Translating Hoare Logic to SMT, the Making of a Proof Checker

Remco Swenker
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
r.a.swenker@student.utwente.nl

ABSTRACT

Writing the correctness proof of a programme in Hoare Logic is both tedious and error-prone. But programme verification, such as Hoare logic, is important. If it is not used, many computer programmes could be faulty. To lower the frustration of the learning process of Hoare logic and help with a better understanding of programme verification, we made a proof checker that will make this process easier and faster. We use an SMT checker and translate Hoare logic to SMT and interpret the answers from it back to Hoare logic. The prototype of this tool can handle most standard operators. This tool differs from it is competitors in that it gives enough feedback to pinpoint the problem but not spoil the learning process.

1. INTRODUCTION

Programme verification is an important field of computer science. Being able to tell that a piece of software will not misbehave can save lives or prevent millions in damage. Imagine what could happen if the software of a nuclear reactor was faulty. Not all programme errors have such dire consequences, but a programmer that understands verification will avoid a lot of errors.

Hoare logic is one of the methods for program verification that has been in use of a long time and been taught to many students. However, checking the correctness of Hoare logic is a very tedious and error-prone job leading to frustration. Therefore we developed a proof checker for Hoare logic to use in a teaching environment. The student can run the proof checker on his home PC and see if his proof is correct. Thus freeing up allot of time for both teacher and student to do more productive things. However this gives us restrictions to the amount of feedback we can give, because the student should not be able to just ask the checker to do the exercises for them. On the other hand, we cannot be too brief with our answers either, or we would deliver no benefit at all.

Before we started there was no proof checker that gave the type of feedback we were looking for. A program like Boogie can check if code is correct but it only gives yes/no answers. Boogie can even do this with hardly any proof annotations. Then there is a tool called Jape that gives enough feedback but that is not what we are looking for either. Because Jape has the problem that it gives to much feedback and it is no longer a checker but a proof builder. This defeats our purpose, as it allows for very little error, and we want people to make errors and learn from them. A third program KeY is made for Java programs and has a Hoare logic implementation. But is has the same problem as Jape; it spoils the learning process by not letting you make wrong decisions.

Therefore we built our proof checker on top of an SMT solver. This is because SMT is a proof technique that is relatively simple to use and understand, and it is easy to prove the validity of Hoare logic statements when using it. It takes first order logic like statements and and checks if there is a way to make those true. Also SMT solvers give the proper amount of feedback. In this paper and the checker, we use a simple imperative programming language that looks like Java as our input, with Algorithm 1 as a running example. We implemented the tool in Java.

Algorithm 1 Adding one to a non-negative integer.

```
{$x \geq 0$}
$x = x + 1$;
{$x > 0$}
```

The checker can handle a variety of language elements including assignment, ifs, while loops, and functions. The checker was made using a step by step process of implementing one language element at a time to prevent errors with one of the elements form interfering with the new elements.

To make a proof with these elements we transformed the Hoare logic problem into an SMT problem. Then we let the SMT solver prove that piece of code. If we get an error we take the results form the SMT solver and translate these back to Hoare logic. And we then return this counter example to the user.

We will begin this paper with information about Hoare logic and SMT. Then we will go on to explain the tools Boogie, KeY, and Jape. From there, we will continue to the research goal and questions. Then we will explain how we planed to perform this research, followed by the results of implementation and we will end with the conclusion and future work.

2. BACKGROUND

In this section, we will talk about Hoare logic then some terminology about logic and finally about SMT.

2.1 Hoare Logic

Hoare logic was developed in 1969 by Sir. C.A.R. Hoare as a method for proving the correctness of programmes[1, 7].

17th Twente Student Conference on IT June 25th, 2012, Enschede, The Netherlands.

Hoare logic is built upon triples to prove the correctness of programmes, this is written like {P} S {Q}. Triples are made up of what is true at the beginning of the procedure, a precondition P, then the statement S, and what is true at the end of the execution of S, a postcondition Q. A small example of how such a Hoare statement would look like. In our running example, we increment the integer x by one so S $\equiv x = x + 1$. But we only want to do this for non-negative integers, so {P} $\equiv \{x \geq 0\}$. Finally we can say something about x when we're done namely that it will be larger than 0, so {Q} $\equiv \{x > 0\}$. These statements then make the following Hoare logic statement: $\{x \geq 0\}x = x + 1\{x > 0\}$.

In the next part we will talk about five main rules of Hoare logic.

$$\frac{}{\{p[t/x]\}x := t\,\{p\}}\;Assignment\ axiom$$

This is the Assignment axiom, where $p[t/x]$ means the variable x is replaced by the expression t in the assertion p. When we then assign t to x we may conclude p without substitution.

$$\frac{\{p\}S_1\{r\}, \{r\}S_2\{q\}}{\{p\}S_1;\ S_2\{q\}}\;Composition\ rule$$

With the Composition rule if you have two statements not separated with a Hoare predicate, the statements between curly brackets. When the sequential composition of these two statements is correct there must be a postcondition for S_1 that is equal to the precondition of S_2.

$$\frac{\{p \wedge e\}S_1\{q\}, \{p \wedge \neg e\}S_2\{q\}}{\{p\}if\ e\ then\ S_1\ else\ S_2\ fi\{q\}}\;If\ then\ else\ rule$$

The If-Then-Else rule explains how the pre- and postconditions, together with the guard e, get split into a *then* section and an *else* section.

$$\frac{\{p \wedge e\}S\{p\}}{\{p\}while\ e\ do\ S\ od\{p \wedge \neg e\}}\;While\ rule$$

The While rule tells that p is true at the beginning of S and at the end as well, this is known as the loop invariant. And that the negation of the guard e in the postcondition is replaced by e in the precondition.

$$\frac{p \to p_1, \{p_1\}S\{q_1\}, q_1 \to q}{\{p\}S\{q\}}\;Consequence\ rule$$

The Consequence rule allows for modification on the precondition and the postcondition. This of course under some constraints namely that the new precondition is implied by the old one. And the new postcondition implies the old postcondition. This means you can use a weaker precondition and or a stronger postcondition.

We will not prove the correctness of these rules. For proof and detailed examples, we reference to Ten years of hoare's logic[1].

2.2 Terminology

In this section we will explain a few concepts of logic, namely validity, satisfiability, and unsatisfiability. We will also explain the term Hoare predicate in more detail. These terms are used in this paper so we will explain them to avoid confusion.

Validity means that in any valuation of a formula the answer is true. For example, $p \vee \neg p$ is always true.

Satisfiability means that there is at least one valuation of the formula that is true. An example of this is $p \wedge q$. If p is true and q is true, then we have a valuation that gives true. Note that anything that is valid is also satisfiable but not everything that is satisfiable is also valid.

Unsatisfiability means that there is no valuation of the formula that is true. For example, $p \wedge \neg p$ is always false. Note that if something is unsatisfiable, it is never satisfiable, and if something is satisfiable, it is never unsatisfiable. The same is true for validity and unsatisfiability.

Hoare Predicate this is the element of Hoare logic that is placed between curly brackets. When the execution of a program passes over one of these elements it tells what should be true at that moment. So elements before it see it as a postcondition elements after it see it a precondition.

2.3 SMT

SMT stands for Satisfiability Modulo Theories[3, 6]. It is a decision problem for logical formulas expressed in first order logic. To be exact a check whether a given logical formula φ is satisfiable, for a background theory T which constrains the interpretation of the symbols used in φ. To better explain this we will use an example. Suppose we want to know if $x - y < 0$ while $x + y = 0$ is satisfiable, then $\varphi \equiv x - y < 0$ and $T \equiv x + y = 0$. The SMTsolver will then return a result of satisfiable or unsatisfiable. Our example is satisfiable: we can give x the value -1 and y the value 1 and have something that does not violate one of the two statements. Of course SMT does not need two statements but allows an arbitrary amount of statements. This makes it possible that more difficult problems can be solved with hundreds of variables and hundreds of constraints.

A few things to keep in mind when working with SMT solvers. The first is that they are not like a programming language. This means that statements like $x := x + 1$ are not possible. As := is seen as equality and no number is equal to it is own successor. The second is that it has settings that support different data types. Some only allow integers and reals but others also work with booleans and others with arrays. Two programmes that perform SMT logic are called jSMT-LIB and Z3. jSMT-LIB is a Java program and Z3 is created by Microsoft. However, they are similar to use, as they both conform to the SMT-LIB standard. SMT-LIB is an international standard for SMT and SMTsolvers. SMT-LIB defines how the input and output between the user and the system should behave and what it should look like[3].

3. RELATED WORK

In this section of the paper we will talk about three tools, Boogie, KeY, and Jape, that have capabilities that are similar to what we have made. However, they lack certain features that we find very desirable.

3.1 Boogie

Boogie is an intermediate verification language[8], designed to make the prescription of verification conditions natural and convenient. It works as an intermediate language between various source languages and theoremprovers. It can also be used as input output format for other logic techniques. Other features of the Boogie language include being able to handle imperative and mathematical input. And features like parametric polymorphism, partial orders, logical quantifications, non determinism, total expressions, partial statements, and flexible control flow. Boogie is also a tool that can take the boogie language as input and work like a theorem prover instead of just an abstraction layer. When it is used as a verification tool boogie makes use of weakest precondition calculus to calculate if the given program is valid[2].

The difference between boogie and what we made is the following. Boogie proves the correctness of code independent of any provided proof. Our proof checker tells if the proof given with a piece code is correct. So there is a possibility that the given proof is incorrect but but that boogie ignores all the given proof and tells you if the program is right. And we want to check the proof given with the programme, as we want people to learn something about proofs, not the programme it self. Another problem with Boogie is that it gives very little feedback, yes/no answers are not very descriptive for a problem. So it might tell you that it is wrong but not where and how the programme that was tested is wrong.

3.2 KeY

KeY is a proof builder[4]. It guides the user to building a proof showing for each line what kind of Hoare rules and statements should be used. The only time that the user is asked to make more of a addition then clicking next is when a loop invariant is needed. It uses first-order Dynamic Logic for Java as it is proving mechanism. But this has a problem, it is a guided process that will lead the user to the correct answer. So there is no real way to make mistakes with this programme and you learn the most from mistakes. And want people to learn about logic checking and it is pitfalls. In the learning environment, KeY is a better tool than Boogie, because you can see the entire proof as it is being built by KeY.

3.3 Jape

Then there is Jape[5] it is allot like KeY in that it is a proof builder and a guided process that will lead the user to the correct answer. It has A strong GUI for viewing of proof steps. Jape also claims to have a short an shallow learning curve so that it beginners can work with it without to much instruction. Jape is also capable of handling other forms of logic including sequent calculus and natural deduction.

Jape is also more education focussed than Boogie or KeY. As can be deduced from the short learning curve. The difference between Jape and what we made is that Jape helps you make your proof, we check if a proof is correct. So there is no real way to make mistakes with this programme and you learn the most from mistakes. And want people to learn about logic checking and it is pitfalls.

4. RESEARCH GOAL AND RESEARCH QUESTIONS

In this chapter we will give our research goal and the related research questions, with a small explanation.

4.1 Research Goal

The goal was to make an education- and beginner-focussed diagnostic tool that can check Hoare proofs. The idea of the tool is to allow the user to enter Hoare logic annotated Java like code and receive feedback on the correctness of the Hoare proof. This to prevent the wasting of time by the teacher having to check each proof by hand. And also giving the student a shorter feedback cycle on the work they did.

There are a few difficulties with this. The first is that we need to strike a balance between enough feedback and too much feedback. As explained in the introduction, you want to be useful but not give the entire game away right at the beginning. Related to that, we have our counterexample in SMT and not Hoare logic. We need to find out how to make a counterexample in Hoare logic from the SMT counterexample.

Listing 1. Adding one to a non-negative integer in SMT

```
(1) (declare-fun x () Int)
(2) (declare-fun y () Int)
(3) (assert (>= x 0))
(4) (assert (= y (+ x 1)))
(5) (assert (not (> y 0)))
(6) (check-sat)
```

4.2 Research Questions

The main research question is:

- How do we get useful errors for Hoare Logic?

and the sub-questions related to that are:

- What are useful errors for Hoare logic?

  What kind of response should the system give to the user so that they can figure out what they did wrong with there Hoare logic, while not giving away the answer to the problem and have them learn nothing?

- How do you match the SMT counterexample to Hoare logic?

  Because SMT gives output and input in a slightly different style then Hoare logic, we need to translate between them. Therefore we need to match the counterexample from SMT to the Hoare logic that we are checking.

5. RESEARCH METHOD

In this section we will explain the research method by means of the following elements: how to go from Hoare logic to SMT, how we use the results form SMT, which existing pieces of software we used, the pieces of software we made during the research, how the elements interact with each other, and finally what our input language grammar and syntax is like.

5.1 Hoare logic to an SMT problem

To make a Hoare problem usable for an SMT solver we need to perform two transformations to the Hoare proof. We will use our running example from Algorithm 1. First we need to check if the Hoare triple contains an assignment with the same variable on both sides of the assignment. Our example contains such a variable, namely x. If it contains such a variable, we substitute the variable before the assignment with a new one, and we also do this in the postcondition. In this case we will use y, it does not occur in the rest so it is safe to use. Now we need to negate the postcondition, so that when we find that the SMT problem is satisfiable, we have found a way to make the postcondition false. This is because we wish to prove that $T \rightarrow \varphi$ is valid, that happens when φ is always true. This means that $T \rightarrow \neg\varphi$ must be unsatisfiable. Then we fill in T and φ $T \equiv (x \geq 0) \wedge (y = x + 1)$ and $\varphi \equiv (y > 0)$. If the problem is unsatisfiable, there is no counterexample and we have found that the postcondition is true in all interpretations of the precondition and the program. In Algorithm 1 we show how our SMT program would look like with the running example. In line one and two the integers x and y are initiated, line three the precondition, four the statement, five the post condition, and finally the command to check for satisfiability.

Listing 2. A possible counterexample of the proof checker

```
error: postcondition violated
in file: example on line 3 column 1 until
line 3 column 5
with variable values:
x = 0
new_x = 1
```

5.2 Solving SMT

To go to the solving part of our proof checker, we needed to decide which solver to use for our programme. We used the following criteria for this: feedback of the solver, ease of use, and ease of implementation. The advantage of jSMT-LIB is it is a Java program, so it is platform-independent. Z3 is a Microsoft programme and maybe not as portable. For the rest, both programmes have the same amount and quality of feedback and ease of use, as they are both made to conform to the SMT-LIB standard. As a Java programme jSMT-LIB is also easy to programme with when there is an API, which unfortunately is not available. So a command line interface would have had to be made for both of the programmes. Fortunately someone was already working on that for Z3; this makes Z3 the favourite out of these two because of ease of implementation. A third candidate is Boogie which can also be used as an SMT solver. It has the advantage of being quite powerful and capable of checking an entire programme at once. However it gives far less feedback then the SMT solvers which can return variable values and therefore give better feedback. As we want to give proper feedback, we will use Z3 instead of Boogie or jSMT-LIB.

5.3 SMT Errors to Hoare logic

From the SMT solver, we get the result satisfiable or unsatisfiable. If the result from the SMT solver turns out to be unsatisfiable, we are done and can continue to the next step in the Hoare proof. The SMT problem generated for our running example is unsatisfiable so our Hoare triple is valid. If it is satisfiable, we can show a counterexample. We do this by querying the values of all the variables from the SMT solver and entering them into the postcondition. Then we output to the user the location of the Hoare triple we are checking, together with the variables and there values. To show an example of SMT feedback we will change the zero from line 5 in Listing 1 to one. Then we can add in a line 7, (*get-value*$(x\ y)$), and we get our counterexample $((x\ 0)(y\ 1))$. Using this we make our own counterexample in Listing 2.

5.4 Components

In Figure 1, the system parts of the proof checker are shown. Because the checker is part of a bigger system, the VERCORS project by the FMT group at the University of Twente, some parts have already been made or are someone else's responsibility. These parts have been marked with ovals. Hoare annotated Java like code is given to the parser, which outputs an abstract syntax tree. it is the job of the translator together with the SMT interface to get the Hoare triples from the AST to the SMT solver in the correct form(see section 5.1). Then it is the job of Brain to decide what needs to be done with the response from the SMT solver. It has the choice to stop when a counterexample has been found and give that counterexample to the user, as described in section 5.3 and a counterexample like Listing 2, or to continue to the next Hoare triple if the current Hoare triple is correct.

5.5 Grammar and semantics of the tool

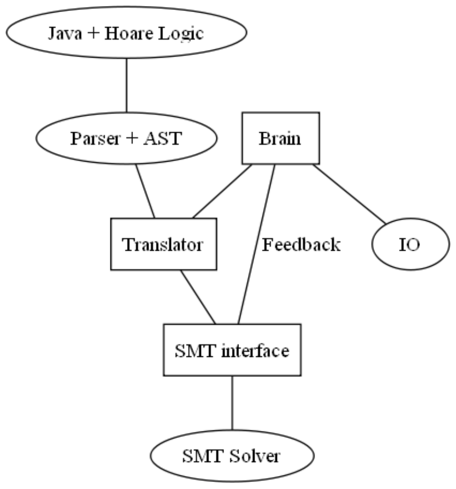

Figure 1. System design. The boxes are elements that were made during the research. The ovals were elements that already existed.

Listing 4. The appearance of a function

```
/*@ requires <expression>
    ensures <expression>
@*/
public <return type> <name>(<parameters>){
     <body>
     return <expression>
}
```

In Listing 3 the grammar of the input language is described. It can handle most binary operators and is capable of several statements. These include: assignment, if, while, and Java like functions. It should be noted that while loops have an invariant explicitly linked with them this means that it does not follow the Hoare rule exactly as it has an explicit precondition and and explicit postcondition. In Listing 4 function is shown to show more explicitly then Listing 3 how it works. Listing 4 shows the language looks allot like Java but it is not. the Grammar does not force it but every statement should have a precondition and a post condition.

6. IMPLEMENTATION & ANALYSIS

In this chapter we will talk about how the programme was implemented and some of the problems that came with the implementation. We start with the high level overview of the implementation. Then the elements that were needed to handle the input. then some problems that were discovered during the implementation. Then how we made counterexamples for faulty Hoare triples. And lastly we will present some testing information.

6.1 Element Interaction

The brain element controls and starts the entire logic checking process. It gets called when the VERCORS tool is started with the option for the Hoare logic checker. During this call it also gets the file that needs to be checked in the form of an Abstract Syntax Tree. Brain then makes an SMTInterface to handle the communication with the SMT solver. Then it makes the Translator to translate the elements from AST format to SMT format. During a check the Translator will send Hoare triples towards the Brain who then makes sure some forbidden characters are removed from the Hoare triple and sends it on to the SMT-Interface. The SMTInterface then returns the result of the

Listing 3. The grammar of the proof checker

```
program ::= function*

function ::= contract type ident (parameter*) block-statement

block-statement ::= (hoare-predicate [statement])* [hoare-predicate return]

statement ::= if expression block-statement [else block-statement]
            |  invariant while expression block-statement
            |  ident = expression
            |  ident = ident (expression*)

contract ::= requires expression ensures expression

hoare-predicate ::= /*{ expression }*/

invariant ::= (//@ loop_invariant expression)+

expression ::= ident
             |  const
             |  unary-operator expression
             |  expression operator expression

parameter ::= type ident

operator ::= + | - | * | > | < | >= | <= | == | || | &&

unary-operator ::= - | !

type ::= boolean | int
```

check on the Hoare triple to the Brain who transforms any counterexamples to something that is useful in Hoare logic.

6.2 Language elements

The language elements will be discussed in the following order:

1. Assignments.
2. If-statements.
3. While loops.
4. Functions.

In $Translator$ we have two main function namely SMT and verify. The function SMT can handle an arbitrary amount of of statements. To the components in translator the SMT function looks like it is the SMT solver, as it is the component that tells this is satisfiable or it is unsatisfiable. A call to verify a system that find the correct function among all the verify functions to handle that type of statement. The verify function takes input in the form of a precondition with some statement and a post condition. It will then tell if that input is part of a valid program or not.

6.2.1 Skip-Statements

In a *skip-statement* the postcondition is directly implicated by the precondition. So we send to the Z3 solver the precondition and the negated post condition as can be seen in Algorithm 2. This is a use of the rule of consequence from Hoare logic were we try to see if P implies Q. The declaration of variables happens in a similar manner as an assignment since the state does not change into a state that cannot be implied by the previous state. This only happens when a value is assigned to these variables. therefore variable declaration is also handled by the skip statement.

Algorithm 2 the skip statement

```
$verify(P, skip, Q)\{$
  $SMT(P, \neg Q)$
$\}$
```

6.2.2 Assignment-Statements

The *assignment-statement* is related to the *skip-statement* as they are some of the few things that sends something to SMT directly. This can be seen in Algorithm 3. Also a small modification was made to the statement here, as mentioned in section 2.3 Z3 has no changing variables, the name S_0 was substituted with S_0'. The same substitution was done in the postcondition Q.

Algorithm 3 the assignment statement

```
$verify(P, assign, Q)\{$
  $S_0 = assign.getVaraible()$
  $e = assign.getExpression()$
  $SMT(P, S_0' == e, \neg Q[S_0 := S_0'])$
$\}$
```

6.2.3 Block-Statements

To handle longer pieces of linear code there is the *Block-Statement*. As can be seen in Algorithm 4 we loop over all elements in the *block-statement*(4). It looks for the *Hoare predicates*(5) to combine the previous statements into Hoare triples(6). If it is a normal element in the body then it is stored in search of the *Hoare predicate* that follows it(10). When the no execution statement exist between two *Hoare predicates* it performs a *skip-statement*(7).

6.2.4 If-Statements

The *if-statement* is not that complex to understand. As can be seen in Algorithm 5 first we get the guards and bodies(2,3,4). Then we check the *If-body* with the guard

Algorithm 4 the Block statement

```
(1)$verify(P, BlockStatement, Q)\{$
(2)  $newP = P$
(3)  $currentBody = skip$
(4)  for $all\, elements\, e\, of\, BlockStatement$
(5)    if $(e\, equals\, HoarePredicate)\{$
(6)      $verify(newP, currentBody, e)$
(7)      $currentBody = skip$
(8)      $newP = e$
(9)    $\}$ else $\{$
(10)      $currentBody = e$
(11)    $\}$
(12)  $\}$
(13)  $verify(newP, currentBody, Q)$
(14)$\}$
```

and P as precondition and Q as postcondition(5). Then we check if there is an *else* part(6) and if it exists we check it with the P and negation of the *if guard* as precondition and Q as postcondition(7). When the else body does not exist we check skip with P and negation of the *if guard* as precondition and Q as postcondition(9).

Algorithm 5 the if statement

```
(1)$verify(P, If, Q)\{$
(2)  $guard = If.getIfGuard()$
(3)  $body = If.getIfBody()$
(4)  $elsebody = If.getElseBody()$
(5)  $verify(P \wedge guard, body, Q)$
(6)  if $(elsebody \neq null)\{$
(7)    $verify(P \wedge \neg guard, elsebody, Q)$
(8)  $\}$ else $\{$
(9)    $verify(P \wedge \neg guard, skip, Q)$
(10)  $\}$
(11)$\}$
```

6.2.5 While-Statements

Then we got while loops the function on how the loops are handled can be seen in Algorithm 6. First the invariants are combined(2) then there is a check to see if the invariant can follow from the precondition(5). Then the body of the loop is checked(6). And finally we check if the invariants together with the negation of the guard produces the postcondition(7).

Algorithm 6 the while statement

```
(1)$verify(P, While, Q)\{$
(2)  $inv = While.getInvariants()$
(3)  $guard = While.getGuard()$
(4)  $body = While.getBody()$
(5)  $verify(P, skip, inv)$
(6)  $verify(inv \wedge guard, body, inv)$
(7)  $verify(inv \wedge \neg guard, skip, Q)$
(8)$\}$
```

6.2.6 Function-Statements

The *function-statement* works like a combination of the *assignment-statement* and some new elements, this can be seen in Algorithm 7. It checks if the precondition of the $method-invocation$, this is how procedures and functions are know to the AST, matches the *requirement* of the contract(9). And it looks to see if the *assurance* of the contract matches the postcondition(10). To make this matching work the variables of the contract need to be substituted with the variables of the pre- and postcondition(9,10). The part where it borrows from assignment is in the postcondition Q where S_0 is replaced by $result$(10).

Algorithm 7 the function statement and function declaration

```
(1)$req :\ P_1(\vec{x})$
(2)$ens :\ Q_1(\vec{x}, result)$
(3)$type\ F(\vec{x});$
(4)
(5)$verify(P, S_0 := F(\vec{e}), Q)\{$
(6)  $contract = F.getContract()$
(7)  $contP = contract.getRequirement()$
(8)  $contQ = contract.getAssurance()$
(9)  $verify(P, skip, contP[\vec{x} := \vec{e}])$
(10)  $verify(contQ[\vec{x} := \vec{e}], skip, Q[S_0 := result])$
(11)$\}$
```

6.3 Implementation problems

During the implementation an problems was found when making the function that handled the *If statement*. Originally reading the assignments and variables was done in a linear fashion. The *Hoare predicates* and the *statements* were added to a list and when two *Hoare predicates* were read the last one was negated and everything was checked. With the help of a visitor pattern, that walked the AST, this was simple and effective and found all the mistakes. Variables when declared are added to a variable list that keeps track of all variables used up to that point in the code.

Then we got to the *if-statements* and we found a problem with the previous implementation for reading the AST. The *if-statement* allowed a path to split and merge. Because of that some *Hoare predicates* had to be checked twice. This was not feasible with the normal visitor pattern because pre and postcondition had to be given as arguments for the visit method. And since the visitor pattern of Java does not contain this it had to be made. This meant that the entire programme that we had built up to that moment had to be changed.

6.4 Making counterexamples

We will show how the counter example is made by using an example that contains an error. In Listing 5 there is a small error in the *Hoare predicate* on line 13 a minus was forgotten so it says y = x instead of y = -x. This is reported by the SMT checker who will report it found a violation and with what values the violation was possible. This is then transformed to the output from Algorithm 6. The filling in of the code with the found variables is a bit tricky. This is because the piece of code that reads the file and the piece of code that makes the counter example are not directly linked. The filling in of the counterexample has therefore been left to future work. The variables that are not used in the error state are shown like the variable result this is something for future work. The _y variable is the new y. This is the solution for the no variables in Z3 problem, this solution is applied even when the problem would not occur.

6.5 Testing

To test our system, we used exercises from several courses on formal methods. We use the answers from these exercises and see if our logic checker agrees, and also if it finds errors we introduce on purpose. Testing was done on a computer with: AMD Phenom II X6 1090T Processor at 3.2GHz with 8GB of RAM and running Windows 7

Listing 5. error_example.java

```
(1)/*@ requires true;
(2)    ensures \result >= 0;
(3)@*/
(4)public static int M(int x){
(5)      /*{true}*/
(6)      if(x >= 0){
(7)               /*{x >= 0}*/
(8)               y = x;
(9)               /*{y >= 0 && y = x}*/
(10)     }else{
(11)              /*{x < 0}*/
(12)              y =-x;
(13)              /*{y > 0 && y = x}*/
(14)     }
(15      )/*{y >= 0}*/
(16)     return y;
(17)}
```

Listing 6. error output

```
The postcondition at:
file error_example.java from line 13
column 25 until line 13 column 50
was violated with the following values:
result = 0
x = -4
y = 0
__y = -4
```

64bit version. One of the things that was noticed during the testing that we did at the end of each phase of construction was that Z3 calls take a while. Some calls of Z3 took up to three seconds to return with a result. In the appendix some other test cases with results are presented.

7. CONCLUSIONS

In this chapter we will look at the results of this research in relation to the research questions. We also give some ideas for future work that give expansions to the functionality of the programme.

7.1 Research answers

We asked ourselves how do we get useful errors for Hoare problems. And we had three sub questions with that. The first what are useful errors for Hoare Logic. It turns out that it is useful to give the values of variables and the context where these values were found. Any form of parsing information is just to much information and not relevant. It also leads to cluttering of the command line by to much information that we don't need.

The second one was how do you match SMT counter examples to Hoare logic? Matching the counter examples to Hoare logic was not that difficult as variables can be freely moved between Z3 and Hoare logic. So the matching was done by taking the variable values form SMT and the lines of code where the problem occurred this was handed over to the user. And that is how the counter examples were matched.

So the main question of how we get useful errors for Hoare logic turned out to be. Transform the problem into and SMT problem check that for errors. And place the values of any violation back into the source code and show this to the user.

7.2 Future work

Here we have a few ideas for future expansion to the logic checker. In short they are more language elements so that our function can handle them and maybe even more input languages. They are explained in more detail in the following sections.

7.2.1 New Language Elements

One of the things that could be added to the checker in future work is the addition of several extra language elements. Elements like global variables, Arrays, For loops, Do While loops, Universal Quantifier for arrays, Enumerators, and Object Orientation handling. These would allow for more functions to be used and give a little more freedom in the use of the tool. These are common language elements and being able to check them and let people learn how to properly verify them is useful. The implementation of procedures and functions was not done during the research so this is something that needs to be done along with the suggested new language elements.

7.2.2 Array-Statements

Unfortunately arrays statements were not implemented. Despite that the SMT standard gives room for array types. To reason about them in a decent way we require for-all statements, Universal Quantification, and related statements that reason over groups of data. And that was not on the planning to be implemented. This made it so that arrays would take to much time to implement.

7.2.3 Missing Preconditions

People can sometimes forget that Hoare logic only reasons about one Hoare triple at a time. This together with the Consequence rule, leads to forgotten preconditions, where something that was proved earlier is not copied over to the procedure that needs it. For example in Algorithm 8 we forgot to copy the $y \geq 0$ to a later precondition(5). This is allowed by the *Consequence rule* because $\{y \geq 0\}$ is implied by $\{x > 0 \wedge y \geq 0\}$. And several long manipulation before that one it was decided that y wasn't needed any more. This would then lead to an error in the last Hoare triple because there is no precondition to constrain y and therefore y can be any value including one that violates the postcondition.

Algorithm 8 Forgotten precondition example

```
(1) $\{x \geq 0 \wedge y \geq 0\}$
(2) $x = x + 1;$
(3) $\{x > 0 \wedge y \geq 0\}$
(4) ...//some long manipulation of x
(5) $\{x > 0\}$
(6) $y = y + 1;$
(7) $\{x > 0 \wedge y > 0\}$
```

This problem can be handled by our program without spoiling the learning process because it a simple oversight and not a big error. These were also not implemented because of time constraints and finding the correct bit of a precondition to use turned out to be very difficult. The tool would have to look at every precondition that came before this one and attempt to find a bit of the older preconditions that would restrict enough to make the Hoare triple valid again. This could have quite a severe effect on the performance of our tool. As there could be easily ten calls of Z3 before the missing bit was found. These Z3 calls take quite some time as was discovered during testing and implementation. Having to wait around thirty seconds if the tool needed to check ten combinations of preconditions is quite long. This is such an effect on the performance that it is notable to mention.

Listing 7. An ideal counterexample of the proof checker

```
error: triple not valid
in file: example on line 1 column 1 until
line 3 column 5
{x >= 0}
x = x + 1;
{x > 1}
counter example:
{0 >= 0}
1 = 0 + 1;
{1 > 1}
```

7.2.4 Formatting counterexamples

The counter example has some variables that are not used during some of the calls of Z3. These will have to be removed to improve readability of the return. The other problem is that only the line of code is given and not the piece of code that is in violation. These line of code should also be returned to the user with the variables filled in to the violation values. This ideal can be seen in Listing 7. An other extension to the counter example would be the direct pointing out the part of a state that is in violation. This by pulling it apart at the conjunctions and seeing if the elements on there own are in violation.

7.2.5 Extra Functionality

Other elements that might be interesting in the future is the checking of code in other languages than our Java like code. As not all code is written in Java and having options to check other models like scripting languages would add allot of versatility. A second thing that would be interesting in the language department is the adding of different checkers. An example of this would be when jSMT-LIB releases an API this has the potential to reduce the complexity of the SMTInterface.

8. REFERENCES

[1] K. R. Apt. Ten years of hoare's logic: A survey part i. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483, Oct. 1981.

[2] M. Barnett, B.-Y. Chang, R. DeLine, B. Jacobs, and K. Leino. Boogie: A modular reusable verifier for object-oriented programs. In F. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer Berlin / Heidelberg, 2006.

[3] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, 2010.

[4] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach.* LNCS 4334. Springer-Verlag, 2007.

[5] R. Bornat and B. Sufrin. Animating formal proof at the surface: The jape proof calculator. *The Computer Journal*, 42(3):177–192, 1999.

[6] D. R. Cok. The smt-libv2 language and tools: A tutorial. Technical report, GrammaTech, Inc., February 2011.

[7] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969.

[8] K. R. M. Leino. This is boogie 2. Technical Report KRML 178, Microsoft Research, Redmond, WA, USA.

Listing 10. loop_error.java

```
(1)/*@ requires n > 0 ;
(2)     ensures \result == n*n;
(3)@*/
(4)public static int f_err(int n){
(5)     int res;
(6)     /*{n > 0}*/
(7)     int i;
(8)     /*{n > 0}*/
(9)     res = 0;
(10)   /*{res == 0 && n > 0}*/
(11)   i = 0;
(12)   /*{ res == 0 && n > 0 && i == 0 }*/
(13)   //@ loop_invariant res == i*n;
(14)   //@ loop_invariant i < n;
(15)   //@ loop_invariant n > 0;
(16)   while(i < n) {
(17)       res = res + n;
(18)       /*{res == (i+1)*n && i < n &&
              n > 0}*/
(19)       i = i+1;
(20)   }
(21)   /*{res == n*n && i == n && n > 0}*/
(22)   return res;
(23)}
```

Listing 11. loop_error output

```
The postcondition at:
file loop\_error.java from line 13 column
36 until line 13 column 45
was violated with the following values:
result = 0
n = 2
res = 4
i = 1
__res = 0
__i = 2
```

Listing 8. loop_example.java

```
(1)/*@ requires n > 0 ;
(2)     ensures \result == n*n;
(3)@*/
(4)public static int f_ok(int n){
(5)     int res;
(6)     /*{n > 0}*/
(7)     int i;
(8)     /*{n > 0}*/
(9)     res = 0;
(10)   /*{res == 0 && n > 0}*/
(11)   i = 0;
(12)   /*{ res == 0 && n > 0 && i == 0 }*/
(13)   //@ loop_invariant res == i*n;
(14)   //@ loop_invariant i <= n;
(15)   //@ loop_invariant n > 0;
(16)   while(i < n) {
(17)       res = res + n;
(18)       /*{res == (i+1)*n && i < n &&
              n > 0}*/
(19)       i = i+1;
(20)   }
(21)   /*{res == n*n && i == n && n > 0}*/
(22)   return res;
(23)}
```

Listing 9. loop_example output

```

No errors found
```

APPENDIX

In Listing 8 we show a simple loop statement that calculates n^2.
Since this is a good example of valid code the output, as seen in Listing 9, is trivial and uninteresting.
In Listing 10 a small mistake was made in line 14. instead of having i <= n we wrote i < n.
This error is a bit harder to read from the return. It speaks about line 13 only but this is part of the loop invariant and it test all elements of that at once. This is a limitation on the system. The Hoare triple that is checked here is the following.
P ≡ /*{ res == (i+1)*n && i < n && n > 0 }*/
S ≡ i = i+1;
Q ≡ //@ loop_invariant res == i*n; ∧ //@ loop_invariant i < n; ∧ //@ loop_invariant n > 0;