached from l_j once l_j has been reached and the guard φ_l evaluates to true. Furthermore, the variables $b^j_1,b^j_2,\dots,b^j_{m_j}$ are assigned to $y_1,y_2,\dots,y_{m_j}$ and the parameters $x_1,x_2,\dots,x_{m_i}$ are assigned to $b^i_1,b^i_2,\dots,b^i_{m_i}$, meaning that the constraints of φ_l are applied to the y parameters and the b^i variables related to the location l_i, just as they are applied when deducing $B_i(x_1,x_2,\dots,x_{m_i})$ from $B_j(y_1,y_2,\dots,y_{m_j})$. Thus, we can conclude that l_i is only reachable using the created edge with variables $b^i_1,b^i_2,\dots,b^i_{m_i}$ valued $x_1,x_2,\dots,x_{m_i}$ from l_j with variables $b^j_1,b^j_2,\dots,b^j_{m_j}$ valued $y_1,y_2,\dots,y_{m_j}$, when $B_i(x_1,x_2,\dots,x_{m_i})$ is deducible from $B_j(y_1,y_2,\dots,y_{m_j})$ using C_l.

Example 4 *From Example 1, the second step of the forward transformation for induction CHCs would create three edges from Equation 2, 3 and 4:*

- $e_1 = (l_A, op_1, l_B)$ *for* $B(n,x) \leftarrow A(n) \land x > 0$*, where* op_1 *consists of the assignment* $n = a_1$*, then the guard* $x > 0$*, and the assignments* $b_1 = n, b_2 = x$ *at last,*
- $e_2 = (l_B, op_2, l_C)$ *for* $C(y,x) \leftarrow B(n,x) \land y = n - x \land y > 0$*, where* op_2 *consists of the assignments* $n = b_1, x = b_2$*, then the guard* $y = n - x \land y > 0$*, and the assignments* $c_1 = y, c_2 = x$ *at last,*
- $e_3 = (l_C, op_3, l_A)$ *for* $A(n) \leftarrow C(y,x) \land n = y + (y \bmod x)$*, where* op_3 *consists of the assignments* $y = c_1, x = c_2$*, then the guard* $n = y + (y \bmod x)$*, and the assignment* $a_1 = n$ *at last.*

The created edges can be seen in the right-hand side gray rectangle on the CFA in Figure 4.

Step 2/c. Create query edges

For each query CHC $C_l : \bot \leftarrow B_j(y_1, y_2, \ldots, y_{m_j}) \land \varphi_l$ where $j \in \{1,2,\ldots,n\}$ an edge is created to the error location l_{Err} from l_j, the location representing B_j. The labels on the created edge consist of the following, in the specified order:

- $y_1 = b^j_1, y_2 = b^j_2, \ldots, y_{m_j} = b^j_{m_j}$, assignment of the variables corresponding to the input parameters to the passed values,
- φ_l, the interpreted formula in the CHC's body as a guard.

The bodies of CHC queries should not be deducible, otherwise $\bot$ can be deduced and the problem is unsatisfiable. This behaviour is captured by the created edge: if the edge's source is reachable with values that make the guard of the edge true, then the error location is reachable, making the program unsafe.

In a formal way, the head of the query CHC $\bot$ is only deducible when both $B_j(y_1, y_2, \ldots, y_{m_j})$ is deducible, and φ_l is true. Similarly, the error location l_{Err} can only be reached from l_j once l_j has been reached and the guard φ_l evaluates to true. Furthermore, the variables $b^j_1, b^j_2, \ldots, b^j_{m_j}$ are assigned to $y_1, y_2, \ldots, y_{m_j}$, meaning that the constraints of φ_l are applied to the y parameters, just as they are applied when deducing $\bot$ from $B_j(y_1, y_2, \ldots, y_{m_j})$. Thus, we can conclude that l_{Err} is only reachable using the created edge from l_j with variables $b^j_1, b^j_2, \ldots, b^j_{m_j}$ valued $y_1, y_2, \ldots, y_{m_j}$, when $\bot$ is deducible from $B_j(y_1, y_2, \ldots, y_{m_j})$ using C_l.

Example 5 *In Example 1, the second step of the forward transformation for query CHCs would create the edge* $e = (l_A, op, l_{Err})$ *from Equation 5, where* op *would consist of the assignment* $n = a_1$ *and the guard* $n \geq 100$*. The created edge can be seen in the bottom-left gray rectangle on the CFA in Figure 4.*

To summarize, first a location l_i and variables $b^i_1, b^i_2, \ldots, b^i_{m_i}$ are created for each uninterpreted function $B_i(b^i_1, b^i_2, \ldots, b^i_{m_i})$, then all CHCs are transformed into edges. Since the edges are created in a way that l_i can only be reached with the corresponding variables $b^i_1, b^i_2, \ldots, b^i_{m_i}$ valued $x_1, x_2, \ldots, x_{m_i}$ if, and only if $B_i(x_1, x_2, \ldots, x_{m_i})$ can be deduced, we can conclude that the described transformation successfully converts the problem of satisfiability into a question of error location reachability. Thus, using a model checker to decide the latter will yield a result for the former as well: if the CFA is *unsafe*, the CHC problem is *unsatisfiable*; if the CFA is *safe*, the CHC problem is *satisfiable*.

It is worth to consider what the transformation results in, when there is no fact or query CHC in the set of CHCs. In the former case, there will not be any outgoing edges from the initial location of the

CFA. As a result, none of the locations will be reachable, meaning the predicates need not be true for any input, which can be expressed as $B_i \equiv false, \forall i \in \{1,2,\ldots,n\}$.

In the latter case, there will not be any edges going to the error location of the CFA. As a result, all locations are reachable in the abstract state $\top$, meaning the predicates can be true for any input, which can be expressed as $B_i \equiv true, \forall i \in \{1,2,\ldots,n\}$.

3.2 Proof Transformation

Proof transformation is the step of converting the result of the model checking algorithm to an answer to the CHC problem. This consists of two parts, depending on the result: the generation of a satisfying model from the ARG built during verification, or the creation of a refutation from the counterexample provided by the model checking algorithm.

3.2.1 Satisfying Model Generation

An SMT problem is called *satisfiable*, when a *model* (i.e., an assignment to constants) fulfilling all constraints exists. In the case of a CHC problem this means the definition of all uninterpreted functions $B_1(b_1^1, b_2^1, \ldots, b_{m_1}^1)$, $B_2(b_1^2, b_2^2, \ldots, b_{m_2}^2)$, …, $B_n(b_1^n, b_2^n, \ldots, b_{m_n}^n)$ present in the set of CHCs, that satisfy all of the CHCs.

The transformation in Subsection 3.1 ensures that a location l_i in the CFA can only be reached with the corresponding variables $b_1^i, b_2^i, \ldots, b_{m_i}^i$ valued $x_1, x_2, \ldots, x_{m_i}$ if, and only if $B_i(x_1, x_2, \ldots, x_{m_i})$ can be deduced. If a node $S_j = (l_i, L_1^j, \ldots, L_{k_j}^j)$ is present in the ARG, it means l_i has been reached under the condition $L_1^j \wedge \cdots \wedge L_{k_j}^j$. Consequently, it is guaranteed that B_i can be deducted under the condition $L_1^j \wedge \cdots \wedge L_{k_j}^j$. This is true for all $S^i = \{S_j \mid S_j = (l_i, L_1^j, \ldots, L_{k_j}^j)\}$ nodes in the ARG, therefore B_i needs to evaluate to true under either of their conditions, which can be represented by concatenating them with $\vee$. This gives the following the definition for $B_i, \forall i \in \{1,2,\ldots,n\}$:

$$B_i(b_1^i, b_2^i, \ldots, b_{m_i}^i) = \bigvee_{S_j=(l_i, L_1^j, \ldots, L_{k_j}^j)}^{S^i} \left(L_1^j \wedge \cdots \wedge L_{k_j}^j \right) \tag{6}$$

At the end of verification of a safe CFA, the ARG is fully expanded, i.e., all reachable abstract states have been visited and none are in an erroneous location. Furthermore, no erroneous state can be reached from any of the nodes in the ARG. Therefore the definitions provided by Equation 6 guarantee that there can not be a deduction to $\bot$, meaning they satisfy the CHC problem.

The type of information present in any L^j needs to be taken into consideration when defining the function. If L^j contains information about any other variable x then the variables $b_1^i, b_2^i, \ldots, b_{m_i}^i$ representing the input parameters of B_i, then unless some information about a b^i is dependent on x (e.g. $b_1^i > x$), L^j can be left out. If there is a dependent b^i, then x needs to be defined with a universal quantifier inside the function ($\forall x$).

Example 6 *Applying model checking with predicate abstraction to the CFA in Figure 4 may result in the Abstract Reachability Graph (ARG) seen in Figure 5. The regular arrows represent transitions between abstract states, while the dotted arrow denotes that the source abstract state is covered by the target abstract state.*

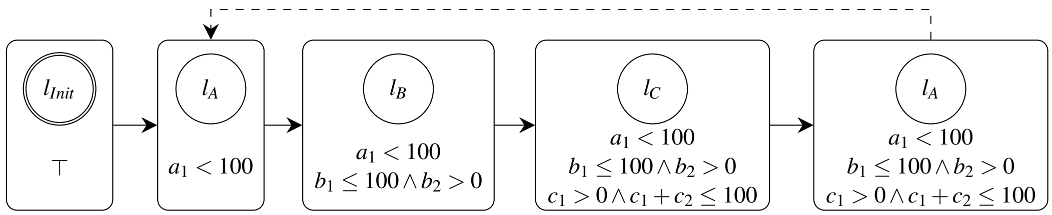

Figure 5: ARG resulting from the model checking of the CFA in Figure 4.

As described in Example 2, the uninterpreted function $A(n)$ corresponds to the location l_A and the variable a_1. Therefore its definition depends on the predicates of the abstract states that are in l_A, more specifically $(l_A, a_1 < 100)$ and $(l_A, a_1 < 100 \wedge b_1 \leq 100 \wedge b_2 > 0 \wedge c_1 > 0 \wedge c_1 + c_2 \leq 100)$. Using these states, we can define $A(n)$ as the disjunction of the predicates by converting a_i to n: $A(n) = n < 100 \vee (n < 100 \wedge b_1 \leq 100 \wedge b_2 > 0 \wedge c_1 > 0 \wedge c_1 + c_2 \leq 100), \forall b_1, b_2, c_1, c_2$. Since predicates of n do not depend on other variables, they can be left out, leading to $A(n) = n < 100 \vee n < 100 = n < 100$.

Similarly, $B(n,x)$ can be defined using abstract states that are in l_B, namely the single abstract state $(l_B, a_1 < 100 \wedge b_1 \leq 100 \wedge b_2 > 0)$. Converting b_1 and b_2 back to n and x gives $B(n,x) = a_1 < 100 \wedge n \leq 100 \wedge x > 0, \forall a_1$, which can also be simplified to $B(n,x) = n \leq 100 \wedge x > 0$ by omitting unused variables.

Lastly, $C(y,x)$ is defined using the abstract state $(l_C, a_1 < 100 \wedge b_1 \leq 100 \wedge b_2 > 0 \wedge c_1 > 0 \wedge c_1 + c_2 \leq 100)$. Converting c_1 and c_2 back to y and x results in $C(y,x) = a_1 < 100 \wedge b_1 \leq 100 \wedge b_2 > 0 \wedge y > 0 \wedge y + x \leq 100, \forall a_1, b_1, b_2$, which leads to the definition of $C(y,x) = y > 0 \wedge y + x \leq 100$ after getting rid of unused variables.

While it may not be trivial to see why this definition is a good model of the CHC problem, part of the reasoning is that using the definition of $A(n) = n < 100$, the query CHC Equation 5 takes the form $\bot \leftarrow n < 100 \wedge n \geq 100$. The body of this CHC is clearly unsatisfiable, thus, $\bot$ cannot be deduced.

3.2.2 Refutation Creation

When a CHC problem is unsatisfiable, a deduction can be found from the facts to $\bot$ that is always valid, regardless of how the uninterpreted functions are defined. The refutation is then a series of applications of the CHCs in the CHC set that start with a fact CHC and end with a satisfiable query CHC.

The counterexample provided by the model checker is an alternating sequence of concrete states of the CFA and edges. It starts at the initial location of the CFA with some values assigned to the variables and ends in the error location. The transformation described in Subsection 3.1 ensures that a location l_i in the CFA can only be reached with the related variables $b^i_1, b^i_2, \ldots, b^i_{m_i}$ valued $x_1, x_2, \ldots, x_{m_i}$ if, and only if $B_i(x_1, x_2, \ldots, x_{m_i})$ can be deduced. Consequently, all predicates corresponding to the locations of the concrete states in the counterexample are deducible, with the valuations present in the concrete states as parameters. The transformation also creates a one-to-one mapping of CHCs and edges. Thus, mapping the edges in the counterexample back to their CHCs, with the values of variables in the concrete states substituted as parameters, amounts to a valid refutation of the CHC problem's satisfiability.

Example 7 *Since the motivating Example 1 is satisfiable, consider a modified version of it, in which the only fact is replaced with $A(n) \leftarrow n > 0 \wedge n \leq 100$. The forward generated CFA would be similar to the one in Figure 4, with the exception of the edge going from l_{Init} to l_A having $n \leq 100$ instead of $n < 100$ in its guard.*

The model checking algorithm would return the following counterexample, with the irrelevant variable values omitted:

$$\begin{aligned}
&(l_{Init}, n = 100)\\
&(l_{Init}, ([n > 0, n \leq 100], a_1 = n), l_A)\\
&(l_A, n = 100, a_1 = 100)\\
&(l_A, (n = a_1, [n >= 100]), l_{Err})\\
&(l_{Err}, n = 100, a_1 = 100)
\end{aligned}$$

This could be mapped to the refutation below:

$$\begin{aligned}
A(n) &\leftarrow (n > 0 \wedge n <= 100) \wedge n = 100\\
\bot &\leftarrow (A(n) \wedge n \geq 100) \wedge n = 100
\end{aligned}$$

Since all variables have values assigned to them, it is trivial to check that this is indeed unsatisfiable.

4 Evaluation

Implementation The CHC to CFA transformation steps were implemented as ANTLR frontends [16] in the open-source model checking framework THETA [11]. The implementation is able to check the satisfiability of a CHC problem; however the generation of refutations and proofs is not implemented yet. Backward transformation was also implemented in a similar manner in the tool for comparison.

Goals and Design The aim of this evaluation is to show the effectiveness of the bottom-up approach by comparing it to the top-down approach. It also aims to study the performance of the approach with different configurations of CEGAR, e.g., different abstract domains.

The main comparison was done inbetween configurations of THETA only. Thus we were able to compare the different transformation approaches while the verification process was the same. Additionally, we also compared THETA to other state-of-the-art CHC solvers.

The implementation was evaluated on 585 linear CHCs over the background theory of linear integer arithmetic from the LIA-Lin track of the CHC-COMP21 benchmark repository[1]. The benchmarks were run on machines with 8 logical CPU cores and 16 GB of memory, with a timeout of 300 seconds.

domain	interpolation	pred-split	transformation	
			BACKWARD	FORWARD
EXPL	NWT_IT_WP	-	77	138
EXPL	NWT_WP_LV	-	82	137
EXPL	SEQ_ITP	-	81	175
PRED_BOOL	BW_BIN_ITP	WHOLE	110	288
PRED_CART	BW_BIN_ITP	WHOLE	141	302
PRED_SPLIT	SEQ_ITP	ATOMS	131	310
PRED_SPLIT	SEQ_ITP	WHOLE	142	318
PRED_SPLIT	BW_BIN_ITP	ATOMS	83	291
PRED_SPLIT	BW_BIN_ITP	WHOLE	114	328

Table 1: Number of solved tasks by certain configurations.

[1] https://github.com/chc-comp/chc-comp21-benchmarks

THETA (FW)	**328**
THETA (BW)	142
ELDARICA	337
UNIHORN	380
Z3	437

Table 2: Comparison to other tools.

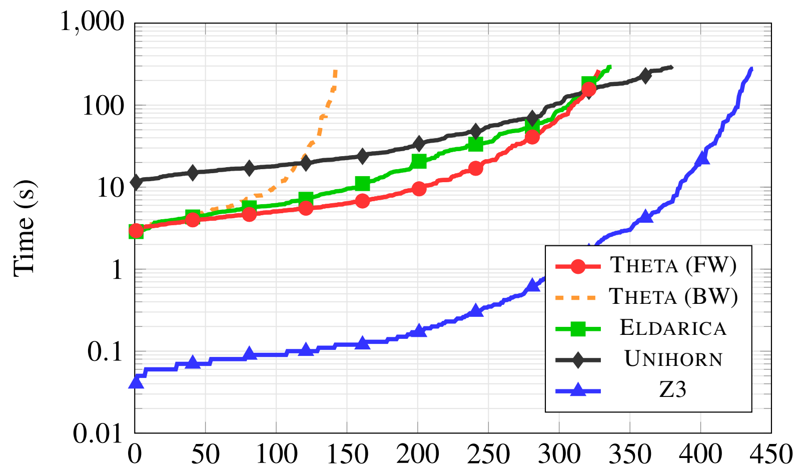

Figure 6: Number of solved tasks by tools under a certain time.

Results Table 1 shows the results of THETA with the different configuration options of THETA [11]. The results of the tool were either correct or timeout for all of the tasks.

Forward transformation proved to be far more effective than backward transformation in all configurations. The configurations using boolean predicate based abstraction with sub-state splitting (`PRED_SPLIT`) performed the best, with the other predicate based abstraction methods not too far behind.

The same benchmarks were also run with the top solvers of the LIA-Lin track from CHC-COMP21 [7], namely Z3, UNIHORN and ELDARICA. These solvers were run using their default configuration and with the same constraints as THETA. Table 2 shows the number of solved tasks compared to the best-performing configuration of THETA. Although THETA performs worse than the other solvers, its performance is comparable to ELDARICA's.

A quantile plot of the tools' performances can be seen on Figure 6. THETA performs better than both UNIHORN and ELDARICA for easier tasks, but it starts to get slower at a faster pace for tougher tasks than the other tools.

Conclusion As shown in Table 1, the performance of THETA was greatly improved by the forward transformation process for all of the tested configurations. This improvement gains even more significance when compared to other tools: just by changing the transformation method, THETA becomes a relevant competitor for some of the best linear CHC solvers of CHC-COMP. Based on our findings, we propose that tools employing software verification techniques for CHC solving implement our novel approach, to potentially significantly increase the number of successfully solved CHC problems.

Acknowledgement. We want to thank Martin Blicha for the help and fruitful discussion regarding the topic.

Funding. This research was partially funded by the ÚNKP-22-{2,3}-I New National Excellence Program and the 2019-1.3.1-KK-2019-00004 project from the National Research, Development and Innovation Fund of Hungary.

References

[1] Emanuele De Angelis & Hari Govind V K (2022): *CHC-COMP 2022: Competition Report*. *Electronic Proceedings in Theoretical Computer Science* 373, pp. 44–62, doi:10.4204/eptcs.373.5.

[2] Levente Bajczi, Zsófia Ádám & Vince Molnár (2022): *C for Yourself: Comparison of Front-End Techniques for Formal Verification*. In: *2022 IEEE/ACM 10th International Conference on Formal Methods in Software Engineering (FormaliSE)*, doi:10.1145/3524482.3527646.

[3] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs & K. Rustan M. Leino (2006): *Boogie: A Modular Reusable Verifier for Object-Oriented Programs*. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf & Willem-Paul de Roever, editors: *Formal Methods for Components and Objects*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 364–387, doi:10.1007/11804192_17.

[4] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu & Helmut Veith (2003): *Counterexample-Guided Abstraction Refinement for Symbolic Model Checking*. *J. ACM* 50(5), p. 752–794, doi:10.1145/876638.876643.

[5] Edmund M. Clarke, William Klieber, Miloš Nováček et al. (2012): *Model checking and the state explosion problem*. *Lecture Notes in Computer Science*, p. 1–30, doi:10.1007/978-3-642-35746-6_1.

[6] Zafer Esen & Philipp Ruemmer (2022): *TRICERA: Verifying C Programs Using the Theory of Heaps*. In: *CONFERENCE ON FORMAL METHODS IN COMPUTER-AIDED DESIGN–FMCAD 2022*, TU Wien Academic Press, pp. 360–391, doi:10.34727/2022/isbn.978-3-85448-053-2_45.

[7] Grigory Fedyukovich & Philipp Rümmer (2021): *Competition Report: CHC-COMP-21*. In Hossein Hojjat & Bishoksan Kafle, editors: *Proceedings 8th Workshop on Horn Clauses for Verification and Synthesis, HCVS@ETAPS 2021, Virtual, 28th March 2021*, *EPTCS* 344, pp. 91–108, doi:10.4204/EPTCS.344.7.

[8] Orna Grumberg, Doron A Peled & EM Clarke (1999): *Model checking*. MIT press Cambridge.

[9] Arie Gurfinkel (2022): *Program Verification with Constrained Horn Clauses (Invited Paper)*. In Sharon Shoham & Yakir Vizel, editors: *Computer Aided Verification*, Springer International Publishing, Cham, pp. 19–29, doi:10.1007/978-3-031-13185-1_2.

[10] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli & Jorge A. Navas (2015): *The SeaHorn Verification Framework*. In Daniel Kroening & Corina S. Păsăreanu, editors: *Computer Aided Verification*, Springer International Publishing, Cham, pp. 343–361, doi:10.1007/978-3-319-21690-4_20.

[11] Ákos Hajdu & Zoltán Micskei (2020): *Efficient Strategies for CEGAR-Based Model Checking*. *Journal of Automated Reasoning* 64(6), pp. 1051–1091, doi:10.1007/s10817-019-09535-x.

[12] Matthias Heizmann, Jochen Hoenicke & Andreas Podelski (2009): *Refinement of Trace Abstraction*. In Jens Palsberg & Zhendong Su, editors: *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*, *Lecture Notes in Computer Science* 5673, Springer, pp. 69–85, doi:10.1007/978-3-642-03237-0_7.

[13] 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.

[14] Yusuke Matsushita, Takeshi Tsukada & Naoki Kobayashi (2021): *RustHorn: CHC-Based Verification for Rust Programs*. *ACM Trans. Program. Lang. Syst.* 43(4), doi:10.1145/3462205.

[15] Leonardo de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In C. R. Ramakrishnan & Jakob Rehof, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[16] Terence Parr (2013): *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, doi:10.5555/2501720.

[17] A. M. Turing (1937): *On Computable Numbers, with an Application to the Entscheidungsproblem*. *Proceedings of the London Mathematical Society* s2-42(1), pp. 230–265, doi:10.1112/plms/s2-42.1.230.

[18] Jeffrey D. Ullman (1989): *Bottom-Up Beats Top-Down for Datalog*. In Avi Silberschatz, editor: *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 29-31, 1989, Philadelphia, Pennsylvania, USA*, ACM Press, pp. 140–149, doi:10.1145/73721.73736.