S C I E N C E ▪ P A S S I O N ▪ T E C H N O L O G Y

Mario Comanici

A Tool for Verifying Programs using Constrained Horn Clauses

BACHELOR'S THESIS

Bachelor's degree programme: Software Engineering and Management

Supervisor

Benedikt Maderbacher

Institute of Applied Information Processing and Communications
Graz University of Technology

Graz, 05 2022

Abstract

The problem of program verification is nothing new in the field of computer science. Over many years there have been approaches and solutions on how to do so. One of those technologies are so-called Constrained Horn Clauses (CHCs), used in this field to specify and verify programs. Many companies, including Microsoft, have developed specific tools to prove a program correct. Microsoft does so with its tool Z3 [1] that for example is suitable to check Constrained Horn Clauses. However these clauses have to be generated first and in this thesis we are going to have a look at the underlying theories on Constrained Horn Clauses to get a basic understanding on the topic including prerequieries on Constrained Horn Clauses, Hoare logic and an algorithm based on weakest liberal preconditions to then further look at the implementation of a tool that specifies Constrained Horn Clauses on a given program.

Keywords: Constrained Horn Clauses, Horn Clauses, Weakest Liberal Precondition, Spacer, Program Verification

Contents

List of Figures

1 Introduction

Program verification has been a field of computer science for many years. One of the theories, as already mentioned, is the verification using Constrained Horn Clauses. Constrained Horn Clauses are part of formal methods underlying the idea of Horn Clauses that have been introduced by Alfred Horn in the year 1951 [2]. Those Horn Clauses can then be expanded to Constrained Horn Clauses. Constrained Horn Clauses in general are a powerful method to be used to verify programs and their validity and thus assuring wanted outcomes of programs as they have proven themselves in this field over the past years.

Also located in the field of formal methods is the idea of Hoare logic introduced by Tony Hoare [3] and also by Robert Floyd [4] in the 1960s, that propose the splitting of programs in the program itself as well as a precondition and a postcondition to hold before and after execution as we will see in some of the following chapters. The understanding of both theories mentioned allows a combination of them to finally find appliance in program verification as it will be used in this thesis.

With that, we want to build a tool that can take a program and translate it into Constrained Horn Clauses. The idea behind the necessity of a tool is based on the fact that proving larger programs by hand will eventually become very expensive regarding time and also very prone to errors. To do so there will be an introduction on an algorithm based on the theory on weakest liberal preconditions taken from the paper by Nikolaj Bjørner et al. [5].

There are many other ways to translate code into Constrained Horn Clauses, but eventually they all serve the purpose to further use them and verify their satisfiability and thus proving a given program to execute right. Companies like Microsoft or the Seahorn Framework do so with their implementation of the tools Z3 [1] and Spacer [6].

The final outcome of this thesis will be the understanding of the underlying theories and the appliance of those in a tool for automatically proving programs. We implemented the algorithm presented by Nikolaj Bjørner et al. [5] using Scala [7] as programming language to generate Constrained Horn Clauses and verified those clauses in the implementation using Microsoft's Z3 Python library [1]. The evaluation of the right behaviour was done by using our own examples on the implementation.

We will have a look at the given program following in the next figure 1.1 throughout this work as practical reference along the theoretical parts to support the process of understanding and ultimately answer the question if this program does what it should. This program consists of a single function called *sum_upto* that takes the parameter x as variable to then calculate the sum of x inside of the while loop by summing it up and decreasing it in each step. For example the number 5 would be sum up to 5 + 4 + 3 + 2 + 1 leading to the result of 15. The *assume* and *assert* statement are already part of one verification step we will introduce in this thesis called pre- and postcondition. In this case the precondition in line two (*assume* $(x > 0)$) states that the input parameter is assumed to be greater than zero assuring that the number to sum up is not a negative number whereas the postcondition in line eight (*assert* $(r \geq x)$) assures that the return value r is at least greater or equal to x after execution.

```
1  def sum_upto(x) {
2    assume (x ≥ 0)
3    r := 0
4    while (x > 0) {
5      r := r + x
6      x := x - 1
7    }
8    assert (r ≥ x)
9  }
```

Figure 1.1: A small Example Program [8]

Outline. The structure of this paper is as follows: in chapter 2, we introduce the relevant background on Hoare logic that is necessary for the understanding of CHCs. Then we move on to basic background information on Horn Clauses in chapter 3 to then further deepen the knowledge by introducing Constrained Horn Clauses. After having covered the theoretical part we will get a look on the algorithm to see how program code is translated into CHCs using weakest liberal preconditions. Then we have a look on Spacer [6] to get an idea of how the satisfaction of those clauses is solved by already existing tools in this area. With that in chapter 6 we can finally have a look on the implementation of the tool that takes a given program to then translate it into CHCs using the underlying theories and the algorithm introduced before in this work. Finally, in chapter 8 we conclude our findings and discuss potential future work.

2 Hoare Logic

The following chapter will be about the Hoare logic, more specifically about Hoare triples. The understanding of Hoare logic is a necessity to further understand how we can prove and verify the functionality of code with Constrained Horn Clauses that are discussed in chapter 3.

2.1 Background

As already mentioned, Hoare logic is a set of axioms with logical rules used to reason about the correct execution of programs and code. They were first introduced by Tony Hoare in 1969 [3]. The background on Hoares theory was based on earlier work from Robert Floyd who developed a similar theory two years earlier in 1967 [4].

2.2 Hoare Triple

The main element of Hoare logic is the so-called Hoare triple consisting of three parts as the name already indicates.

$$\{P\}S\{Q\}^1$$

Figure 2.1: Hoare Triple

As seen in figure 2.1, the first element of the triple called P is the precondition of the program. In the middle stands S that is the program itself. Then comes Q as the postcondition after the execution of the program. This notation states that if a program S is executed in a way such that its state is satisfying the precondition P then the state of S also satisfies the postcondition Q after S terminates. In this case one would say that the triple holds true for its specific program. If at the beginning we do not have a precondition the according Hoare triple can be written as $\{true\}S\{Q\}$.

However, the Hoare triple as specification only provides partial correctness because it is not necessary given that the program S terminates for the triple to hold true just because its precondition P is valid. For the triple to show total correctness we also need the program S to terminate with the precondition P as valid and the postcondition Q to hold afterwards. If the program does not terminate, the postcondition can be an arbitrary term. In practice, the verification of partial correctness and termination are looked at separately [3] [8] [9].

[1]Hoare used a different notation of the Hoare triples that looked like $P\{Q\}R$

Taking a look at figure 1.1, we first build the Hoare triple corresponding to the specified code in retrospect of calling the function with *sum := sum_upto (x)*. The variable *sum* will have a value that is greater or equal zero if x is greater or equal to zero. If x is less than zero then *sum* will have the value zero. In this case the precondition P is $\{x \geq 0\}$ seen as **assume** ($x \geq 0$) in the code, the program code S is *sum := sum_upto(x)* and the postcondition Q is $\{sum \geq x\}$ seen as **assert** ($r \geq x$) in the code as validation for the return value r that is then assigned to *sum*. All together written as:

$$\{x \geq 0\}\ sum := sum\_upto(x)\ \{sum \geq x\}$$

2.3 Axiom and Rules

The following axiom and rules will be applied further in this work as we then take a look on an algorithm to translate code into Constrained Horn Clauses in chapter 4. The axiom and the rules are as presented and discussed by Mike Gordon [9].

2.3.1 Axiom of Assignment

As mentioned by Hoare [3], the assignment is an essential feature of computer programming. The assignment in logical programming for a variable V being assigned the value E would look as following $V := E$. Now, if this so-called assertion (P) on the left side (precondition) is assumed to be true, the assertion on the right side (postcondition) is true after the assignment. In this case, the axiom of assignment states that we substitute all occurrences of E with V. This can be expressed as:

$$\frac{}{\{P[E/V]\}V := E\{P\}}$$

Figure 2.2: Axiom of Assignment

Per definition, the axiom of assignment is not really an axiom but a schema [3]. The representation in figure 2.2 is different[2] then Hoare himself denoted it in his work. We will however use this representation as it is the natural deduction style used in predicate logic and the representation commonly used nowadays.

2.3.2 Rule of Consequence

We assume that the precondition P_1 implies the precondition P_2. Then the program S is executed with the precondition P_2 and afterwards the postcondition Q_2 holds true. We then assume that the postcondition Q_2 implies the postcondition Q_1 and thus we can say that the triple $\{P_1\}$S$\{Q_1\}$ also holds true. More formally seen in the form of Hoare

[2]Original notation by Hoare: $\vdash P_0$ {x := f} P

logic in figure 2.3[4]. This rule is the combination[3] of so-called precondition strengthening and postcondition weakening [9].

$$\frac{P_1 \to P_2 \text{ , } \{P_2\}\text{S}\{Q_2\} \text{ , } Q_2 \to Q_1}{\{P_1\}\text{S}\{Q_1\}}$$

Figure 2.3: Rule of Consequence

2.3.3 Rule of Composition

This rule[5] simply states that if the result Q, that in this case is our postcondition of the first part of a program S, is the same as the precondition for the second part of the program T that will give the result R, then the whole program will give the result R.

$$\frac{\{\text{P}\}\text{S}\{\text{Q}\} \text{ , } \{\text{Q}\}\text{T}\{\text{R}\}}{\{\text{P}\}\text{S;T}\{\text{R}\}}$$

Figure 2.4: Rule of Composition

2.3.4 Rule of Iteration

While B holds true the loop S is executed. The initial precondition P holds true before and afterwards, while after the loop B is false[6].

$$\frac{\{\text{P} \land \text{B}\}\text{ S }\{\text{P}\}}{\{P\}\text{ WHILE B DO S }\{P \land \neg B\}}$$

Figure 2.5: Rule of Iteration

The assertion P in this case represents a loop invariant. Invariants in general are statements that hold true before and after each iteration of the loop and do not change as the name indicates. This also means that a loop invariant is both a precondition and a postcondition for the loop. [10].

[3]Gordon [9] split the rule of consequence into the two mentioned. We use the rule of consequence as it is seen in figure 2.3

[4]Original notation by Hoare: If $\vdash$P {Q} R and $\vdash$R $\supset$ S then $\vdash$P {Q} S; If $\vdash$P {Q} R and $\vdash$S $\supset$ P then $\vdash$S {Q} R

[5]Original notation by Hoare: $\vdash$P $\{Q_1\}$ R_1 and $\vdash$ R_1 $\{Q_2\}$ R then $\vdash$P $\{(Q_1; Q_2)\}$ R

[6]Original notation by Hoare: If $\vdash$P $\land$ B $\{S\}$ P then $\vdash$P {while B do S} $\neg$ B $\land$ P

2.3.5 Rule of Condition

The rule of condition states that if the precondition P AND the condition B hold true, the program S is executed. Otherwise if the precondition P AND the negation of the condition B hold true, the program T is executed and both paths lead to the postcondition Q.

$$\frac{\{P \wedge B\}S\{Q\}, \{P \wedge \neg B\}T\{Q\}}{\{P\} \text{ IF } B \text{ THEN } S \text{ ELSE } T \ \{Q\}}$$

Figure 2.6: Rule of Condition

3 Constrained Horn Clauses

The following chapter will be about Horn Clauses and Constrained Horn Clauses. To get an understanding of Constrained Horn Clauses that will be discussed later on in this chapter there is a need to understand the basic idea and terminology on Horn Clauses first.

Horn clauses are named after the American mathematician Alfred Horn who first pointed out their importance in 1951 [2].

3.1 What Horn Clauses are used for

We will first dig into the theory of Horn Clauses to gain a basic understanding of the structure they provide. This theory then is extended from Horn Clauses to Constrained Horn Clauses. Horn Clauses find their usage in many scientific fields like for example, theories in logic programming. In our case we will look at the use case of Horn Clauses for automatic program verification or equivalently symbolic model checking [5] [11] [8].

3.2 Basic Terminology

Lets first begin with the basic terminology regarding Horn Clauses. Looking at a logic term, for example $A \rightarrow B$. This is equivalent to $\neg\ A \vee B$. Furthermore, this means that

$$A_1 \wedge .. \wedge A_k \rightarrow B$$

is equivalent to

$$\neg A_1 \vee .. \vee \neg A_k \vee B$$

Logical expressions in the form of $B_1 \vee\ ..\ \vee B_k$ are called clauses.

Atomic formulas or their negations are so-called literals. An example for a positive literal would be the atom x and the negation of that literal would be $\neg\ x$. If the clause has a maximum of one positive literal, the clause is called a Horn Clause for example, written as

$$x_1 \wedge x_2 \wedge ... \wedge x_n \rightarrow q$$

which, as already mentioned, would be equivalent to

$$\neg x_1 \vee \neg x_2 \vee ... \vee \neg x_n \vee q$$

with q being the only positive literal here [12]. So-called definite clauses or strict clauses are Horn Clauses that consist of only one positive literal and at least one negative literal.

For example, the clause we have seen before would be such a strict clause. Clauses with no positive literals are called goal clauses. This clause

$$\neg x_1 \vee \neg x_2 \vee ... \vee \neg x_n$$

would represent a goal clause. Clauses with no negative literals are called facts [13].

3.3 What are Constrained Horn Clauses

Constrained Horn Clauses (CHCs) are a language in so-called first-oder-logic (FOL) used to specify semantics and properties of programs and use the same syntax and semantic as Constraint Logic Programming (CLP). Data structures like booleans, integers, arrays, bit vectors and lists are some of the theories that underlie CHCs [8].

3.4 Basic Terminology

In view of logic programming Horn Clauses in the form of

$$\forall x, y, z \ . \ q(y) \wedge r(z) \wedge \varphi(x, y, z) \rightarrow p(x)$$

are represented as

$$p(x) \leftarrow q(y), r(z), \varphi(x, y, z)[5]$$

```
1 ∏  ::= chc ∧ ∏ | ⊤
2 chc  ::= ∀var . chc | body → head
3 pred  ::= upred | φ
4 head  ::= pred
5 body  ::= ⊤ | pred | body ∧ body |  ∃var . body
6 upred  ::= an uninterpreted predicate applied to terms
7 φ  ::= a formula whose terms and predicates are interpreted over 𝒜
8 var  ::= a variable
```

Figure 3.1: Construction of CHC [5]

The figure 3.1 shows the construction of Constrained Horn Clauses and the needed terminology for them. It is now to be mentioned that we use the same syntax and definition as Bjørner et al. [5]. In this case, the symbol $\prod$ stands for a conjunction of CHCs and *chc* stands for a single clause. Therefore, the syntax for a conjunction of CHCs can either be a single *chc AND* a conjunction of CHCs or a tautology as seen in the first definition. The next representation in definition number two states that the syntax for a single *chc* is either a *body* implying a *head* or the recursive definition of *chc* with $\forall var$ that hold for a *chc*. *Pred*, as seen in the third definition, stands for predicate and is defined as either a uninterpreted predicate (*upred*) or a formula (φ). The *head* simply refers to a predicate. The syntax for a *body* is defined as either a tautology, a predicate, a *body AND* a *body* or the existence of a *variable* that holds for *body*. The definitions 6, 7 and 8 explain themselves. The $\mathcal{A}$ mentioned in definition 7 refers to an assertion language used as an example by Nikolaj Bjørner et al. [5].

3.5 Negation Normal Form Constrained Horn Clauses

The algorithm that is used to translate code into Constrained Horn Clauses as seen later in chapter 4 has the advantage that Constrained Horn Clauses in the form of negation normal form (NNF) fit very well into it [5]. The NNF of Constrained Horn Clauses is defined as follows:

```
∏  ::= chc ∧ ∏ | ⊤
chc  ::= ∀var  . chc | body → ∏ | head
head  ::= pred
body  ::= body ∨ body | body ∧ body | pred | ∃var  . body
```

Figure 3.2: NNF Horn [5]

As seen in figure 3.2, the definition of a conjunction of clauses stays the same as in our original definition of Constrained Horn Clauses seen in figure 3.1. Moving on to the second definition, this now states that the syntax of a single clause is now defined to be either a *head*, a *body* that implies a conjunction of Constrained Horn Clauses or the recursive definition of *chc* with $\forall var$ that hold for *chc*.

With those definitions we now have the basics on the theory on Constrained Horn Clauses that we need to further translate code into clauses in the next chapter.

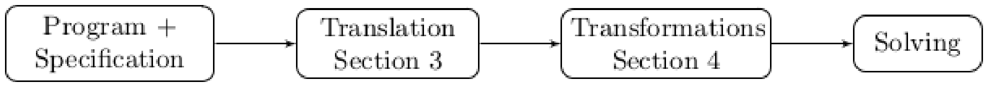

Figure 3.3: Program Verification using CHCs [5]

The figure 3.3 shows the flow of the process for program verification using CHCs. The sections mentioned in this figure refer to the sections in "Horn Clause Solvers for Program Verification" [5].

3.6 Defining the Programming Language

The following programming language described and used throughout this thesis is similar to the Boogie [14] system defined as:

```
program ::= decl*
decl ::= def p(x){local v; S}
S ::= x := E | S₁; S₂ | if E then S₁ else S₂ | S₁ □ S₂
          | havoc x | assert E | assume E | while E do S
          | y := p(E)
E ::= arithmetic logical expression
```

Figure 3.4: Programming Language [5]

The **havoc***(x)* statement is used to assign an arbitrary value to the variable x. The statement $S1\Box S2$ is used to choose either S_1 or S_2 non-deterministically to further run the program. For simplification reasons function calls are restricted to only one parameter x in the definition, but when writing a program with this syntax there can be more than one parameter. However, note that there is no notation on return values. When writing programs, there is the restriction in this case that whenever something is to be returned, the variable used for that has to be called *ret* [5].

3.7 CHC Examples

3.7.1 Example 1

The following clauses correspond to the clauses for the program seen in figure 1.1. If those Constrained Horn Clauses given[7] as a set are satisfied then the validity of the given Hoare triple is proven and the program code is verified [8].

```
false ← M > Sum, M ≥ 0, sum_upto(M, Sum)
sum_upto(X, R) ← R0 = 0, while(X, R0, R)
while(X1, R1, R) ← X1 > 0, R2 = R1 + X1, X2 = X1 - 1, while(X2, R2, R)
while(X1, R1, R) ← X1 ≤ 0, R = R1
```

Figure 3.5: The corresponding Clauses for the Program 1.1

Note that these clauses have not been generated with the algorithm introduced later on. However, they provide a solid structure for the understanding on what program part is covered in which clause. Regarding this, clause number one corresponds to the start of the program in its initial state. Then in clause number two we are in the line of code initializing the value of r with 0 and the initial state of the while loop. Clause

[7]For now it is enough to just consider the clauses as given. We will later see how we can build our own clauses in chapter 4.

number three shows the execution of the while loop, while on the other hand the last clause shows the state in which the loop is not executed.

As mentioned by E. De Angelis et al. [8] in clause number one the statement $M > Sum$ is the negation of our postcondition thus implying the state of false. This whole clause is to be understood as follows: false is to be derived if the precondition holds true but the postcondition does not. The whole system of CHCs holds true if we do not derive false.

As mentioned, the clauses seen before have not been generated using the algorithm from Section 4.2. The following Constrained Horn Clauses however were generated using the algorithm and ultimately provide the same outcome, namely the program being verified as working correctly.

```
sum_upto_pre(x) ← (x ≥ 0)
(r ≥ x) ← (x ≥ 0), sum_upto(x, r)
inv(0, x, x0) ← (x0 = x), sum_upto(x)
inv(r'+x', x'-1, x0') ← x0 = x, sum_upto(x), inv(r', x', x0'), (x' > 0)
sum_upto(x0', r') ← x0 = x, sum_upto(x), inv(r', x', x0'), ¬(x'>0)
```

Figure 3.6: The corresponding Clauses for the Program 1.1 as generated by the Weakest Liberal Precondition Algorithm

3.7.2 Example 2

For demonstration purpose we will have a look at another example, taken from the paper by Nikolaj Bjørner et al. [5]. This example will be easier to translate using the proposed algorithm later on as it has no if-else statements, as well as no while loops in it. The following program as seen in figure 3.7 consists of function declarations, also called procedure declarations, in the form of **def** *p(x)* and function calls also called procedure calls within the declarations. The **def** *main(x)* is a special procedure functioning as the entry point of the program [5].

```
def main ( x ) {
  assume init(x)
  z := p(x)
  y := p(z)
  assert φ₁ (y)
}

def p ( x ) {
  z := q(x)
  ret := q(z)
  assert φ₂ (ret)
}

def q ( x ) {
  assume ψ(x, ret)
}
```

Figure 3.7: Procedure Call Example [5]

The next figure shows the Constrained Horn Clauses that correspond to the code seen in the figure above. In this case *init* defines the precondition while φ_1 and φ_2 are the postconditions. ψ is the definition of the behaviour of the procedure q [5]. With the code seen above we can use the algorithm *ToHorn* that will later be discussed in chapter 4 to then generate those clauses by our own.

For the sake of completeness the corresponding Hoare Triple regarding this code looks as follows:

$$\{\textbf{assume } init(x)\}\ main(x)\ \{\textbf{assert } \varphi_1(x)\}$$

```
main(x) ← ⊤
φ₁(y) ← main(x), init(x), p(x, z), p(z, y)
p_pre(x) ← main(x), init(x)
p_pre(z) ← main(x), init(x), p(x, z)
p(x, y) ∧ φ₂(y) ← p_pre(x), q(x, z), q(z, y)
q_pre(x) ← p_pre(x)
q_pre(z) ← p_pre(x), q(x, z)
q(x, y) ← q_pre(x), ψ(x, y)
```

Figure 3.8: The corresponding Clauses for the Program 3.7 [5]

4 Translating Code into CHCs

One major aspect of verifying programs with Constrained Horn Clauses is the correct translation of code into the clauses that then are to be proven. In this chapter we will have a look on how to do so based on the algorithm presented by Nikolaj Bjørner et al. [5]. Our implementation of this algorithm is then presented in the chapter 6.

4.1 Weakest Liberal Precondition

First we define the weakest precondition (*wp*). Given a program S and the corresponding postcondition Q, the corresponding weakest precondition is a predicate that for any given precondition guarantees that the postcondition holds and does so while being the requirement that is least restrictive. Then from this we can derive the definition of the weakest liberal precondition (*wlp*). The weakest liberal precondition is the same as the weakest precondition but has the difference that it does not guarantee the termination of the program. This means that the *wlp* either gives you the weakest precondition under the assumption that the program terminates or the weakest precondition to fulfill the postcondition. This is denoted as $wlp(S, Q)$ where S stands for the program and Q is the postcondition [15] [16].

$$\{P\}S\{Q\} \iff P \Rightarrow wlp(S, Q)$$

Figure 4.1: Weakest Liberal Precondition [17]

4.2 The Algorithm

The figure 4.2 shows the algorithm *ToHorn* used by Nikolaj Bjørner et al. [5] to translate code into Constrained Horn Clauses using the weakest liberal precondition. The original algorithm consists of two more rules that define the behaviour on **goto** statements that we do not make use of. Note that the $\boldsymbol{\omega}$ seen next is used as the set of all variables within the scope of a function/procedure.

The practical usage of $ToHorn$ is best described when looking at a example on how the rules are applied to result in CHCs. The following example in the next section 4.3 is done by hand but will ultimately be the same when done by the implementation with the difference of having different variable names when substitutions take place.

$$1)\ \mathit{ToHorn}(\mathit{program}) := \mathit{wlp}(\mathit{Main}(), \top) \land \mathit{ToHorn}(\mathit{decl}) \bigwedge_{\mathit{decl} \in \mathit{program}} \mathit{ToHorn}(\mathit{decl})$$

$$2)\ \mathit{ToHorn}(\mathbf{def}\, p(x)\ \{S\}) := \mathit{wlp}\begin{pmatrix} \mathbf{havoc}\ x_0;\ \mathbf{assume}\ x_0 = x; \\ \mathbf{assume}\ p_{pre}(x); S, \quad p(x_0, \mathit{ret}) \end{pmatrix}$$

$$3)\ \mathit{wlp}(x := E, Q) := \mathbf{let}\ x = E\ \mathbf{in}\ Q$$

$$4)\ \mathit{wlp}((\mathbf{if}\ E\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2), Q) := \mathit{wlp}(((\mathbf{assume}\ E; S_1)\square(\mathbf{assume}\ \neg E; S_2)), Q)$$

$$5)\ \mathit{wlp}((S_1 \square S_2), Q) := \mathit{wlp}(S_1 Q) \land \mathit{wlp}(S_2, Q)$$

$$6)\ \mathit{wlp}(S_1; S_2, Q) := \mathit{wlp}(S_1, \mathit{wlp}(S_2, Q))$$

$$7)\ \mathit{wlp}(\mathbf{havoc}\ x, Q) := \forall x\ .\ Q$$

$$8)\ \mathit{wlp}(\mathbf{assert}\ \varphi, Q) := \varphi \land Q$$

$$9)\ \mathit{wlp}(\mathbf{assume}\ \varphi, Q) := \varphi \rightarrow Q$$

$$10)\ \mathit{wlp}((\mathbf{while}\ E\ \mathbf{do}\ S), Q) := \mathit{inv}(\boldsymbol{\omega}) \land \forall \boldsymbol{\omega}\ .\ \begin{pmatrix} ((\mathit{inv}(\boldsymbol{\omega}) \land E) \rightarrow \mathit{wlp}(S, \mathit{inv}(\boldsymbol{\omega}))) \\ \land\ ((\mathit{inv}(\boldsymbol{\omega}) \land \neg E) \rightarrow Q) \end{pmatrix}$$

$$11)\ \mathit{wlp}(y := p(E), Q) := p_{pre}(E) \land (\forall r\ .\ p(E, r) \rightarrow Q[r/y])$$

Figure 4.2: The Algorithm ToHorn [5]

4.3 Applying the Algorithm to Code

4.3.1 Example on Procedure Call Code

Lets remember the program from figure 3.7 and their corresponding CHCs from figure 3.8. We now want to apply *ToHorn* to see how that works in practice. This of course is then done automatically by the implementation of the tool. Lets start by applying the first rule thus splitting the program:

$$\mathit{ToHorn}(\mathit{program}) \ := \ \mathit{wlp}(\mathit{Main}(), \top) \ \land \ \mathit{ToHorn}(\mathbf{def}\, \mathit{main}(x)\{S_1\} \ \land$$
$$\mathit{ToHorn}(\mathbf{def}\, p(x)\{S_2\}) \ \land \ \mathit{ToHorn}(\mathbf{def}\, q(x)\{S_3\})$$

We will for now only look at the construction of the clause for the procedure **def** $q(x)$ but for the other procedures the procedure of translation is the same but longer. We now apply the second rule.

$$\mathit{ToHorn}(\mathbf{def}\, q(x)\ \{S_3\}) := \mathit{wlp}\begin{pmatrix} \mathbf{havoc}\ x_0;\ \mathbf{assume}\ x_0 = x; \\ \mathbf{assume}\ q_{pre}(x);\ \mathbf{assume}\ \psi(x, \mathit{ret}), \quad q(x_0, \mathit{ret}) \end{pmatrix}$$

The sixth rule can now be applied resulting in

$$\mathit{ToHorn}(\mathbf{def}\, q(x)\ \{S_3\}) := \mathit{wlp}\begin{pmatrix} \mathbf{havoc}\ x_0;\ \mathit{wlp}(\mathbf{assume}\ x_0 = x; \\ \mathbf{assume}\ q_{pre}(x);\ \mathbf{assume}\ \psi(x, \mathit{ret}), \quad q(x_0, \mathit{ret})) \end{pmatrix}$$

The **havoc** rule further allows us to change the existing clause to

$$\mathit{ToHorn}(\textbf{def}\ q(x)\ \{S_3\}) := \forall x_0\ .\ \ wlp(\textbf{assume}\ x_0 = x;$$

$$\textbf{assume}\ q_{pre}(x);\ \textbf{assume}\ \psi(x, ret),\quad q(x_0, ret))$$

We now apply the rule number six again and combine it with the **assume** rule leading to

$$\mathit{ToHorn}(\textbf{def}\ q(x)\ \{S_3\}) := \forall x_0\ .\ x0\ =\ x\ \to wlp(\textbf{assume}\ q_{pre}(x);\ \textbf{assume}\ \psi(x, ret),$$

$$q(x_0, ret))$$

Again rule number six gives us

$$\mathit{ToHorn}(\textbf{def}\, q(x)\ \{S_3\}) := \forall x_0\, .\, x0\ =\ x\ \to wlp(\textbf{assume}\, q_{pre}(x);\ wlp(\textbf{assume}\, \psi(x, ret),$$

$$q(x_0, ret)))$$

By applying the **assume** rule twice we get

$$\mathit{ToHorn}(\textbf{def}\ q(x)\ \{S_3\}) := \forall x_0\ .\ x0\ =\ x\ \to (q_{pre}(x) \to (\psi(x, ret) \to$$

$$q(x_0, ret)))$$

We now want to remove x_0 by replacing it with x.

$$\mathit{ToHorn}(\textbf{def}\ q(x)\ \{S_3\}) := \forall x\ .\ (q_{pre}(x) \to (\psi(x, ret) \to\ q(x, ret)))$$

By simplifying this with the equivalence of

$$a \to (b \to c) \iff (a \wedge b) \to c$$

we get the clause

$$\mathit{ToHorn}(\textbf{def}\ q(x)\ \{S_3\}) := \forall x\ .\ (q_{pre}(x) \wedge \psi(x, ret)) \to\ q(x, ret).$$

We now inverse the implications leaving us with the final clause seen in 3.8 with the only difference on having the variable *ret* instead of the variable *y*.

$$q(x, ret) \leftarrow q_{pre}(x), \psi(x, ret)$$

As mentioned, applying *ToHorn* to the other parts of the program will ultimately lead to all of the CHCs for this program. This can of course also be done to the program seen in figure 1.1 but by hand this translation leads so a lot of steps to be done until the final Constrained Horn Clauses are derived due to its nature of having the while loop in it.

It is to be mentioned that the algorithm needs a *main()* function to lead to the correct clauses. For this we will have some simple changes in our program to adapt it to the specifications needed. First we will have to define a *main()* function that calls our function *sum_upto(x)*. We will also add our precondition as well as the postcondition to the *main()* function and remove them from the function *sum_upto(x)*. Theoretically one could also check again in the function *sum_upto(x)* without violating the rules but this would be redundant in this case. These changes lead to the code from figure 1.1 looking as seen in the following figure 4.3.

```
def main () {
  assume (x ≥ 0)
  sum := sum_upto(x)
  assert (sum ≥ x)
}

def sum_upto(x) {
  ret := 0
  while (x > 0) {
    ret := ret + x
    x := x - 1
  }
}
```

Figure 4.3: Adapted Code from Figure 1.1

5 Spacer and Z3

Now we have covered all the theories on how to translate a given code into Constrained Horn Clauses. The remaining part would now be to verify the validity of the clauses and with that also implying the validity of the program. There are some technologies out there that do so and in this chapter we will have a look at Spacer used for example by *SeaHorn* [6] and also by Microsoft's Z3 [1].

5.1 What is Spacer

Spacer is a Satisfiability Modulo Theories (SMT) based CHC solver that by now has been integrated into the Z3 solver by Microsoft and is the default for solving CHCs in Z3 as well as being used as already mentioned by Seahorn. The SMT - Theories supported by Spacer are linear real arithmetic, nonlinear arithmetic, integer arithmetic, quantifier-free theory of arrays, data-structures and bit-vectors [11].

To get an idea of where the Spacer solver takes its place in the process of verification, take a look at Figure 5.1 that shows the verification process in the software Seahorn [6].

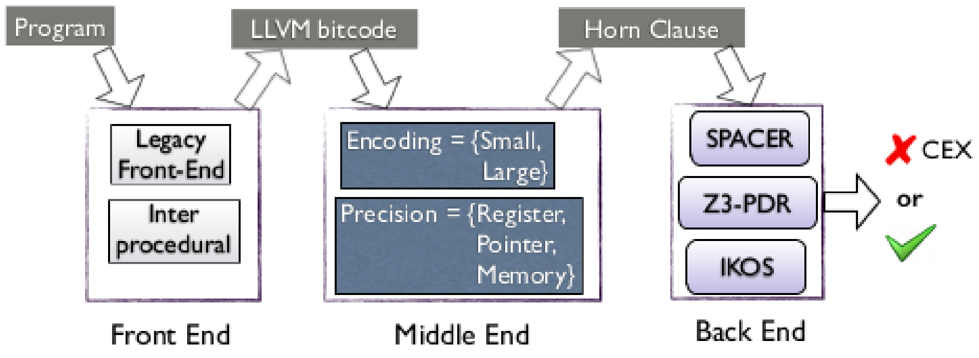

Figure 5.1: Seahorn Verification of a Program [6]

Furthermore, Spacer for example, extends other technologies like IC3 [18] and PDR [19] from the Boolean Satisfiability Problem (SAT) to Satisfiability Modulo Theories (SMT) and from proposition logic to Linear Integer Arithmetic (LIA) plus arrays [20][11][21].

5.2 How does Spacer work

Lets define a set of Constrained Horn Clauses as Φ. We now give this set to Spacer and that set now is worked through iteratively. Spacer is looking for bounded derivations that are false in retrospect of Φ that is explored backwards also called top-down. Whenever Spacer fails and no derivation is found, it is analyzed why it fails to further derive consequences of Φ that then gives an explanation of why at least N + 1 steps are derived for a derivation of false. This then is repeated until one of the following three options occur. If false is derived, then Φ is proved unstatisfiable. If a solution is formed for them then Φ is satisfiable. The last option would be that the procedure does not end, implying either that there is no solution or that there at least is no short proof for the unstatisfiability [11].

5.3 Z3 Usage on Code Example 1

The last step in this procedure of program verification is the proving of the CHCs. For this we will use the tool provided by Microsoft called Z3 as already mentioned that uses Spacer.

Lets remember the program seen in figure 1.1 and the corresponding CHCs seen in figure 3.5. Z3 provides a python library that can be imported by the command $from\ z3\ import *$ after installing Z3 on your device. We now define a solver:

```
s = SolverFor("HORN")
```

and start adding our clauses to our solver. First we must define used variables and the functions *sum_upto* and *sum_upto_pre* as well as our invariant. The invariant must also be declared as a function.

```
sum_upto = Function('sum_upto', IntSort(), IntSort(), BoolSort())
sum_upto_pre = Function('sum_upto', IntSort(), BoolSort())
inv = Function('inv', IntSort(), IntSort(), IntSort(), BoolSort())
```

The definition of integer variables for example looks like this:

```
x = Int('x')
```

Adding the first clause

$$sum\_upto\_pre(x) \ \leftarrow \ (x \geq 0)$$

to our solver looks as followes:

```
s.add(ForAll([x], Implies(x ≥ 0, sum_upto_pre(x))))
```

After adding all clauses to the solver we can check if the clauses are satisfiable with the command

```
s.check()
```

and finally become the result of *sat* stating that the program is correct.

6 Implementation of the Tool

After talking of all the theories behind program verification and Constrained Horn Clauses we will now discuss our own implementation of a verification tool. The idea of the tool is to take a program as input, transform it into Constrained Horn Clauses and then forward those to Z3 [1] to prove it right or wrong. The flow of the tool itself is seen in the following Control-Flow-Graph.

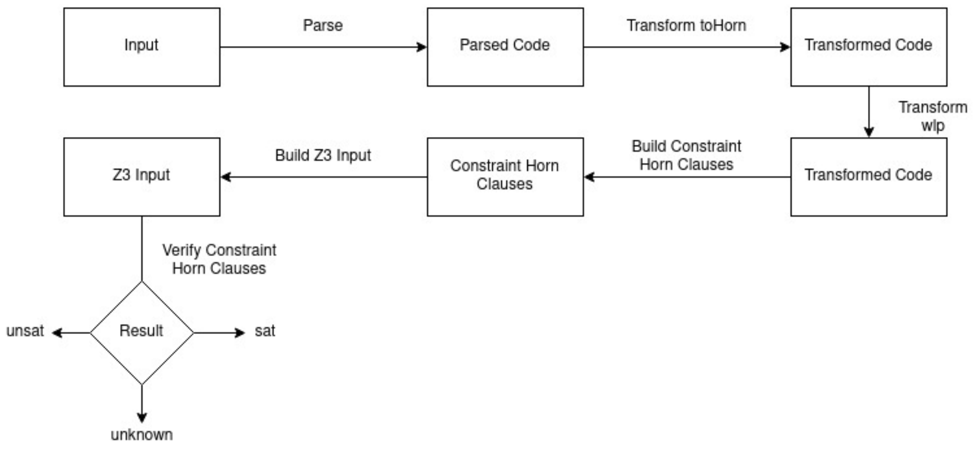

Figure 6.1: CFG of the Tool

The tool is written in Scala [7] using the version *scala-sdk-2.11.12*. For the installation of the needed dependencies have a look into the README.md file. The project is available on `https://github.com/thenextmz/bachelorThesis/releases/tag/v1.0.0`.

6.1 Input

The tool takes a .txt file as program argument when started. This single or multi-line text document contains the code written as Boogie language type code [14].

6.2 Parsing

The first step the tool now does is parsing the text input into Scala case classes [7] that are equivalent to the definition of the Boogie language. This is done with the parser combinator [22] library provided by Scala called *scala.util.parsing.combinator.JavaTokenParsers*

[23]. For example the definition of the if-statement seen in figure 3.4 looks as follows in the code:

```
case class IfElse(E: Expr, S1: Body, S2: Body) extends Statement
```

In this case the definition of an if-statement consists of a logical expression *E* and two Bodies *S1, S2* that contain statements.

6.3 ToHorn

The parsed code is then forwarded to the function *toHorn* that transforms the parsed input for every definition of a function into the corresponding wlp-statement seen in figure 4.2.

When translating definitions there are a few differences regarding whether the function is the main-function or not and also if the function has arguments or not. If the function is the main-function that means that it hast no $p_{pre}(x)$ that is to be added to the wlp transformation. Also as the main does not return the statement $p(x_0, ret)$ is replaced with a $\top$.

Regarding the parameters the *havoc* x_0 statement and the *assume* $x_0 = x$ statement are only added if the function has parameters. The *toHorn* function returns a tuple consisting of a one-dimensional Listbuffer containing the wlp-element and a list of variables that occur in the procedure itself as they are needed for the while rule of the wlp-algorithm later on. The variables inside a procedure are searched and stored by another function called *getAllVariables*. Unlike most of the other functions in the implementation the *toHorn* function is not recursive as this is not needed in this case.

6.4 Wlp-Algorithm

To compute the weakest liberal preconditions we use the function called *wlpAlgorithm*. This function is a one to one implementation of the algorithm seen in figure 4.2. It tries to match the given statements to one of the rules and transforms them. The function calls itself recursively as the transformation rule number six splits the statements and uses the wlp itself as *Q* statement. The outcome of this function is a set of rules consisting of the *Implies-*, *And-* and *Forall*-rule. This set of rules leads to a tree structure. The following example in figure 6.2 visualizes the need of the tree structure in rules.

```
Implies(A, B)
A := Implies(C, D)
B := Implies (E, F)
⇒ Implies(Implies(C, D), Implies(E, F))
```

Figure 6.2: Tree Structure of Rules

6.5 Simplify

As the outcome of the program by now is one single Constrained Horn Clause containing all rules we want to split the Constrained Horn Clause into smaller parts representing the representation seen in figure 3.1 as by now they are in negation normal form as seen and discussed in the chapter 3.5. This is done by the *simplify* function that applies three transformation rules seen and described next.

$$Rule\ 1: \ a \rightarrow (b \wedge c) \Leftrightarrow (a \rightarrow b) \wedge (a \rightarrow c)$$
$$Rule\ 2: \ a \rightarrow (b \rightarrow c) \Leftrightarrow (a \wedge b) \rightarrow c$$

The third rule is the handling of inner for-all's. In this case the transformation takes place by first changing all variable names captured by the for-all inside of the statement that it holds and adding them to the outer for-all of the CHC set.
Assume the following CHC for demonstration:

$$forall\ (x, x_0) : (x_0 = x) \rightarrow (... \rightarrow ... \wedge forall\ (x, x_0, ret) : ((ret > 0) \wedge (x > 0)\ ...$$

Now when simplifying, the variables in the inner for-all get renamed to x_1, x_{01} and ret_1. With that we can now insert them into the outer for-all, rename the occurrence inside of the inner for-all with the new variable names and remove the inner for-all statement completely resulting in the simplified CHC that looks as follows:

$$forall\ (x, x_0, x_{01}, x_1, ret_1) : (x_0 = x) \rightarrow (... \rightarrow ... \wedge ((ret_1 > 0) \wedge (x_1 > 0)\ ...$$

For this to work there is the condition that the for-all statements have to be positive meaning that for example a negated for-all statement looking like

$$\neg(\forall\ a : \ a)$$

would not be valid. This is always the case in these expressions generated as NNF-Horn rules. Also only the free occurrences of variables are renamed meaning that captured variables are not renamed. This would be the case if the for-all statement has another for-all statement inside of itself that captures the same variable names as the ones we want to rename.

6.6 Generate Z3 Input

The *generateZ3Input* function now takes the finished CHCs and transforms them to strings corresponding to the Python syntax including the Z3 Python Library as in Z3 version 4.8.14. As expected from Z3 we first need to get all variables used in the Constrained Horn Clauses to define them. Also all functions and invariants need to be defined with the right parameter number. An example transformation can be seen next.

```
Implies(E3(Variable(x,false),>=,Constant(0),false),Function(sum_upto_pre,
    Variable(x,false)))
```

The representation seen above gets transformed into:

```
Implies((x>=0),sum_upto_pre(x))
```

After that the main function writes all the generated lines of code into a file called *z.py* to then finally execute the newly generated Python program.

6.7 Output

The tool provides some feedback on what was done in the background. First the parsed code is displayed. Next come the steps of the *toHorn* and *wlpTransformation* functions followed by the generated Constrained Horn Clauses. For verification only the last output is important as it then states if the program was either *sat*, *unsat* or *unknown*. If it was *sat* then also a model generated by Z3 is printed. The model contains a representation that interprets the assignment of constants and functions. If there are loops in the program that was verified the generated invariants are also in the model [24].

```
def main(){
  assume (x > 0)
  result := sub(x, y)
  assert (result == (x - y))
}

def sub(x, y){
  ret := (x - y)
}
```

Figure 6.3: Program for Model Example

Considering the program seen in figure 6.3 that calls a function called *sub* and returns the result of x - y, the model corresponding to the satisfied Constrained Horn Clauses looks as follows:

```
[sub_pre = [else -> Not(Var(0) <= 0)],
 sub = [else ->
        And(Not(Var(0) <= 0), Var(2) == Var(0) + -1*Var(1))]]
```

Figure 6.4: Model of Program

This is to be understood as follows. The predicate *sub_pre* is the precondition for the function *sub* to ensure the correctness for the surrounding program. The predicate *sub* describes the relation between input and output of the function.

7 Usage of the Implementation

7.1 Using the Tool with the sum_upto Exmaple

As one now wants to test a program the only thing to do is write it according to the syntax specified in figure 3.4. When using the tool the first argument passed to it is the path of the program to verify. After that clauses are generated, the Python file is generated and Z3 solves them also again ultimately leading to the result of *sat* for this program, but this time done all automatically with the tool.

7.2 Another Example

As we now take a look on another example the benefits of automatic program verification become clearer. The following program seen in the next figure 7.1 is now to be verified and doing so by hand would take a lot of transformation steps ultimately leading to 16 Constrained Horn Clauses. The program itself consists of three procedures including the main function and first sums up two parameters given to the procedure *sum_upto* and then also checks if two other variables given to the procedure *both_even* are even numbers. This leads to the result of *sat* as all postconditions hold true after execution with corresponding preconditions. Changing the program in a way that the preconditions violate the postconditions or vice versa leads to *unsat*. In this specific example one could say that either the variable *z* or the variable *z2* or both are not even numbers in the assumption in line 4 and 5 and thus the *assert* in line 9 would not hold true leading to *unsat*. Also changing the *assert* from line 9 to $assert(sum <= (x + y))$ leads to *unsat* as result as this would not represent the behaviour of the program.

```
def main(){
  assume (x >= 0)
  assume (y >= 0)
  assume ((z % 2) == 0)
  assume ((z2 % 2) == 0)
  sum := sum_upto(x, y)
  even := both_even(z, z2)
  assert (sum >= (x + y))
  assert (even == 1)
}
def sum_upto(x, y){
  ret := 0

  while (x > 0){
    ret := ret + x
    x := x - 1
  }

  while (y > 0){
      ret := ret + y
      y := y - 1
    }
}

def both_even(z, z2){
    if((z % 2) == 0){
      ret := 1
    } else {
      ret := 0
    }

    if(ret == 1){
      if((z2 % 2) == 0){
        ret := 1
      } else {
        ret := 0
      }
    } else {
      ret := 0
    }
}
```

Figure 7.1: Another Code Example used with the Tool

7.3 A brief Benchmark on Run Time

One efficiency trait using this verification algorithm lies in the creation of invariants. As seen for example the usage of while loops inside of the code that is to be verified creates invariants in the generated Constrained Horn Clauses that replace the while loop. Lets for example think of a while loop with millions of iterations. Verifying the behaviour now always only takes the invariant, no matter how many iterations the loop has. Using proving algorithms that use the iteration when verifying would have a significantly higher run time when increasing the loop iterations. The downside of the invariant comes with the number of invariants needed. Nested or multiple while loops for example create more invariants that need to be processed and thus increasing the run time regarding the algorithm used in this thesis.

Although the run time when testing was mostly very short leading for example to a total run time for the code seen in 4.3 of roughly 0.4 seconds there are cases where the run time drastically increases due to the verification process of Z3.

```
def main(){
  assume (x > -10)
  assume (x < 10)
  assume ((x % 2) == 0)
  even := oddEven(x)
  assert (even == 1)
}

def oddEven(x){
  if(x == 0){
    ret := 1
  } else {
    if(x > 0){
      x := x
    } else {
      x := -x
    }

    while (x > 0){
      x := x - 2
      if(x == 0){
        ret := 1
      } else {
        ret := 0
      }
    }
  }
}
```

Figure 7.2: Program with High Runtime

The previous figure 7.2 shows a program havig the just mentioned run time problem. Line 2 and 3 show restrictions on the variable x to be between -10 and 10 thus assuring termination of the verification process in about 0.4 seconds. Increasing the limits to be between -100 and 100 leads to a significantly higher run time that was about 8.5 seconds. This effect comes from the fact that Z3 builds invariants here that have every possible solution in them and without restrictions this will, as already mentioned, never terminate. A part of the invariant build by Z3 can be seen in the next figure.

```
inv1 = [else ->
          And(Not(Var(0) >= 99),
              Or(Not(Var(0) <= 0), Not(Var(2) >= 2)),
              Or(Not(Var(0) <= 0), Not(Var(2) <= 0)),
              Not(Var(0) == 3),
              Not(Var(0) == 1),
              Not(Var(0) == 5),
              Not(Var(0) == 7),
              Not(Var(0) == 9),
              Not(Var(0) == 11),
              Not(Var(0) == 13),
              Not(Var(0) == 15),
              Not(Var(0) == 17),
              Not(Var(0) == 19),
              Not(Var(0) == 21),
              ...
```

Figure 7.3: Invariant with Bounding

As the corresponding program to this invariant calculates if a variable is even, the idea behind the generated invariant becomes more clear as it represents all possibilities of variables that would be odd with the restriction of the variable being smaller than 100.

8 Conclusion

With that we now have covered several topics on program verification. First we introduced the idea of Hoare logic and then the theories on Constrained Horn Clauses. Then with that we introduced the algorithm on weakest liberal precondition transformation in combination with the Hoare logic to generate Constrained Horn Clauses from code. With this in mind we talked about the verification process of Constrained Horn Clauses as done by Spacer and Z3 and further introduced our implementation of the tool that builds on all the previous covered topics. The tool was presented in theory and also the practical usage was shown along with its benefits and downsides on run time.

As seen, CHCs are a powerful theory to prove the validity of programs used in this field and implemented successfully by many companies. With the knowledge about them they can be applied in those mentioned implementation and be used for the purpose of program verification assuring the security of wanted outcome of programs. In addition to that we are now not only able to prove given Constrained Horn Clauses correct, but we also have the tool introduced in this thesis to automatically generate them from code input.

However, there are a lot more extensions that could be done regarding this tool like for example trying other techniques for translation and compare those in regards of efficiency. Also the expansion of the covered theories of the tool to other theories as for example strings and arrays could be considered. The last step done by the theorem solvers is by now only briefly mentioned and could also lead to further work regarding deeper insights on how they work, other approaches and also working on own solutions.

It is to be mentioned that by now, if we have a program that is unsatisfied, we have no feedback regarding the information on where the bug might be leading to even more further work that could be done as seen for example in SMT solvers by the approach of providing counter examples and error tracing.

Bibliography

[1] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2008, pp. 337–340.

[2] A. Horn, "On sentences which are true of direct unions of algebras1," *The Journal of Symbolic Logic*, vol. 16, no. 1, pp. 14–21, 1951.

[3] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[4] R. W. Floyd, "Assigning meanings to programs," *Proceedings of Symposium on Applied Mathematics*, vol. 19, pp. 19–32, 1967. [Online]. Available: `http://laser.cs.umass.edu/courses/cs521-621.Spr06/papers/Floyd.pdf`.

[5] N. Bjørner, A. Gurfinkel, K. McMillan, and A. Rybalchenko, "Horn clause solvers for program verification," in *Fields of Logic and Computation II*, Springer, 2015, pp. 24–51.

[6] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, "The seahorn verification framework," in *International Conference on Computer Aided Verification*, Springer, 2015, pp. 343–361.

[7] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "An overview of the scala programming language," 2004.

[8] E. De Angelis, F. Fioravanti, J. P. Gallagher, M. V. Hermenegildo, A. Pettorossi, and M. Proietti, "Analysis and transformation of constrained horn clauses for program verification," *arXiv preprint arXiv:2108.00739*, 2021.

[9] M. Gordon, "Background reading on hoare logic," *Lecture Notes, April*, 2012.

[10] C. A. Furia, B. Meyer, and S. Velder, "Loop invariants: Analysis, classification, and examples," *ACM Computing Surveys (CSUR)*, vol. 46, no. 3, pp. 1–51, 2014.

[11] A. Gurfinkel and N. Bjørner, "The science, art, and magic of constrained horn clauses," in *2019 21st International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, IEEE, 2019, pp. 6–10.

[12] F. Wotowa, *Software-paradigmen skriptum zur lehrveranstaltung*, 2007.

[13] M. H. Van Emden and R. A. Kowalski, "The semantics of predicate logic as a programming language," *Journal of the ACM (JACM)*, vol. 23, no. 4, pp. 733–742, 1976.

[14] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *International Symposium on Formal Methods for Components and Objects*, Springer, 2005, pp. 364–387.

[15] Wikipedia contributors, *Predicate transformer semantics — Wikipedia, the free encyclopedia*, [Online; accessed 25-November-2021], 2021. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Predicate_transformer_semantics&oldid=1056113106`.

[16] A. E. Santosa, "Comparing weakest precondition and weakest liberal precondition," *arXiv preprint arXiv:1512.04013*, 2015.

[17] A. Gurfinkel, T. Kahsai, J. A. Navas, A. Komuravelli, and N. Bjorner, "Building program verifiers from compilers and theorem provers," CARNEGIE-MELLON UNIV PITTSBURGH PA PITTSBURGH United States, Tech. Rep., 2015.

[18] A. R. Bradley, "Understanding ic3," in *International Conference on Theory and Applications of Satisfiability Testing*, Springer, 2012, pp. 1–14.

[19] N. Eén, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," in *2011 Formal Methods in Computer-Aided Design (FMCAD)*, IEEE, 2011, pp. 125–134.

[20] K. Suenaga and T. Ishizawa, "Generalized property-directed reachability for hybrid systems," in *International Conference on Verification, Model Checking, and Abstract Interpretation*, Springer, 2020, pp. 293–313.

[21] A. Gurfinkel, S. Shoham, and Y. Vizel, "Discovering universally quantified solutions for constrained horn clauses," 2018.

[22] D. Leijen and E. Meijer, "Parsec: Direct style monadic parser combinators for the real world," 2001.

[23] scala-lang.org. "JavaTokenParsers." (2022), [Online]. Available: `https://www.scala-lang.org/api/2.12.3/scala-parser-combinators/scala/util/parsing/combinator/JavaTokenParsers.html` (visited on 03/09/2022).

[24] N. Bjørner, L. d. Moura, L. Nachmanson, and C. M. Wintersteiger, "Programming z3," in *International Summer School on Engineering Trustworthy Software Systems*, Springer, 2018, pp. 148–201.