Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Abstraction Based Techniques for Constrained Horn Clause Solving

Scientific Students' Association Report

Author:

Márk Somorjai

Advisors:

Mihály Dobos-Kovács
Levente Bajczi
Dr. András Vörös

2022

Contents

Kivonat

A biztonságkritikus rendszerek szoftveres komponensei egyre komplexebbek. Egy biztonságkritikus rendszerben fellépő hiba hatalmas gazdasági veszteségekkel, környezeti károkkal vagy akár életvesztéssel járhat, emiatt az ilyen rendszerek helyességét biztosítani kell. Hagyományos tesztelési technikák nem tudnak kimerítőek lenni, így más módszerekre van szükség. A formális verifikáció matematikailag precíz bizonyítékot vagy cáfolatot tud előállítani a program biztonsági tulajdonságairól, mint például a hibaállapot elérhetősége.

A formális verifikáció programok formális modelljein működik, ezen formalizmusok egyike a Control Flow Automaton (CFA). Ez egy gráfszerű reprezentációja a programoknak, amelyben a hibahelyek a gráf adott csomópontjaiként jelennek meg, melyek elérhetetlenségét kell bizonyítani. Egy másik széleskörűen használt köztes nyelv a verifikációs folyamatokban a Constrained Horn Clauses (CHCs). Ezek a programokat és kívánt tulajdonságaikat az elsőrendű logikai formulák egy jól definiált részhalmazában írják le változókkal és nem-interpretált függvényekkel. Ebben a reprezentációban a helyesség kérdése a formulák kielégíthetőségeként jelenik meg, melyet a logikai formulák kielégíthetőségének eldöntésére tervezett SMT megoldók tudnak meghatározni.

A hibaállapotok elérhetőségének megítélésére CFA-ban egy elterjedt megoldás a CHC-re való konverziójuk, melyben SMT megoldók segítségével lehet kielégítő hozzárendelést találni. Ezen feladat azonban nehéznek bizonyul az SMT megoldók számára, köszönhetően az exponenciális számú lehetséges értékadásnak a változók számához viszonyítva. Ebben a munkában egy fordított megközelítés, a CHC megoldásának CFA-vá transzformált verziójában való keresése kerül bemutatásra. Ezen reprezentációban ugyanis elérhetővé válnak absztrakció-finomítás alapú modellellenőrzési algoritmusok, melyek absztrakció segítségével csökkentik a lehetséges értékadások terét, ezzel potenciálisan jelentős hatékonyságnövelést érve el a hagyományos CHC megoldási technikákhoz képest. A megközelítés szintetikus és ipari példákon is kiértékelésre kerül.

Abstract

Software components of safety-critical systems are becoming more and more complex. Failure in a safety-critical system can lead to enormous financial loss, environmental damage, or even loss of life; thereby, the correctness of such systems needs to be ensured. Conventional testing can not be exhaustive in reasonable time, leaving the need to prove the safety of programs in other ways. Formal verification takes on this task by providing a mathematically precise proof of correctness or refutation to the safety properties of programs, such as the reachability of an erroneous state.

Formal verification works on formal models of programs, one such formalism being the Control Flow Automaton (CFA). It is a graph-like notation to represent programs, in which erroneous locations map to specific nodes that should be unreachable. Another formalism that is widely used as an intermediate language in the verification process is Constrained Horn Clauses (CHCs). They can describe programs and their desired properties in a well-defined subset of first-order logic formulae on variables and uninterpreted functions. Therefore the question of correctness in this representation is embodied in the satisfiability of a query, which can be decided by SMT solvers designed to determine the satisfiability of mathematical formulae.

Conventionally, one approach to deciding reachability in CFAs has been to transform the model into CHCs and utilize an SMT solver to find a satisfying model. However, this task tends to be difficult for SMT solvers due to the exponential number of possible assignments with respect to the number of variables and predicates. In contrast, in this work, I propose an approach to solving CHCs by transforming them into a CFA, which provides access to powerful abstraction-refinement-based model checking algorithms. Such algorithms employ abstraction to reduce the space of possible assignments, which could potentially lead to significant improvements in efficiency compared to traditional CHC solving techniques. I evaluate my approach both on synthetic and industrial examples.

Chapter 1

Introduction

As the prevalence of electronics and digitalization keeps growing in the world, our dependence on software becomes stronger and stronger. This trend can be seen in many areas of our lives, one such area being safety-critical embedded systems: an increasing amount of tasks are performed by embedded software instead of physical components. Failures in these systems can cause environmental damage, substantial financial loss, or even loss of human lives. Therefore the correct behaviour of software components of such systems is imperative. One widespread approach to ensure functional correctness is formal verification, a technique aiming at mathematically establishing certain properties of programs. Traditionally, the programs and their desired properties are encoded in mathematical formulae [16], converting the verification problem into a question of the satisfiability of the generated formulae. The generated problems are called *Satisfiability Modulo Theory (SMT)* problems, and they play a crucial role in the safety of embedded systems.

Generally, a SMT problem is not decidable. A subset of SMT problems that is easier to handle while still being able to represent programs and their specific properties uses *Constrained Horn Clauses (CHC)*. The main characteristic of these clauses is that they are logical implications between uninterpreted functions, which makes them a suitable representation of deduction problems. The statements in Constrained Logic Programming are CHCs, and the provability of the goal is equivalent to the satisfiability of the set of statement CHCs. Despite the fact that CHC is only a subset of the general class of SMT problems, it is still very useful in many areas related to embedded systems: the declarative data representation language Datalog [24] also uses CHCs to store information [25], which has found its use in distributed knowledge databases [21] or memory representation in embedded software. The most widespread utilization of CHCs comes from software verification [15]. Many effective software verification tools are based on converting programs into CHCs, including the C verification frameworks Seahorn [16] and Tricera [12], and the Rust verification framework RustHorn [22]. Deciding the satisfiability of CHCs would therefore be beneficial in the verification of programs written in the aforementioned low-level languages that are widely used in embedded software.

The satisfiability of a set of CHCs depends on the premise of a special kind of CHC that has $\bot$ as its implication: if $\bot$ can be deduced, the problem is unsatisfiable. A trivial way of solving the satisfiability problem of CHCs then would be to apply all CHCs as long as they are applicable and check whether $\bot$ could be deduced. This approach would be infeasible in practice, because the space of possible applications would be enormous. On top of that, it would not necessarily terminate, due to CHCs that could be applied forever. One way of combating these issues would be to transform the problem into a

domain, which has algorithms that are equipped to deal with similar problems of huge state spaces and infinite cycles.

In this work, I propose an approach to efficiently solve CHC problems. My novel approach reduces the satisfiability problem of CHCs into a software verification problem of reachability. I devised a transformation that converts CHCs into *Control Flow Automata (CFA)*, a formal representation of programs. The conversion provides access to powerful abstraction-refinement based model checking algorithms that harness the capabilities of abstraction to reduce huge state spaces and handle infinite cycles. Additionally, a prototype of the transformation was implemented in the open-source model checking framework Theta [17], which is evaluated on a set of synthetic and industrial benchmarks.

The report is structured as follows: in Chapter 2, the necessary concepts and definitions are introduced, which the rest of the work builds upon. In Chapter 3, state-of-the-art approaches to solving satisfiability problems of CHCs are described. In Chapter 4, two different approaches to transforming CHCs to CFAs are presented: the first is my contribution, the second is an already existing approach. In Chapter 5, a prototype implementation of the presented transformations are evaluated, and their performance is compared to other approaches. Finally, in Chapter 6 my work is summarized.

Chapter 2

Background

To understand the presented work, some background knowledge is required about mathematical logic and software verification. This chapter presents the necessary concepts and definitions, as well as their interpretation in the context of the presented work.

2.1 Satisfiability Modulo Theories

Logic formulae are deeply embedded in the foundations of mathematics and computer science. They describe relations between values and variables using logic symbols. The symbols used and the domain of variables may vary based on the background of its consumer, which leads to the introduction of *Satisfiability Modulo Theories (SMT)* [4]. The interpretation of the variables and symbols of formulas in SMT is described by a background theory. For example, given the formula $a + b < 1 \wedge \neg(a < 0)$, the intended interpretation is for a and b to be numbers, $+$ to be the addition, $<$ to be the less-than relation over numbers.

A *model* of a formula is a set of assignments, where each variable in the formula is assigned a value exactly once, and substituting the variables to their values evaluates the formula to true. If a formula has at least one model, the formula is *satisfiable*. On the other hand, if no such model exists, the formula is *unsatisfiable*. The satisfiability of a formula in SMT is formalized in *Satisfiability Modulo Theories problems* [14], where the decision problem is whether a set of formulae can be satisfied within a given theory.

Definition 1 (Satisfiability Modulo Theory problem). A theory T, often referred to as the *background theory* is a (possibly infinite) set of sentences, which has at least one satisfying model. Given a formula φ with some interpreted symbols in T, φ is *satisfiable* if there is a model M^T that satisfies all sentences in T and $M^T \vDash \varphi$. If no such model exists, the formula is *unsatisfiable*.

Example 1. *For the formula $a + b < 1 \wedge \neg(a < 0)$ over the theory of linear integer arithmetic, $\{(a = 0); (b = 0)\}$ would be a satisfying model.*

SMT problems can be decided by specialized software called SMT solvers. They approach the problem in different ways, which makes them excel at different sets of theories, e.g. linear arithmetic, arrays, bit-vectors, etc.

Even though the interpretation of symbols in a formula is restricted to a background theory, SMT problems can still be undecidable within some theories [4]. In the following,

we focus on a subset of logic formulae, in order to make the decision problem easier by reducing the space of possible formulae.

2.2 Horn Clauses

Horn clauses are a fragment of logic formulae within some background theory. They can describe deduction problems well, and thus are widely used in the fields of logic programming and software verification.

Definition 2 (Horn Clause). A *Horn clause* is a disjunction of negated terms and, at most, one ponated term, where terms are logic relations applied to variables and values. It can have one of the following forms:

$$\forall X : (\neg\varphi_1 \vee \neg\varphi_2 \vee \cdots \vee \neg\varphi_m \vee \beta) \Leftrightarrow \forall X : ((\varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_m) \rightarrow \beta),$$
$$\forall X : (\neg\varphi_1 \vee \neg\varphi_2 \vee \cdots \vee \neg\varphi_m) \Leftrightarrow \forall X : ((\varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_m) \rightarrow \bot),$$
$$\forall X : \beta \Leftrightarrow \forall X : (\top \rightarrow \beta),$$

where

- $X = \{x_1, x_2, \ldots, x_k\}$ is a set of variables,
- $\varphi_1, \varphi_2, \ldots, \varphi_m$ and β are terms over X,
- and $\forall X(\cdot)$ denotes $\forall x_1 : (\forall x_2 : (\ldots \forall x_k(\cdot)) \ldots)$. ▪

In the following, the implication notation of Horn clauses will be used, that is, the right-hand side of the equivalences in each of the three forms, which contain only ponated terms.

2.2.1 Constrained Horn Clauses

The subset of Horn clauses can further be narrowed down by putting constraints on what can appear on each side of the implications, one such thing being *uninterpreted functions*.

Definition 3 (Uninterpreted function). An *uninterpreted function* F with input parameters $p_1, p_2, \ldots, p_n$ denotes a mathematical function that maps its inputs to a boolean value[1]. The function F evaluated with variables $x_1, x_2, \ldots, x_n$ is expressed as $F(x_1, x_2, \ldots, x_n)$, which is a term in a formula.

Uninterpreted functions will also be referred to as *predicates* in the rest of this work.

With the introduction of uninterpreted functions, the interpretation of a formula's model also needs to be extended. The model of a formula with predicates contains definitions of the uninterpreted functions, on top of the variable and value assignment pairs, that together evaluate the formula to true.

[1]Some definitions of uninterpreted functions allow mappings to non-boolean values as well. In this work, uninterpreted functions are only used in relation to CHCs, making boolean values an adequate restriction.

Example 2. *Given the uninterpreted function $F(x)$, the formula $F(1) \wedge \neg F(0)$ is satisfiable with the model $\{(F(x) := x > 0)\}$. On the other hand, the formula $F(1) \wedge \neg F(1)$ is unsatisfiable, because mathematical functions are one-to-one mappings by definition, therefore there is no mathematical function $F(x)$ that gives different outputs for the same inputs.*

The formulae on each side of an implication of a Horn clause can be divided into uninterpreted functions and interpreted formulae. This leads to the definition of *Constrained Horn Clauses* [26].

Definition 4 (Constrained Horn Clause). A *Constrained horn clause (CHC)* is a Horn clause on variables and uninterpreted functions in one of the following three forms:

$$\forall X : (B_1(X) \wedge B_2(X) \wedge \cdots \wedge B_n(X) \wedge \varphi \rightarrow H(X)), \tag{2.1}$$
$$\forall X : (B_1(X) \wedge B_2(X) \wedge \cdots \wedge B_n(X) \wedge \varphi \rightarrow \bot), \tag{2.2}$$
$$\forall X : (\varphi \rightarrow H(X)), \tag{2.3}$$

where

- X is a set of variables,
- $B_1(X), B_2(X), \ldots, B_n(X)$ and $H(X)$ are uninterpreted functions applied to variables in X,
- $\varphi = \varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_m$ is the conjunction of interpreted formulae over X. If $m = 0$, then the CHC takes the form of $\forall X : (\top \rightarrow H(X))$. φ is often referred to as the *condition*. ▪

The premise of the implication is called the *body* or *tail* of the CHC, while the consequence is referred to as the *head* of the CHC. Equation 2.1 is called an *induction*, Equation 2.2 is called a *query*, and Equation 2.3 is called a *fact* [5].

The SMT problem for a set of CHCs, the *CHC problem* is whether the uninterpreted functions have an interpretation that satisfies all of the clauses. If the set of CHCs contains one in query form, the question of satisfiability can be interpreted as whether the body of the query can be deduced. If an interpretation of the uninterpreted functions exists that makes the deduction of the query's body impossible, the set of CHCs are satisfiable. If no such interpretation exists, $\bot$ can be deduced, meaning that the CHC problem is unsatisfiable.

It is worth noting that if no query is present in the set of CHCs, then the problem is trivially satisfiable by defining all $B_1(X), B_2(X), \ldots, B_n(X)$ uninterpreted functions in the CHCs as $B_i(X) \equiv true, \forall i \in \{1, 2, \ldots, n\}$, independently from the parameters. With such a model, both fact and induction CHCs would take the form of body $\rightarrow true \Leftrightarrow \neg$body $\vee\, true \Leftrightarrow true$, which evaluates to true regardless of the body of the CHC.

On another note, if no facts are present in the set of CHCs, then the problem can once again be satisfied trivially, by setting all $B_1(X), B_2(X), \ldots, B_n(X)$ uninterpreted functions in the CHCs as $B_i(X) \equiv false, \forall i \in \{1, 2, \ldots, n\}$, independently from the parameters. With such a model, both induction and query CHCs would take the form of $false \rightarrow$

$false \Leftrightarrow \neg false \vee false \Leftrightarrow true \vee false \Leftrightarrow true$, which satisfies the CHCs regardless of the conditions.

Consequently, only sets of CHCs that include at least one fact and a query are considered to be CHC problems in the rest of this paper.

Example 3. *Given the uninterpreted function $F(x)$, consider the CHC problem over integer arithmetic with the following CHCs:*

$$\begin{aligned} \forall x : x = 0 &\rightarrow F(x) \\ \forall x, y : F(x) \wedge x \leq 1 \wedge y = x + 1 &\rightarrow F(y) \\ \forall x : F(x) \wedge x > 2 &\rightarrow \bot \end{aligned}$$

From the first CHC, we know that any satisfying model of the problem would have to define F at 0 to be true. No other value of x will satisfy the condition of this CHC, meaning we can not expect to gain additional information from it.

Evaluating the second CHC using our observation at $x = 0$ gives $F(y) \leftarrow F(0) \wedge 0 \leq 1 \wedge y = 0 + 1$, which means that F also needs to evaluate to true at 1. As a result, the evaluation of the second CHC at $x = 1$ is $F(y) \leftarrow F(1) \wedge 1 \leq 1 \wedge y = 1 + 1$, by which we can deduce that F needs to be true at 2 as well. Trying the same at $x = 2$ won't result in any new information though, since in the second CHC, the $x \leq 1$ condition no longer holds due to $2 \nleq 1$.

So far, we have deduced that a satisfying model would need to define F to be true at $x \in \{0, 1, 2\}$. As discussed above, the body of the third (query) CHC needs to be unsatisfiable for the CHC problem to be satisfiable. Negating the body gives $\neg(F(x) \wedge x > 2) \Leftrightarrow \neg F(x) \vee x \leq 2$, meaning that either x needs to be less than or equal to 2, or $F(x)$ needs to evaluate to false. This can be fulfilled along with our previous observations by a model $F(x) = x \leq 2 \wedge x \geq 0$, which is true for $0, 1, 2$, and false for any $x > 2$. Since a satisfying model can be found, the CHC problem is satisfiable.

If the $x > 2$ condition in the query were to be replaced with $x \geq 2$, then the body of the query would evaluate to true at 2, making $\bot$ deducible. In order to avoid this, F would need to evaluate to both true and false at 2. A mathematical function can not achieve that, therefore the modified problem would be unsatisfiable.

As it can be seen in the example above, the structure of CHCs makes them a perfect fit for deduction problems [15]. They can be used to describe knowledge, where based on some facts and rules, the deducibility of some statements can be queried. CHCs are also often used for software verification [26] [16] [22], where uninterpreted functions represent locations in a program, while the query encapsulates the goal of the verification, e.g. whether a path to an erroneous state is feasible or not.

From here on, the universal quantifier is conventionally emitted, and the head of a CHC is written before the tail. For example, $\forall X : B_1(X) \wedge B_2(X) \wedge \cdots \wedge B_n(X) \wedge \varphi \rightarrow H(X)$ is written as $H(X) \leftarrow B_1(X) \wedge B_2(X) \wedge \cdots \wedge B_n(X) \wedge \varphi$. It is important to keep in mind that even in this form, the variables are still local to each CHC, that is, a variable x appearing in different CHCs is a different variable due to the omitted universal quantifier.

2.2.2 Linear Constrained Horn Clauses

CHCs can be futher divided into two categories based on their linearity.

Definition 5 (Linearity of a CHC). A constrained Horn clause is *linear* if its body only contains, at most, a single uninterpreted function. If its body has two or more uninterpreted functions, the CHC is *non-linear*.

A CHC problem is called *linear* if every CHC in the problem set is linear. If any member of the set is non-linear, the CHC problem is also *non-linear*.

Example 4. *Given the variables* $X = x_1, x_2$ *and uninterpreted functions* $B_1(X), B_2(X)$ *and* $H(X)$*:*

- $H(X) \leftarrow B_1(X) \wedge x_1 > 0$ *is a linear CHC,*
- $H(X) \leftarrow B_1(X) \wedge x_1 > 0 \wedge x_2 < 3$ *is also linear CHC, with any number of additional interpreted formulae,*
- $H(X) \leftarrow B_1(X) \wedge B_2(X) \wedge x_2 < 3$ *is a non-linear CHC.*

Linear CHCs can arise from the verification of non-recursive programs [20], because the execution of a program goes from one location (represented by uninterpreted functions) to another and is never in two locations at once. On the other hand, knowledge deduction problems usually result in non-linear CHCs [25], since rules in these systems often involve multiple preconditional statements.

The main focus of the rest of this work is solving linear CHC problems using abstraction-based software verification techniques. In the following, the used formalisms and model checking techniques are introduced.

2.3 Formal Software Verification

The goal of software verification is to mathematically prove certain properties of a program. One such property is the safety of a program, that is whether or not an erroneous location can be reached in the program. 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*. To prove these properties *model checking* is often employed, during which the reachable states of the program are explored, and their erroneousness is decided. Due to the large state-space of programs, state reduction techniques are usually employed. A model checking algorithm using abstraction is described later in the section. But first, a formal representation of programs is introduced, which is often used in software verification.

2.3.1 Control Flow Automata

Software can take many shapes and forms, most notably, it can be represented as source code. While it is convenient for software development due to its readability, its usage in model checking can be complicated due to its complex syntax and semantics. For that purpose, a formal representation of the software is used, which allows for easier verification

of basic properties, such as *error reachability*. A formal representation that is often used to model programs is the *Control Flow Automaton*. [2]

A *Control Flow Automaton* represents a program as a directed graph, as described in the following.

Definition 6 (Control Flow Automata). 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 in 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*. ▪

In formal 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*, that is, transitions that are directed away from them.

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 $S = (l_S, d_0, d_1, ..., d_n)$, 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 $1 \leq i \leq |V|$.

The state of the CFA in its initial location is its *initial state*. The uninitialized values of variables at the beginning of the program depend on the programming language. In a language where uninitialized variables have the value of whatever memory garbage is at their assigned location in the memory, the values of variables would be non-deterministic. Therefore, the CFA of programs written in such languages may have many initial states. Other languages (such as Java) assign a default value to uninitialized variables, resulting in a single initial state of the CFA.

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.

2.3.2 Abstraction

The size of the state-space of a program presents the greatest challenge in software verification: just to represent all possible states with a single 32-bit integer variable 2^{32} states are needed, moreover, it grows exponentially with the number of variables present in the program. It goes without saying that checking the reachability of all states would be unfeasible, leaving the need for some kind of reduction technique on the state-space. One such technique is abstraction.

An *abstract state* is a set of states of the CFA with the same location, which stores information that is present in all said states by abstracting information away from the states of the CFA. A trivial way of achieving this is by not storing the values of certain variables. Another common technique is *predicate abstraction*, where boolean expressions about the variables are stored instead of their values (e.g. $v > 0 \wedge v < 10$). The word *predicate* in predicate abstraction does not denote an uninterpreted function, but rather an interpreted formula on variables. An abstract state can be represented by a tuple $S = (l, L_i, ..., L_j)$, where l is the location of the represented set of states, and $L_i, ..., L_j$ information is valid in all of the represented set of states.

Every state of the CFA is part of an abstract state. The set of all abstract states and the transitions between them is the *abstract state-space*, where a *transition* is an operation between two abstract states. To calculate all transitions going out from an abstract state, *expansion* is used. When an abstract state is *expanded*, a transition is created towards all abstract states that can be reached by performing an outgoing transition from the location of the abstract state.

Repeated expansion starting from the initial abstract state can be used to calculate the reachable abstract states. However, the expansion can easily become never-ending if there is a directed circle between abstract states due to repeated expansion of the same nodes. To avoid this, the *covering* relation is introduced.

If an abstract state $S_1 = (l_{S_1}, L_i, ..., L_j)$ has not yet been expanded, and another abstract state $S_2 = (l_{S_2}, L_k, ..., L_l)$ exists for which $l_{S_1} = l_{S_2}$ and $(L_i, ..., L_j) \implies (L_k, ..., L_l)$, then S_2 *covers* S_1 (or S_1 is covered by S_2). Semantically, this means that S_2 represents a larger set of CFA states than S_1, including all states represented by S_1, ensuring that the expansion of S_1 can not result in an abstract state that would not have been reachable from S_2 as well. Therefore the expansion of S_1 is pointless.

Using the definitions above, the *Abstract Reachability Graph (ARG)* can be built. Starting from the abstract state representing the initial state of the CFA, expansion is repeated on all nodes that are not covered by already expanded nodes. The resulting graph is a directed acyclic graph, where the nodes represent abstract states, and the directed edges represent transitions between those abstract states.

2.3.3 Counterexample-Guided Abstraction Refinement

Counterexample-Guided Abstraction Refinement (CEGAR) [8] is an abstraction-based model checking algorithm. It takes the formal representation of a program (such as a CFA) with distinguished error locations and decides whether or not the program is safe, that is, if the error locations are reachable from the initial location of the program. It does this by either exploring all reachable abstract states of the program and deeming them non-erroneous or by providing a counterexample to the program's safety, which is a concrete execution of the program in which an error-state is reached.

The core of the algorithm is the CEGAR-loop Figure 2.1, made up of two main parts: the *abstractor* and the *refiner*. The abstractor builds the ARG using the *expand* operation and *covering* relation on abstract states, as introduced in the previous section. A parameter of abstraction is precision, which describes how much information about a concrete state is abstracted in the abstract state. An abstract error-state is an overapproximation of the possible error-states, consequently, if no abstract error-state is reachable, then no concrete error-state is reachable, meaning the program is *safe*.

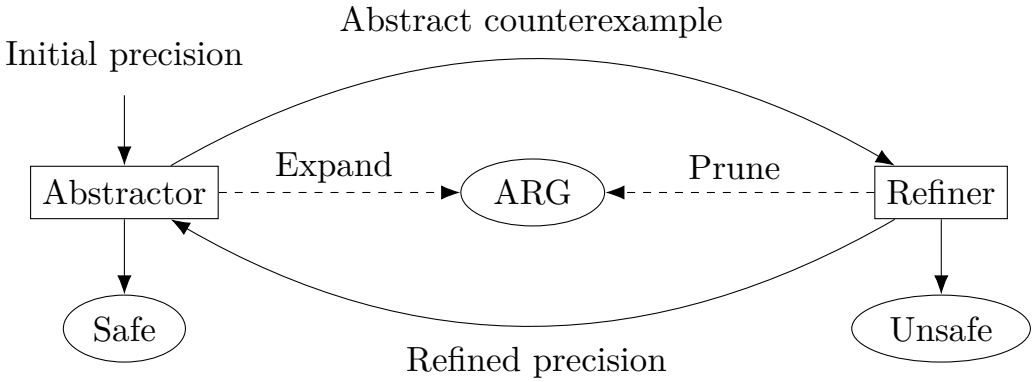

Figure 2.1: The CEGAR loop

On the other hand, if an abstract error-state is reachable, the abstractor produces an *abstract counterexample*, that is, an alternating sequence of abstract states and transitions between them, starting at the initial abstract state, ending in an abstract error-state. This is where the refiner comes in: it decides whether or not 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. 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*.

2.3.4 Verification with Procedures

Procedures are not inherently a part of CFAs and CEGAR. In this subsection, their adaptation to both is described, one after another.

2.3.4.1 Extensions to Control Flow Automata

Procedures are a well-known concept in software that allow modularity, more structured software, as well as the reuse of already written software. However, their semantics and usage can differ between languages and different domains, hence the following definition is introduced.

Definition 7 (Procedure). A *procedure* is an encapsulated part of software represented by the tuple $F = (I, O, body)$, where:

- I: A set of variables called *input parameters*, where each $v \in I$ can have values from its domain D_v.
- O: A set of variables called *output parameters*, where each $v \in O$ can have values from its domain D_v. They must be assigned a value in the *body*.
- *body*: The software encapsulated by the procedure, e.g. part of the source code.

Repeated execution of procedures does not preserve state, meaning all variables are uninitialized at the start of a new execution. ▪

Some programming languages support *inout* parameters, where a variable is passed into the procedure by reference, making all local modifications to a parameter apply to the outer variable as well. These variables can be replaced by an input and an output variable, therefore I chose not to distinguish them for the sake of simplicity.

The union of input parameters and variables that are defined in the body are called *local variables*. These variables only exist within the scope of the procedure and are uninitialized at the start of a new execution.

Procedures can be thought of as small programs: they have their initial and final locations, and they do not preserve the values of variables between executions. Therefore it comes naturally to represent them with a CFA, with the locations of the *body* and the operations between them as L and E, the initial location of the *body* as l_0, and the variables used in the *body* as V.

The use of procedures comes with the introduction of *procedure calls*.

Definition 8 (Procedure call). A *procedure call* is an operation in programs which initiates the execution of the body of a procedure. It can be represented by the tuple $C = (F, P, R)$, where:

- F: The procedure being *called*, the body of which will be executed.
- P: A set of expressions, that are assigned to the input parameters of the procedure (I_F), that is $v_i = p_i, v_i \in I_F, p_i \in P$, where $p_i \in D_{v_i}$ for $1 \leq i \leq |I_F|$.
- R: A set of variables, to which the output parameters of the procedure (O_F) will be assigned to, that is $r_i = v_i, v_i \in O_F, r_i \in R$, where the $v_i \in D_{r_i}$, for $1 \leq i \leq |O_F|$.

A *procedure call* consists of 3 steps:

1. The evaluations of the expressions in P are assigned to the input parameters of F.
2. Execution carries on from the initial location of the procedure's body until a final location of said body is reached.
3. The output parameters of F are assigned to the variables in R, after which execution continues from the location after the *procedure call*. ▪

It is important to note that calling a procedure essentially creates a new instance of it, meaning that if a procedure was called multiple times at the same time, the different

executions of the body would not operate on the same set of local variables. The names of the variables would be the same, but they would be different instances.

As procedure calls are introduced to software verification, complications arise. One is the aforementioned handling of different variable instances, but problems emerge with abstract states and their covering relation as well. An approach to handle these problems is introduced in the following.

2.3.4.2 Extensions to Model Checking

Procedures introduce procedure calls as a valid operation on CFA transitions, therefore they need to be handled during verification. This calls for slight changes in how the model checking algorithm works and what information abstract states store. In the following, I describe adjustments that can be used to support procedures and procedure calls in CEGAR.

Location stack

With procedures, the input of the model checking algorithm is no longer a single CFA, but several *Control Flow Automata (CFAs)*. The bridges connecting these CFAs are procedure calls: after a procedure call, execution carries on from the initial location of the called procedure's CFA. Calling is just one part of the task, though; the continued execution from the calling location, as the final location of the procedure's CFA is reached, also needs to be ensured. To mimic these properties of procedures during model checking, a *location stack* is used, similar to the call stack that is employed in programs.

A location stack P stores all locations from where procedure calls were made to reach the current location l_P. The current location l_P is always on the top of the stack. At the beginning, the stack stores the location that represents the entry point of the program. Afterwards, the stack is modified in the following three situations:

- With every transition in the CFA, the top location of the stack is replaced with the target of the transition. This guarantees that the current location is always on the top of the stack.
- Additionally, if a procedure call is present on the transition, the called procedure's initial location is *pushed* (placed on top) of the stack.
- If a final location of a procedure is reached, the top location is *popped* (removed) from the stack.

With these rules, it is ensured the desired properties of procedures are kept, as well as that the top location of the stack is always the current location. The only thing missing is the assignment of variables, which is discussed later in this section.

The introduction of location stacks means that a concrete state of a CFA is no longer defined by the values of its variables and the location. Instead of the location, the location stack is what can accurately represent a concrete state of a CFA, because it also stores the procedure calls (and the CFAs) through which the current location was reached.

This change also impacts abstract states. Previously, if two states of a CFA had the same location, then the difference between them could only be the evaluations of the variables present in the CFA. The addition of location stacks causes that to no longer be true since

a location can be reached through different procedure calls. Therefore, an abstract state also needs to be extended with a location stack. An abstract state can only represent concrete states with the same location stack as the abstract state.

The *expand* operation on abstract states needs adjustment as well. When an abstract state with the location stack P is *expanded*, a transition is created towards all abstract states that can be reached using an outgoing transition from l_P, the top location of P. The *covering* relation between abstract states also needs to be revised. If an abstract state $S_1 = (P_{S_1}, L_i, ..., L_j)$ has not yet been expanded, and another abstract state $S_2 = (P_{S_2}, L_k, ..., L_l)$ exists for which $l_{1_i} = l_{2_i}, l_{1_i} \in P_{S_1}, l_{2_i} \in P_{S_2}, 1 \leq i \leq max\{|P_{S_1}|, |P_{S_2}|\}$ and $(L_i, ..., L_j) \implies (L_k, ..., L_l)$, then S_2 *covers* S_1. This means that an abstract state can only cover another one if all locations in their locations stacks are equal.

Variable instances

Another desired property of procedures is their template-like behaviour, that is, new instances of their variables are created with every procedure call. One approach would be to copy the local variables uniquely with every procedure call.

However, the variables cannot be replaced on the CFA's transitions because there is only one CFA per procedure. Therefore an *instance mapping* is required, which associates a local variable with its uniquely copied version (*instance*). Note the use of *local variables*: global variables and output parameters do not need to be instantiated because, in the first case, there is just the single instance of them; in the second case their value can only be used in the next assignment anyway, so there is no point in managing separate versions of them. Using the mapping, local variables can be replaced by their mapped instances during verification when expressions are evaluated.

By default, *instance mappings* need to be created every time a procedure is called. However, due to the nature of CEGAR, previously instantiated versions of variables need to be accessible sometimes. This can happen, when the refiner creates a refined precision, and a new iteration of expansion starts. The refined precision contains information about previously created instances of variables that are used in comparison with the same variable versions' evaluations in the new iteration. One solution is to store the instance mappings associated with location stacks. This way, instances can be reused in procedures called from the same location stack, and the refined precision can be utilized.

To summarize, when a procedure is encountered during verification, the association of location stacks and instance mappings is checked. If an instance mapping exists for the location stack of the current state, then that mapping is used; if not, a new one is created with unique copies of the called procedure's local variables. This way, it is ensured that variables on a CFA transition can be replaced with their correct instances at any point during verification with CEGAR.

Parameter assignments

The last defined property of procedures that remains unaccounted for is parameters and their assignments. To address this, additional transitions can be created in the CFAs, with the assignments of parameters on them. Caution needs to be taken around which CFA to add these transitions to, and which version of variables to use.

Let $F_1 = (I_1, O_1, body_1)$ be the *outer* procedure, $F_2 = (I_2, O_2, body_2)$ be the *called* procedures, and let $C = (F_2, P, R)$ be a procedure call on a transition between locations l_i and l_j in $body_1$.

The output parameters of F_2 will be used by variables in F_1, for this reason it makes sense to assign them in $body_1$ after the procedure call. This can be done by the following:

1. A new location l_k is created.
2. The transition with the procedure call is moved so that it goes from l_i to l_k.
3. A new transition is created from l_k to l_j, with the assignments output parameters $r_i = v_i, v_i \in O_2, r_i \in R, 1 \leq i \leq |O_2|$ as an operation.

Since output parameters do not have versions (because their value is only used right after the procedure call), no further effort is needed to have their correct versions present in the outer procedure.

The input expressions are used by variables in F_2, therefore it makes sense to assign them in $body_2$, before the initial location. Unlike output parameters, input parameters do have versions, therefore additional care needs to be taken with their assignments. For each procedure call C with unique input expressions, the following needs to be done:

1. Each local variable of F_1 used in the input expressions $p_i \in P, 1 \leq i \leq |P|$ is replaced with a *prime version* of itself (e.g. $v \to v'$).
2. A new *initial parameter location* l_C is created.
3. A new transition is created from l_C to the initial location in F_2, with the assignments of input parameters $v_i = p_i, v_i \in I_2, p_i \in P, 1 \leq i \leq |I_2|$ as an operation, using the modified input expressions.

A mapping of the procedure calls associated with their freshly created l_C can be used during verification, to push the correct l_C on top of the location stack whenever a procedure call is encountered. During such an encounter, the *marked versions* of variables mapped to the instances of their original counterparts in F_1 also need to be passed onto the *instance map* of F_2, to allow the assignment of the outer procedure's local variables.

Chapter 3

Related work

This chapter covers state-of-the-art Constrained Horn Clause solvers that participate in the annual CHC solving competition, CHC-COMP[1]. The transformation described in Section 4.2 focuses on linear CHCs. Therefore the techniques used by the top solvers of the linear tracks of CHC-COMP21 [13] are described. First, the best-performing approach is introduced. Then, the second- and third-best approaches are presented, which share characteristics with the approach proposed in this work.

3.1 Bounded Exploration

Bounded methods in transition systems are iterative procedures that explore the state space to a certain number of steps in each iteration to check whether a property in question holds. A version of this algorithm that works with SMT problems with CHCs is Spacer [15] which is used by the prevalent SMT solver, z3 [9].

Spacer works by iteratively looking for a bounded deduction of $\bot$. Each time the algorithm fails to find a deduction of a fixed bound N, the reasons for failure are analyzed to derive consequences of the CHCs that explain why a deduction of $\bot$ must have at least $N + 1$ steps. This process is repeated until either $\bot$ is deduced, meaning the problem is unsatisfiable, or the consequences can be used to give satisfying definitions of the uninterpreted functions, making the problem satisfiable. Though this process may continue indefinitely, Spacer always makes progress by ruling out the possibility of increasingly longer refutations.

Spacer is used in z3 by software verification tools for low-level languages, such as Seahorn [16] for C and RustHorn [22] for Rust.

3.2 Counterexample-Guided Abstraction Refinement

Counterexample-Guided Abstraction Refinement (CEGAR) is a powerful model checking technique. The CHC solver Eldarica [19] uses a variant of CEGAR, in combination with predicate abstraction, to check the satisfiability of Horn clauses. It uses `princess` [23] as an underlying solver responsible for checking the feasibility of counterexamples.

Eldarica constructs an ARG that is built on CHCs, rather than locations of a CFA like the one described in Section 2.3.2. Nodes represent a set of pairs, consisting of an

[1] https://chc-comp.github.io

uninterpreted function of the CHC problem and a set of interpreted formulae on the parameters of the uninterpreted function. The edges of the ARG correspond to the CHCs themselves. The construction continues until there are no more CHCs that can be applied to introduce new nodes to the graph, at which point the ARG is fully constructed, and the CHC is satisfiable. On the other hand, if a query CHC can be successfully applied, a deduction of $\bot$ is found, and a counterexample is generated. In this case, the abstraction is refined, and the CEGAR loop continues.

Eldarica verifies C programs by converting them into CHCs [11]. The tool is also available in CoCoSim [6], an analysis and code generation framework for Simulink, a software widely used in the software development of embedded systems.

3.3 Transformation to Software Verification Problem

Transforming a problem to another domain opens up the possibility of using the new domain's algorithms and techniques to solve the original problem. This approach is used by Unihorn, a yet unpublished part of the Ultimate program analysis framework[2] that was submitted to CHC-COMP21. The tool converts the CHC problem into Boogie [1] program code, by which the problem of satisfiability is turned into a question of location reachability. The latter is then checked by another part of the framework, the software verification tool Ultimate Automizer [18]. The transformation is done with a *top-down* or *backward* approach, meaning verification starts at $\bot$, and exploration is done towards the facts in the CHC problem.

Unihorn creates a program with procedures representing the uninterpreted functions of the CHC problem. The body of each procedure contains the interpreted parts of CHCs that have the uninterpreted function corresponding to the procedure as their heads, in assert statements. After the assertions, procedure calls are made to the procedures representing the uninterpreted functions in the body of the CHC. The generated procedures return when their corresponding uninterpreted function is deducible. If all procedure calls return, an `assert false` statement at the entry point of the program denotes the unsafe property of the program, thereby the unsatisfiability of the CHC problem. An adaptation of the above transformation to Control Flow Automata instead of Boogie programs is described in detail in Section 4.3.

[2] https://monteverdi.informatik.uni-freiburg.de/tomcat/Website/

Chapter 4

Transformation of Constrained Horn Clauses to Control Flow Automata

In this chapter, two approaches of CHC to CFA transformations are described. The first *forward* transformation is my main contribution in this paper. The second *backward* transformation is my adaptation of a similar transformation used by Unihorn at CHC-COMP21 [13]. In the following, an overview of both transformations is presented, then each transformation is described in detail.

4.1 Overview of the 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 4.1.

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 4.1 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

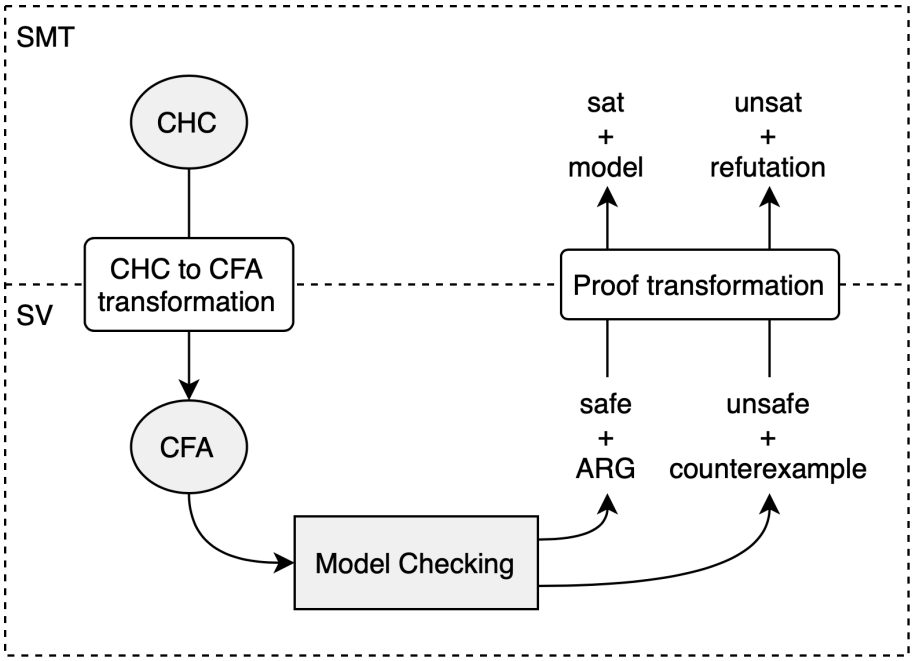

Figure 4.1: Overview of the presented work.

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 my main contribution, the *forward transformation* in Section 4.2 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. This is often referred to as the *backward* or *top-down* approach, and is used by Unihorn to transform CHCs into program code. The *backward transformation* in Section 4.3 is an adaptation of this, made to work with CFAs and CEGAR.

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

Example 5. *Consider the following CHC problem within integer arithmetic:*

$$A(n) \leftarrow n > 0 \land n < 100 \tag{4.1}$$
$$B(n, x) \leftarrow A(n) \land x > 0 \tag{4.2}$$
$$C(y, x) \leftarrow B(n, x) \land y = n - x \land y > 0 \tag{4.3}$$
$$A(n) \leftarrow C(y, x) \land n = y + (y \bmod x) \tag{4.4}$$
$$\bot \leftarrow A(n) \land n \geq 100 \tag{4.5}$$

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 the deduction approach from Example 3 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. As a matter of fact, z3, the prevalent SMT solver with dedicated CHC solving algorithm [15] can not solve this CHC.

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 mod 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.

4.2 Forward Transformation

The *forward transformation* creates a CFA from a linear CHC by mapping the uninterpreted functions directly to locations, and converting the CHCs to edges between these locations. A key property of this transformation is to have the reachability of a location with certain values correspond to the deducibility of the predicate with said values as parameters. If a predicate is deducible with some input parameters, then the location should be reachable with certain variables taking up the values of the input parameters. On the contrary, when a predicate is not deducible with some input parameters, then the location should be unreachable with certain variables taking up the values of the input parameters.

The CFA is created in a way that the verification of it resembles a *forward* or *bottom-up* approach: verification starts from the locations corresponding to the fact CHCs, and the question is whether a feasible path can be found to the CHC query's location, the *error location*. If it can be reached, a refutation can be generated from the path to it, thanks to the direct mapping between predicates and locations. On the other hand, if the error location can not be reached, the nodes of the built ARG can be used to define the predicates by mapping the explored states in each location to true in the corresponding uninterpreted function.

First, the transformation from CHCs to CFA is introduced, then the proof transformation is described.

4.2.1 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 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 $B_1(b_1^1, b_2^1, \ldots, b_{m_1}^1), B_2(b_1^2, b_2^2, \ldots, b_{m_2}^2), \ldots, B_n(b_1^n, b_2^n, \ldots, b_{m_n}^n)$ are mapped to the $CFA = (V, L, l_{Init}, E)$, where:

- $V = \{b_j^i \mid \forall i \in \{1, 2, \ldots, n\} : \forall j \in \{1, 2, \ldots, m_i\}\}$,
- $L = \{l_{Init}, l_{Err}, l_1, l_2, \ldots, l_n\}$,
- 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 6. *From Example 5, the first step of the forward transformation would create the* $CFA = (V, L, l_{Init}, \varnothing)$*, with the 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 CFA can be seen in Figure 4.2 as a graph of isolated nodes, which will be connected in later steps.*

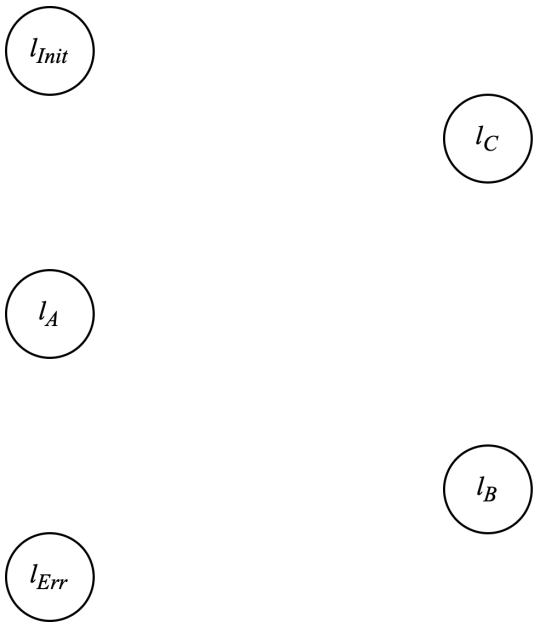

Figure 4.2: CFA after Step 1 of forward transformation.

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 sections. 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,
- $b^i_1 = x_1, b^i_2 = x_2, \ldots, b^i_{m_i} = x_{m_i}$, 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 $B_i(x_1, x_2, \ldots, x_{m_i})$ 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 $x_1, x_2, \ldots, x_{m_i}$ are assigned to $b^i_1, b^i_2, \ldots, b^i_{m_i}$, 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 $B_i(x_1, x_2, \ldots, x_{m_i})$. Thus, we can conclude that l_i is only reachable using the created edge with variables $b^i_1, b^i_2, \ldots, b^i_{m_i}$ valued $x_1, x_2, \ldots, x_{m_i}$, when $B_i(x_1, x_2, \ldots, x_{m_i})$ is deducible using C_l.

Example 7. *In Example 5, the second step of the forward transformation for fact CHCs would create the edge $e = (l_{Init}, op, l_A)$ from Equation 4.1, where the guard of op would be $n > 0 \wedge n < 100$, 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 CFA can be seen in Figure 4.3 as a graph, with the newly created edge and its label.*

Step 2/b. Create induction edges

For each induction CHC $C_l : B_i(x_1, x_2, \ldots, x_{m_i}) \leftarrow B_j(y_1, y_2, \ldots, y_{m_j}) \wedge \varphi_l$ where $i, j \in \{1, 2, \ldots, n\}$, 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:

- $y_1 = b^j_1, y_2 = b^j_2, \ldots, y_{m_j} = b^j_{m_j}$, 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,
- $b^i_1 = x_1, b^i_2 = x_2, \ldots, b^i_{m_i} = x_{m_i}$, assignment of the passed values to the variables corresponding to the input parameters of B_i.

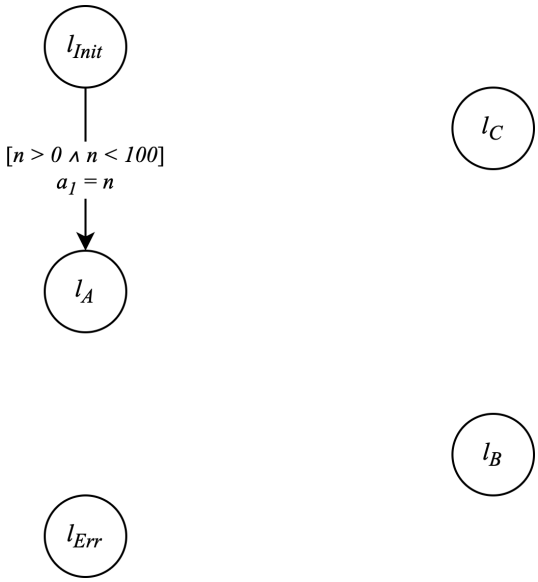

Figure 4.3: CFA after Step 2/a of forward transformation.

In addition to the first assignments, $x_1, x_2, \ldots, x_{m_i}$ 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.

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 $B_i(x_1, x_2, \ldots, x_{m_i})$ is only deducible, when both $B_j(y_1, y_2, \ldots, y_{m_j})$ is deducible, and φ_l is true. 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 $b^j_1, b^j_2, \ldots, b^j_{m_j}$ are assigned to $y_1, y_2, \ldots, y_{m_j}$ and the parameters $x_1, x_2, \ldots, x_{m_i}$ are assigned to $b^i_1, b^i_2, \ldots, b^i_{m_i}$, 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 $B_i(x_1, x_2, \ldots, x_{m_i})$ from $B_j(y_1, y_2, \ldots, y_{m_j})$. Thus, we can conclude that l_i is only reachable using the created edge with variables $b^i_1, b^i_2, \ldots, b^i_{m_i}$ valued $x_1, x_2, \ldots, x_{m_i}$ 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 $B_i(x_1, x_2, \ldots, x_{m_i})$ is deducible from $B_j(y_1, y_2, \ldots, y_{m_j})$ using C_l.

Example 8. *From Example 5, the second step of the forward transformation for induction CHCs would create three edges from Equation 4.2, 4.3 and 4.4:*

- $e_1 = (l_A, op_1, l_B)$ *for* $B(n, x) \leftarrow A(n) \wedge x > 0$*, 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* $C(y, x) \leftarrow B(n, x) \wedge y = n - x \wedge y > 0$*, where* op_2 *consists of the assignments* $n = b_1, x = b_2$*, then the guard* $y = n - x \wedge y > 0$*, and the assignments* $c_1 = y, c_2 = x$ *at last,*
- $e_3 = (l_C, op_3, l_A)$ *for* $A(n) \leftarrow C(y, x) \wedge n = y + (y \bmod x)$*, 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 resulting CFA can be seen in Figure 4.4 as a graph, with the newly created edges and their labels.

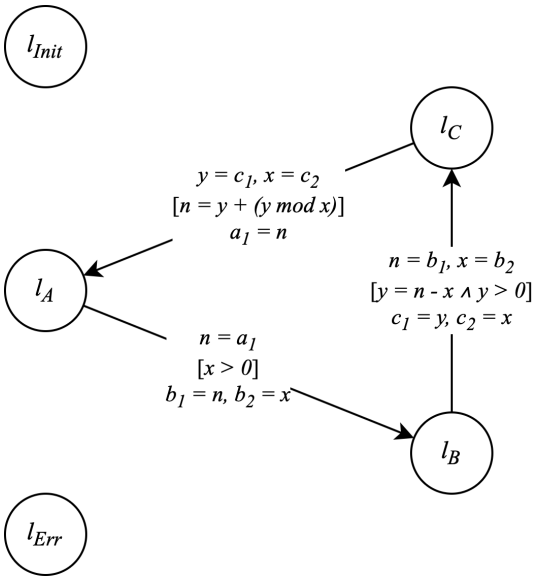

Figure 4.4: CFA after Step 2/b of forward transformation.

It is worth noting that the order of instructions is important: the assignments from the source location's variables need to happen before φ_l is evaluated.

Step 2/c. Create query edges

For each query CHC $C_l : \bot \leftarrow B_j(y_1, y_2, \ldots, y_{m_j}) \wedge \varphi_l$ 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:

- $y_1 = b_1^j, y_2 = b_2^j, \ldots, y_{m_j} = b_{m_j}^j$, 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 is 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 $b_1^j, b_2^j, \ldots, b_{m_j}^j$ 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 $b_1^j, b_2^j, \ldots, b_{m_j}^j$ 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 9. *In Example 5, the second step of the forward transformation for query CHCs would create the edge $e = (l_A, op, l_{Err})$ from Equation 4.5, where op would consist of the*

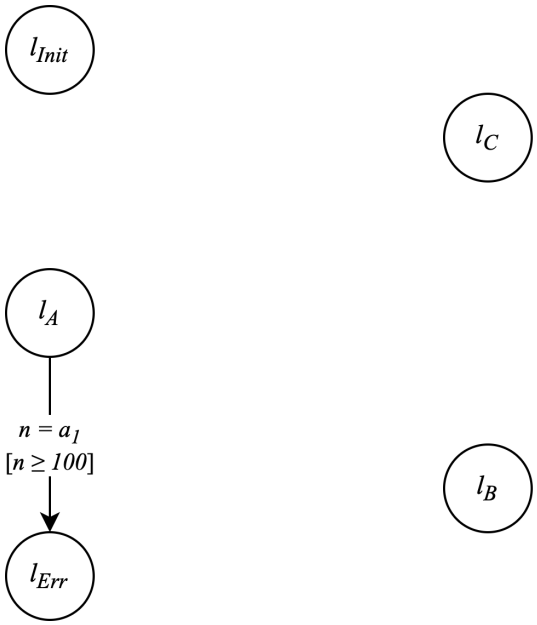

Figure 4.5: CFA after Step 2/c of forward transformation.

assignment $n = a_1$ and the guard $n \geq 100$. The CFA can be seen in Figure 4.5 as a graph, with the newly created edge and its label.

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. This can be expressed as $B_i \equiv false, \forall i \in \{1, 2, \ldots, n\}$, which is the same result as the one we got in Section 2.2.1, when discussing the same topic.

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. This can be expressed as $B_i \equiv true, \forall i \in \{1, 2, \ldots, n\}$, which is once again the same result as we got earlier.

Example 10. *The fully transformed version of the motivating Example 5 using forward transformation can be seen in Figure 4.6 as a graph.*

4.2.2 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

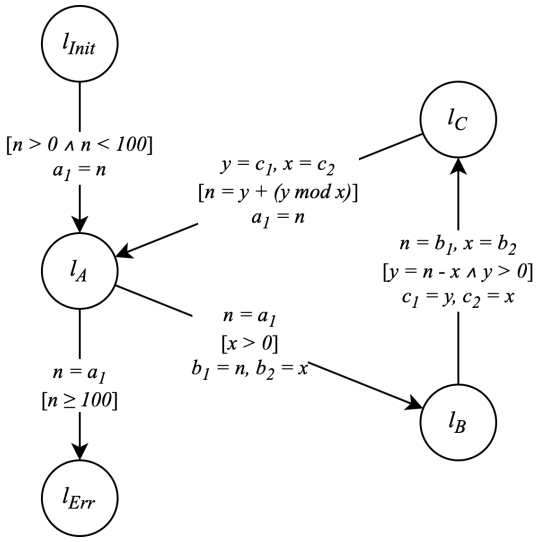

Figure 4.6: CFA of the motivating Example 5 after forward transformation.

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.

4.2.2.1 Satisfying Model Generation

When a SMT problem is satisfiable, a model can be found that satisfies it. In the case of a CHC problem, this means the definition of all uninterpreted functions $B_1(b_1^1, b_2^1, \ldots, b_{m_1}^1), B_2(b_1^2, b_2^2, \ldots, b_{m_2}^2), \ldots, B_n(b_1^n, b_2^n, \ldots, b_{m_n}^n)$ present in the set of CHCs, that satisfy all of the CHCs.

The transformation described in Section 4.2.1 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 $B_i, \forall i \in \{1, 2, \ldots, n\}$:

$$B_i(b_1^i, b_2^i, \ldots, b_{m_i}^i) = \bigvee_{S_j = (l_i, L_1^j, \ldots, L_{k_j}^j)}^{S^i} \left(L_1^j \wedge \cdots \wedge L_{k_j}^j \right) \tag{4.6}$$

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 4.6 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 vari-

ables $b_1^i, b_2^i, \ldots, b_{m_i}^i$ 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 ($\forall x$).

4.2.2.2 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 CFA with some values assigned to the variables and ends in the error location. The transformation described in Section 4.2.1 ensures that a location l_i in the CFA can only be reached with the related 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. 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 11. *Since the motivating Example 5 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 4.6, 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.

4.3 Backward Transformation

The *backward transformation* creates a CFA from a CHC by mapping the uninterpreted functions to procedures in the CFA, and converting the CHCs to procedure calls. In this approach, the reachability of a location in a procedure with certain values corresponds to the necessity of the deducibility of the predicate with said values as parameters, in order for the CHC problem to be *unsatisfiable*. The deducibility of a predicate with certain parameters is represented by the return value of a procedure call with said parameters, to the procedure resembling the predicate.

The creation of the CFA in this transformation is done with a *backward* or *top-down* approach: verification starts at the procedure corresponding to a query CHC, and the question is whether the procedure call to the procedure representing the predicate in the body of the query CHC, and all recursive procedure calls to procedures representing the predicates in the bodies of other CHCs, all return true by reaching a fact CHC branch of a procedure that has no further procedure calls. If the call to the procedure representing the predicate in the query CHC's body eventually returns, then an error location is reached and a refutation to the satisfiability of the CHC problem can be generated using the mapping between procedure calls and CHCs. On the other hand, if the initial procedure call never returns, the nodes of the built ARG can be used to define the predicates by mapping explored states in each procedure to false in the corresponding uninterpreted function.

The presented transformation is an adaptation of a transformation used in the Unihorn tool that transforms CHC problems into Boogie [1] code. My contribution in this section is the adaptation of the transformation used in Unihorn to CFAs.

4.3.1 Constrained Horn Clause Transformation

The transformation first creates the procedure CFAs, then maps the CHCs to edges.

Consider the linear CHC problem with CHC set $\{C_1, C_2, \ldots, C_k\}$ over 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. One constraint of this transformation is the limitation of the number of query CHCs to exactly 1. This is without loss of genericity, however, since queries do not depend on each other, meaning a transformation could be run for each query. If any of them came out to be unsatisfiable, then the problem would also be unsatisfiable; otherwise, it would be satisfiable.

Step 1. Create procedure CFAs

The uninterpreted functions $B_1(b_1^1, b_2^1, \ldots, b_{m_1}^1), B_2(b_1^2, b_2^2, \ldots, b_{m_2}^2), \ldots, B_n(b_1^n, b_2^n, \ldots, b_{m_n}^n)$ are mapped to a procedure $P_i(I_i, \{r_i\}, body_i)$ with CFA $CFA_i = (V_i, L_i, l_{Init}^i, E_i), \forall i \in \{1, 2, \ldots, n\}$, where:

- $I_i = \{b_1^i, b_2^i, \ldots, b_{m_i}^i\}$,
- $body_i$ is empty,
- $V_i = \{b_1^i, b_2^i, \ldots, b_{m_i}^i\}$,
- $L_i = \{l_{Init}^i, l_{Final}^i\}$,
- l_{Init}^i,
- $E_i = \varnothing$.

Semantically, a procedure and a CFA is created for each predicate, along with an initial location l_{Init} and a final location l_{Final}. In addition, a unique variable is created for each parameter in every predicate. It is worth noting that the edge set and the body is empty at this point because edges are added in a later step of the transformation.

Procedures are much closer to predicates in nature than locations. They have parameters, therefore the assignment of the passed parameters and return values will be handled automatically.

Step 2. Create main CFA

The single query in the CHC problem $\bot \leftarrow B_j(y_1, y_2, \ldots, y_{m_j}) \wedge \varphi_l$ with procedure $P_j = I_j, \{r_j\}, body_j$ corresponding to B_j is transferred into $CFA_{main} = (\varnothing, L, l_{Init}, E)$, where:

- $L = \{l_{Init}, l_{Err}, l_1\}$,
- $E = \{(l_{Init}, ([\varphi_l], call\, C), l_1), (l_1, [r_j], l_{Err})\}$
- $C = (P_j, \{y_1, y_2, \ldots, y_{m_j}\}, \{r_j\})$.

Semantically, a CFA is created with initial location l_{Init}, a distinguished error location l_{Err} and a middle location l_1. The edge going from l_{Init} to l_1 is guarded by φ_l, and calls the procedure P_j with its return value stored in r_j, representing the need for B_j to be deducible with the conditions φ_l. Another edge is created from l_1 to l_{Err} guarded by r_j, which encodes that if B_j can indeed be deduced with conditions φ_l, then the error location can be reached, meaning $\bot$ can be deduced.

Step 3. Create procedure CFA edges

In this step, each fact and query CHC is transformed into some edges and locations in the CFAs created in Step 1. The two kinds of CHC are treated differently, as described in the following.

In order to make the transformation sound, the aforementioned CHCs need to be transformed in a way that the procedures representing their heads can only return true with their bodies' constraints, if they can be deduced. This is achieved by creating disjunct paths from the initial location to the final location of the CFAs, with appropriate labels.

Step 3/a. Create induction edges

For each induction CHC $C_l : B_i(x_1, x_2, \ldots, x_{m_i}) \leftarrow B_j(y_1, y_2, \ldots, y_{m_j}) \wedge \varphi_l$ where $i \in \{1, 2, \ldots, n\}$, the following are added to $CFA_i = (V_i, L_i, l_{Init}^i, E_i)$ of the procedure $P_i(b_1^i, b_2^i, \ldots, b_{m_i}^i, r_i, body_i)$ to get $CFA_i' = (V_i', L_i', l_{Init}^i, E_i')$:

- $V_i' = V_i \cup \{r_l\}$,
- $L_i' = L_i \cup \{l_l\}$,
- $E_i' = E_i \cup \{(l^i_{Init}, ([\varphi_l], call\, C), l_l), (l_l, ([r_l], r_i = true), l^i_{Final})\}$,
- $C = (P_j, \{y_1, y_2, \ldots, y_{m_j}\}, \{r_l\})$.

This step is similar semantically to Step 2, with the second edge going to the final location l^i_{Final} and assigning the return value of the procedure r_i. As a result, the procedure can only return using the created edges if both $B_j(y_1, y_2, \ldots, y_{m_j})$ return true and φ_l is true, similarly to how $B_i(x_1, x_2, \ldots, x_{m_i})$ can only be deduced when $B_j(y_1, y_2, \ldots, y_{m_j})$ is deducible and φ_l evaluates to true. Thus, $B_i(x_1, x_2, \ldots, x_{m_i})$ can be deduced from $B_j(y_1, y_2, \ldots, y_{m_j})$ under the conditions φ_l if, and only if $P_i(x_1, x_2, \ldots, x_{m_i})$ returns true using the edges created above.

Step 3/b. Create fact edges

For each fact CHC $C_l : B_i(x_1, x_2, \ldots, x_{m_i}) \leftarrow \varphi_l$ where $i \in \{1, 2, \ldots, n\}$, the $CFA_i = (V_i, L_i, l^i_{Init}, E_i)$ of the procedure $P_i(b^i_1, b^i_2, \ldots, b^i_{m_i}, r_i, body_i)$ is extended with the edge $(l^i_{Init}, ([\varphi_l], r_i = true), l^i_{Final})$.

The head of a fact CHC can be deduced when φ_l is true. Similarly, the procedure P_i will only return true using the created edge when the guard φ_l evaluates to true. Consequently $B_i(x_1, x_2, \ldots, x_{m_i})$ can be deduced using C_l if, and only if $P_i(x_1, x_2, \ldots, x_{m_i})$ returns true using the edge created above.

To summarize, first, a procedure P_i and a CFA_i is created for each uninterpreted function B_i, then a CFA_{main} is created from the query, and finally, all CHCs are mapped to edges in the procedures' CFAs. The edges are created in a way that $P_i(x_1, x_2, \ldots, x_{m_i})$ returns true using the path created for a CHC if, and only if $B_i(x_1, x_2, \ldots, x_{m_i})$ can be deduced by said CHC, therefore we can conclude that the described transformation successfully converts the problem of satisfiability into a question of error reachability. Thus, using a model checker to decide the latter will yield a result for the former as well: if the CFAs are *unsafe*, the CHC problem is *unsatisfiable*; if the CFAs are *safe*, the CHC problem is *satisfiable*.

It is worth noting that in the case of linear CHCs, if any of the procedures reaches its final location, it will trigger a return chain all the way back to CFA_{main}, making the error location reachable. Consequently, the CFAs can only be safe if none of the procedures can return by going through a path corresponding to a fact.

Example 12. *From Example 5, the backward transformation creates the procedures* $A = (\{a_1\}, \{r_A\}, body_A)$, $B = (\{b_1, b_2\}, \{r_B\}, body_B)$ *and* $C = (\{c_1, c_2\}, \{r_C\}, body_C)$ *where the bodies can be seen as the CFAs' paths in Figure 4.7. Procedures calls are denoted with a more conventional notation for simplicity, where* $r_A = A(n)$ *represents call* (A, a_1, r_A).

The CHCs are represented by the following edges (noted with $\Leftrightarrow$*):*

$$A(n) \leftarrow n > 0 \wedge n < 100 \Leftrightarrow (l^A_{Init}, ([a_1 > 0 \wedge a_1 < 100], r_A = true), l^A_{Final})$$
$$B(n, x) \leftarrow A(n) \wedge x > 0 \Leftrightarrow (l^B_{Init}, ([b_2 > 0], r_A = A(b_1)), l_B)$$
$$C(y, x) \leftarrow B(n, x) \wedge y = n - x \wedge y > 0 \Leftrightarrow (l^C_{Init}, ([c_1 = n - c_2 \wedge c_1 > 0], r_B = B(n, c_1)), l_C)$$
$$A(n) \leftarrow C(y, x) \wedge n = y + (y \bmod x) \Leftrightarrow (l^A_{Init}, ([a_1 = y + (y \bmod x)], r_C = C(y, x)), l_A)$$
$$\bot \leftarrow A(n) \wedge n \geq 100 \Leftrightarrow (l_{Init}, ([n \geq 100], r_A = A(n)), l_1)$$

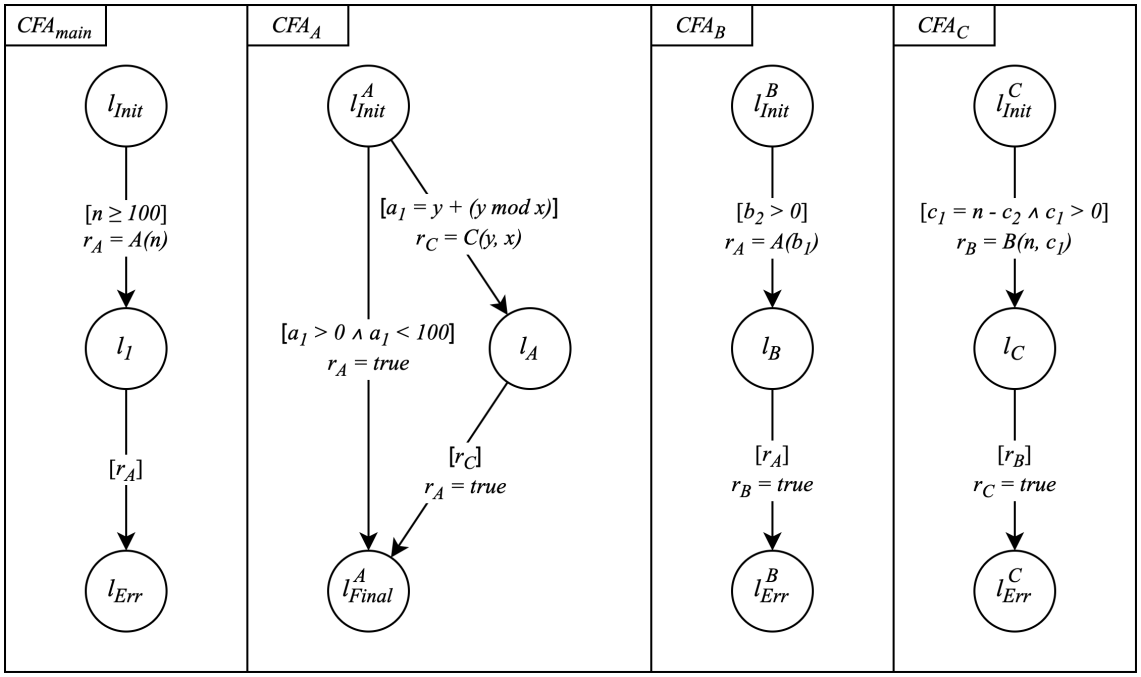

Figure 4.7: CFAs of the motivating Example 5 after backward transformation.

One advantage of this approach over forward transformation is that it also works for non-linear CHCs, because additional calls and guards can be added to the already existing ones. A great disadvantage of it, however, is that it can easily get stuck in infinite recursive calls during verification if a recursive induction exists that does not narrow down the possible values of the parameters.

4.3.2 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.

4.3.2.1 Satisfying Model Generation

When a SMT problem is satisfiable, a model can be found that satisfies it. In the case of a CHC problem, this means the definition of all uninterpreted functions $B_1(b^1_1, b^1_2, \ldots, b^1_{m_1}), B_2(b^2_1, b^2_2, \ldots, b^2_{m_2}), \ldots, B_n(b^n_1, b^n_2, \ldots, b^n_{m_n})$ present in the set of CHCs, that satisfy all of the CHCs.

The transformation described in Section 4.3.1 ensures that a procedure call $P_i(x_1, x_2, \ldots, x_{m_i})$ returns true if, and only if $B_i(x_1, x_2, \ldots, x_{m_i})$ can be deduced, which in the case of a safe program is not the case for any of the procedures. Given a location l_i in the CFA of P_i, 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 $L_1^j \wedge \cdots \wedge L_{k_j}^j$, and therefore the corresponding predicate B_i should not return true under the conditions $L_1^j \wedge \cdots \wedge L_{k_j}^j$. This is true for all $S^i = \{S_j \mid S_j = (l_i, L_1^j, \ldots, L_{k_j}^j), l_i \in L_i, CFA_i = (V_i, L_i, l_{Init}^i, E_i)\}$ nodes in the ARG; thus, B_i needs to evaluate to false under either of their conditions:

$$B_i(b_1^i, b_2^i, \ldots, b_{m_i}^i) = \neg \bigvee_{S_j = (l_i, L_1^j, \ldots, L_{k_j}^j)}^{S^i} \left(L_1^j \wedge \cdots \wedge L_{k_j}^j \right) \tag{4.7}$$

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, none of the procedures could return in any of the abstract states. Therefore the definitions provided by Equation 4.7 guarantee that there can not be a deduction to $\bot$, meaning they satisfy the CHC problem.

4.3.2.2 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.

Refutations can be created similarly to Section 4.2.2.2: the one-on-one mapping between CHCs and paths in the procedure CFAs can be used to convert the counterexample provided by the model checker into a series of CHC applications.

Chapter 5

Evaluation

In this chapter, a prototype implementation of the transformations described in Chapter 4 are evaluated on a set of CHC problems. First, the benchmark environment is described, then the benchmark results are presented. Finally, possible improvements and future plans are discussed.

5.1 Benchmark Setup

A prototype of the CHC to CFA transformations was implemented in the Theta [17], an open-source formal model checking framework. The transformations are done in a frontend using ANTLR[1]. The frontend reads the CHC in the format used in CHC-COMP[2]. The frontend can be turned on with the `--chc` flag, and can be configured to use different transformations by the `--chc-transformation FORWARD/BACKWARD` flag, for forward and backward transformations, respectively. In its current state, the implementation can only give an answer of *satisfiable* or *unsatisfiable* to a CHC problem; the generation of refutations and proofs is not yet implemented. Thereby only the correctness of the answer and the time it took to compute was benchmarked.

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[3]. The benchmarks used are a careful selection of distributed benchmark repositories, as described in the CHC-COMP21 competition report. [13] The CHCs originate from software verification problems and function synthesis tasks, among others.

Theta is a highly configurable framework, therefore many different configurations were tested. All sensible combinations of the following options were tested:

- CHC transformation: `FORWARD`, `BACKWARD`
- domain: `EXPL`, `PRED_CART`, `PRED_BOOL`, `PRED_SPLIT`
- interpolation: `SEQ_ITP`, `BW_BIN_ITP`, `NWT_IT_WP`, `NWT_WP_LV`
- predicate split: `ATOMS`, `WHOLE`
- solver: `z3` [9], `mathsat` [7] [7], `cvc4`

[1] https://www.antlr.org
[2] https://chc-comp.github.io
[3] https://github.com/chc-comp/chc-comp21-benchmarks

The `--chc`, `--init-prec empty` and `--max-enum 1` flags were used in all tests. The actual configurations tested are presented along with the results in Section 5.2. For comparison, the top solvers from CHC-COMP21 on the LIN-Lia track were benchmarked as well in the same environment, using their default configurations.

The execution of benchmarks was done using the BenchExec framework. [3] The tests were run on virtual machines equipped with 8 CPU cores and 16 GB of memory, in the BME-NIIF cloud[4]. A timeout of 300 seconds was chosen for each benchmark, in order allow for the wide variety of configurations to be tested within limited time constraints.

5.2 Benchmark Results

The tested configurations only gave correct answers, meaning they either answered correctly or did not answer within the 5-minute timeout. Therefore, the results are presented from a perspective of performance. More specifically, the number of solved tasks within the 5-minute time constraint is shown, along with a quantile plot of the solved tasks with respect to time passed. The latter chart shows how the number of solved tasks would change (x axis), had the time limit been set to a certain value (y axis).

5.2.1 Theta Configurations

First, we tested sensible configurations of Theta, as described in Section 5.1, with the solver being set to z3 in all cases. The number of solved tasks by each configuration can be seen in Table 5.1.

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 5.1: Number of solved tasks by configuration.

Forward transformation proved to be more effective by far 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 best overall configuration turned out to be `PRED_SPLIT` & `BW_BIN_ITP` & `WHOLE` with `FORWARD` transformation.

5.2.2 Different Underlying Solvers

Under the hood, Theta uses SMT solvers to decide the feasibility of paths in the CFA. To save on time, only the best configurations of Theta based on our experiments in

[4]https://niif.cloud.bme.hu

Section 5.2.1 were chosen and tried out with different underlying solvers: `z3`, `mathsat` and `cvc4`. The number of solved tasks by THETA using each solver configuration can be seen on Table 5.2.

domain	pred-split	z3	mathsat	cvc4
PRED_CART	ATOMS	301	239	223
PRED_CART	WHOLE	302	236	224
PRED_SPLIT	ATOMS	291	238	225
PRED_SPLIT	WHOLE	328	254	246

Table 5.2: Number of solved tasks by different solvers.

THETA performed significantly better when it used `z3` as its solver. It is important to note, however, that the integration of `mathsat` and `cvc4` into THETA has not been without problems due to problems with the SMT-LIB interface [10].

5.2.3 Comparison to Other Tools

Benchmarks were also run with the top solvers from CHC-COMP21 on the LIA-Lin track, namely z3, Unihorn and Eldarica. These solvers were run using their default configuration, with the same hardware, time constraints and benchmark framework as THETA. The number of solved tasks compared to the best-performing configuration of THETA can be seen on Table 5.3.

Theta	z3	Eldarica	Unihorn
328	437	337	380

Table 5.3: Number of solved tasks compared to other tools.

Though the best configuration of THETA performs worse than the other solvers, its performance is in the ballpark of the third-best solver, Eldarica. On top of that, a smart portfolio technique that chooses the best configuration of THETA for each task would result in 388 solved tasks, which would put THETA in second place based on these results.

A quantile plot of the tools' performances can be seen on Figure 5.1. The time it took to solve tasks can be seen on the y axis with a *logarithmic scale*, and the number of tasks that were solved within said time is present on the x axis. 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.

5.2.4 Threats to Validity

In this section, possible biases and threats to the validity of the benchmark results are discussed. First, the correctness of the implemented transformation is further elaborated upon, then, the performance results are examined.

The CHC-COMP21 repository does not provide expected answers to the CHC problems. This meant that the only way to evaluate the soundness of the implementation was to generate the results ourselves, using state-of-the-art SMT solvers with some time limit. We could obtain the expected results of 456 CHCs out of the 585 that we benchmarked with, with 129 remaining unknown. Thereby there is a possibility that the answers given by THETA to the tasks without an expected answer were wrong. There were only 15 of

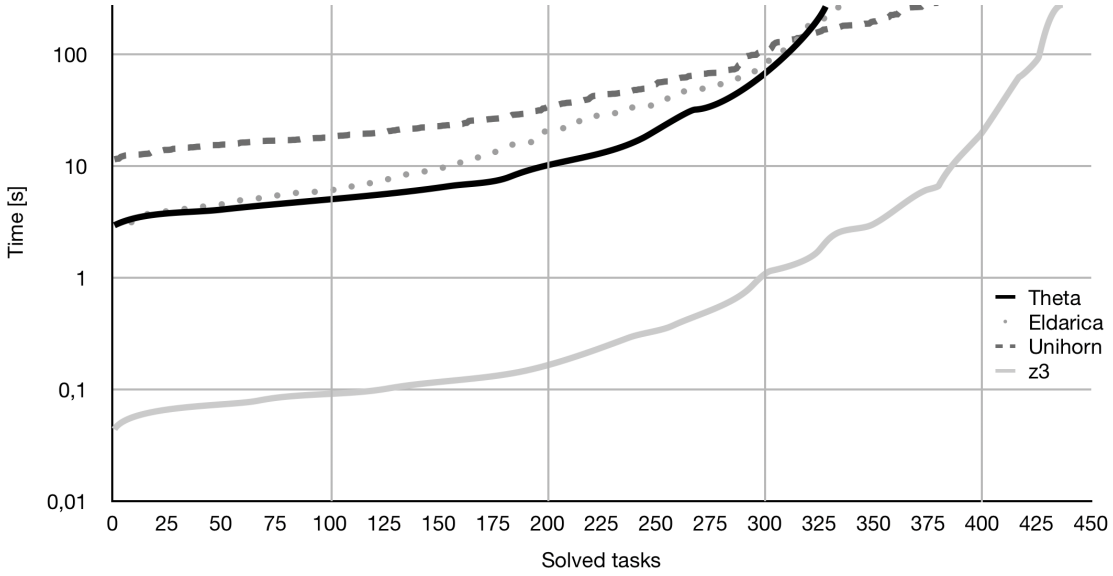

Figure 5.1: Number of solved tasks by tools under a certain time.

the 129 such tasks, though, and the remaining 373 tasks with known expected answers were all correct. Therefore we can be confident that the implemented transformation is correct.

To get benchmark results in reasonable time, the aforementioned 5-minute limit had to be introduced. Given enough time, the number of solved tasks could have turned out differently. However, the satisfiability of a CHC problem is not decidable in general, therefore a limit always has to be used in practice. 5 minutes still may seem a bit too short, as the quantile plot in Figure 5.1 shows that a 2-minute timeout would have resulted in a completely different ranking of the benchmarked tools. This is the reason why the quantile plot is presented in the first place, to provide more insight into how the tools behave. In addition to that, counting solved tasks can be seen as more of a measure of practical performance.

Another thing to note about the comparison to other tools is that the others were used with their default configurations, whereas the best out of many configurations of Theta was used. The main reason for this is that our goal was not to compete with other tools; it was just to get an approximate idea of where this technique of CHC solving lies within the grand scheme of things. On top of that, the best configurations of the other tools were either outdated or just difficult to obtain.

Chapter 6

Conclusion

As software enters more and more parts of our lives, the complexity of software components increases. This is true for safety-critical embedded systems, where the correctness of the embedded software must be ensured. Conventional testing can no longer be exhaustive, leaving the need for other ways to prove correctness. Formal software verification aims to provide mathematical proof of the correctness of programs. A commonly used approach by efficient software verification tools is to convert programs and their correctness into mathematical formulae, the satisfiability of which can be checked. This problem is called a *Satisfiability Modulo Theory (SMT)* problem.

Constrained Horn Clauses (CHC) are logical implications between uninterpreted functions that capture deduction problems well when utilized in an SMT problem. Consequently, they are used in Constrained Logic Programming, but also in embedded systems: in distributed knowledge databases and memory representation. Most importantly, it is used in software verification of programs written in low-level languages: numerous effective software verification tools convert programs written in C or Rust into CHCs, then solve the satisfiability problem of the CHCs to decide the correctness of a program.

Solving the satisfiability of CHCs is not trivial due to the large state space of possible deductions and the possibility of infinite deduction. In Chapter 4, I presented an approach that transforms the satisfiability problem of CHCs into a question of reachability in programs, which allows access to powerful abstraction-refinement based model checking algorithms. This was achieved by transforming the CHCs into *Control Flow Automata (CFA)*, a formal representation of programs. Two different transformations were described: *forward transformation* in Section 4.2, and *backward transformation* in Section 4.3. My contribution was the novel forward transformation, which converted linear CHCs to CFAs with a *bottom-up* approach, starting from the fact CHCs towards the query CHCs. Backward transformation was my adaptation to CFAs of an existing transformation used by Unihorn, which converted CHCs to programs with a *top-down* approach, from the query CHCs towards the fact CHCs. The backward transformation also supports non-linear CHCs, but it is also less likely to terminate for difficult problems.

Prototypes of both transformations were implemented in the open-source model checking framework Theta. The implementation was evaluated on a set of 585 linear CHCs, in different configurations of Theta. Both transformation gave correct answers only, meaning they either gave the correct answer or no answer within the 5-minute timeout. Forward transformation performed significantly better than backward transformation, getting comparable performance results to the top CHC solvers in linear integer arithmetic problems.

6.1 Future Work

In the short term, the prototype implementation could be extended to support Theta's pre-verification optimization techniques, which could result in significant improvement gains in efficiency. Our goal is to submit Theta as a promising competitor for next year's CHC-COMP23.

A longer-term goal is to implement refutation and proof generation. The theory behind it is all laid out in Chapter 4, but integration with all supported domains of Theta will take further investigation.

As for more theory related-tasks, verification using the backward transformation does not always terminate for difficult tasks due to the possibility of infinite recursion. One approach to handling this would be some kind of similarity detection between location stacks in the abstract state space, which could additionally prove to be useful in software verification.

Bibliography

[1] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, pages 364–387, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-36750-5. DOI: 10.1007/11804192_17.

[2] Dirk Beyer and M. Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 184–190, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-22110-1.

[3] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer*, 21 (1):1–29, Feb 2019. ISSN 1433-2787. DOI: 10.1007/s10009-017-0469-y. URL https://doi.org/10.1007/s10009-017-0469-y.

[4] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, NLD, 2009. ISBN 1586039296. DOI: 10.5555/1550723.

[5] Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. *Horn Clause Solvers for Program Verification*, pages 24–51. Springer International Publishing, Cham, 2015. ISBN 978-3-319-23534-9. DOI: 10.1007/978-3-319-23534-9_2. URL https://doi.org/10.1007/978-3-319-23534-9_2.

[6] Hamza Bourbouh, Pierre-Loïc Garoche, Thomas Loquen, Éric Noulard, and Claire Pagetti. Cocosim, a code generation framework for control/command applications an overview of cocosim for multi-periodic discrete simulink models. In *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*, Toulouse, France, January 2020. URL https://hal.archives-ouvertes.fr/hal-02441334.

[7] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. The mathsat5 smt solver. In Nir Piterman and Scott Smolka, editors, *Proceedings of TACAS*, volume 7795 of *LNCS*. Springer, 2013.

[8] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, sep 2003. ISSN 0004-5411. DOI: 10.1145/876638.876643. URL https://doi.org/10.1145/876638.876643.

[9] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction*

and Analysis of Systems, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24.

[10] Mihály Dobos-Kovács. On the verification of safety-critical embedded software systems. Master's thesis, Budapest University of Technology and Economics, 2021. URL https://diplomaterv.vik.bme.hu/en/Theses/Kritikus-beagyazott-szoftverek-verifikacios.

[11] Zafer Esen. Extension of the eldarica c model checker with heap memory. Master's thesis, Uppsala University, Department of Information Technology, 2019.

[12] Zafer Esen and Philipp Ruemmer. Tricera: Verifying c programs using the theory of heaps. In *CONFERENCE ON FORMAL METHODS IN COMPUTER-AIDED DESIGN–FMCAD 2022*, pages 360–391. TU Wien Academic Press, 2022. DOI: 10.34727/2022/isbn.978-3-85448-053-2_45.

[13] Grigory Fedyukovich and Philipp Rümmer. Competition report: CHC-COMP-21. In Hossein Hojjat and Bishoksan Kafle, editors, *Proceedings 8th Workshop on Horn Clauses for Verification and Synthesis, HCVS@ETAPS 2021, Virtual, 28th March 2021*, volume 344 of *EPTCS*, pages 91–108, 2021. DOI: 10.4204/EPTCS.344.7. URL https://doi.org/10.4204/EPTCS.344.7.

[14] Yeting Ge. *Solving Quantified First Order Formulas in Satisfiability Modulo Theories*. PhD thesis, New York University, 2010.

[15] Arie Gurfinkel. Program verification with constrained horn clauses (invited paper). In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification*, pages 19–29, Cham, 2022. Springer International Publishing. ISBN 978-3-031-13185-1. DOI: 10.1007/978-3-031-13185-1_2.

[16] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The seahorn verification framework. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 343–361, Cham, 2015. Springer International Publishing. ISBN 978-3-319-21690-4. DOI: 10.1007/978-3-319-21690-4_20.

[17] Ákos Hajdu and Zoltán Micskei. Efficient strategies for cegar-based model checking. *Journal of Automated Reasoning*, 64(6):1051–1091, Aug 2020. ISSN 1573-0670. DOI: 10.1007/s10817-019-09535-x. URL https://doi.org/10.1007/s10817-019-09535-x.

[18] Matthias Heizmann, Yu-Fang Chen, Daniel Dietsch, Marius Greitschus, Jochen Hoenicke, Yong Li, Alexander Nutz, Betim Musa, Christian Schilling, Tanja Schindler, and Andreas Podelski. Ultimate automizer and the search for perfect interpolants. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 447–451, Cham, 2018. Springer International Publishing. ISBN 978-3-319-89963-3. DOI: 10.1007/978-3-319-89963-3_30.

[19] Hossein Hojjat and Philipp Rümmer. The eldarica horn solver. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–7, 2018. DOI: 10.23919/FMCAD.2018.8603013.

[20] Hari Govind V K, Sharon Shoham, and Arie Gurfinkel. Solving constrained horn clauses modulo algebraic data types and recursive functions. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022. DOI: 10.1145/3498722. URL https://doi.org/10.1145/3498722.

[21] Maurizio Lenzerini. Data integration: A theoretical perspective. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 233–246, 01 2002. DOI: 10.1145/543613.543644.

[22] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. Rusthorn: Chc-based verification for rust programs. *ACM Trans. Program. Lang. Syst.*, 43(4), oct 2021. ISSN 0164-0925. DOI: 10.1145/3462205. URL https://doi.org/10.1145/3462205.

[23] Philipp Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 274–289, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-89439-1. DOI: 10.1007/978-3-540-89439-1_20.

[24] Victor Vianu Serge Abiteboul, Richard Hull. *Foundations of Databases*. Addison-Wesley, 1995. ISBN 0-201-53771-0. URL http://webdam.inria.fr/Alice/.

[25] Letizia Tanca Stefano Ceri, Georg Gottlob. *Logic Programming and Databases*. Springer Berlin, Heidelberg, 1990. ISBN 978-3-642-83954-2. DOI: https://doi.org/10.1007/978-3-642-83952-8.

[26] Tamás Tegzes. Learning and synthesis supported software verification. Master's thesis, Budapest University of Technology and Economics, 2020. URL https://diplomaterv.vik.bme.hu/en/Theses/Tanulo-es-Szintezis-Algoritmusokkal-Tamogatott1.