CS335 Final Report: ProofEngine

Pragati Agrawal (220779), Dhruv Gupta (220361)

20 April 2025

Introduction

ProofEngine is a deductive verification tool that uses Z3 SMTLIB solver to verify the correctness of programs. It is built upon the Chiron Framework, a Python-based framework for teaching program analysis and verification. It takes an analysis source code as an input, and generates an SMTLIB code, that can be given to a Z3 solver to check the satisfiability of the constraints.

Use Cases

- **Proving program correctness:** We can use ProofEngine to prove that a program is correct or that a program follows certain properties by inserting suitable analysis code.
- **Path Feasibility:** We can check if a certain execution path in the program is feasible.
- **Finding bugs:** If a program is incorrect, we can find a counterexample to prove that the program is incorrect. This makes debugging a lot easier.
- **Proving two programs are equivalent :** We can write the analysis code suitably to prove that two programs always behave the same. See *./examples/leapyear.tl* for example testcase.
- **Proving correctness for unbounded loops :** Since ProofEngine utilizes an `invariant` for loop satisfiability, we can do deductive verification and prove correctness for unbounded loops, i.e.,conditions like P is true ∀ n can be proven.

Deductive Verification

Deductive verification is a formal method used to prove the correctness of programs. It involves the use of logical assertions in the form of preconditions, postconditions, and invariants for loops to establish that a program satisfies its specifications. Unlike testing, which can only check a finite set of inputs, deductive verification provides mathematical guarantees of correctness by covering all execution paths symbolically. Typically the structure of analysis code would look like this:

```
assmue(preconditions)                --> Precondition(Pre)
while(loop condition){               --> Guard
  Invariant(invariant conditions)    --> Invariant(Inv)
  loop body code                     --> Body
  ...
}
assert(postconditions)               --> Postcondition(Post)
```

To prove the correctness of a program of the above structure, we would need to verify the following three conditions individually:

- **Initialization Condition:**
$$Pre(x) \implies Inv(x)$$
- **Loop Condition:**
$$Inv(x) \wedge Guard(x) \wedge Body(x, y) \implies Inv(y)$$
- **Final Condition:**
$$Inv(y) \wedge \neg Guard(y) \implies Post(y)$$

If some program does not contain loop, then the structure of analysis code would be:

```
assmue(preconditions)                    --> Precondition(Pre)
code body                                --> Body
...
assert(postconditions)                   --> Postcondition(Post)
```

And we would simply check

$$Pre(x) \wedge Body(x, y) \implies Post(y)$$

Now, to check correctness of any program, we check if the negation of any of these conditions is satisfiable, i.e., if we can get any inputs on which the condition fails. If there is none, this means that our code is correct.

Challenges

- **Finding Invariant:** Figuring out the correct invariant for any loop is an undecidable problem, making it implicitly difficult. So, for checking the correctness of our codes, and for writing test cases, we had to figure out the correct invariant for the code we intend to write.
- **Handling prefix notation:** The SMTLIB code needs to be in prefix notation, but Chiron source code is in infix notation. This requires a conversion from infix to prefix notation and naturally the prefix code becomes harder to read and debug.
- **Renaming Variables:** The SMTLIB code generated from the analysis code needs to be in SSA-like form. This requires us to rename all variables in the code without requiring to generate complete the SSA form.
- **Variables used in the Invariant:** The conditions required to prove the satisfiability of a program involve *Invariant_in* (`Inv(x)`) and *Invariant_out* (`Inv(y)`). For this we need to correctly figure out which version of each variable to use in both of these.
- **Handling branches:** We needed to find a way to handle if-else blocks without requiring ϕ nodes in the CFG or going to full SSA form.
- **Turtle Commands:** The turtle commands are not in the form of arithmetic expressions. We needed to come up with appropriate arithmetic expressions to encode the turtle's movements to generate the equivalent arithmetic expressions.

Code Structure and Implementation Details

Modifications and Additions

We have made modifications and additions only in the /ChironCore directory.

- We have added a new command line flag `-smt/--smtlib` to the `chiron.py` file. If this flag is provided, the program expects an analysis source code (`.tl` file) which follows the format as described in the previous section.
- We have introduced a new command - `AnalysisCommand` in the language which can be of three types - `assume`, `invariant`, `assert`. For this we changed the production rules in `turtparse/tlang.g4` to include these statements. Also, we created a new **AnalysisCommand** class in `ChironAST/ChironAST.py` to handle these commands.
- We have created a new ast builder, `astGenPassSMTLIB` in `/ChironAST/builder.py`. It inherits from the original builder `astGenPass` but handles turtle commands - forward, backward and IfConditional differently.
- We have also added the `Mod`(%) operator in the language since it was needed for compilation of turtle commands.
- All other modules are added in the `ProofEngine` directory.

Generating SMTLIB code

SMTLIB code uses prefix notation whereas in general we write the expressions in infix notation. Hence each expression in each statement needs to be converted into prefix notation. This is done by `ProofEngine/Infix_To_Prefix.py` module.

- **Handling Assign Statements** : Each assign statement is converted to an equivalent **assert** statement in SMTLIB.

$$var = expr \quad \Rightarrow \quad (assert\ (=\ var\ \texttt{Infix\_To\_Prefix}(expr)))$$

For handling turtle commands, we need to maintain the state of the turtle. We represent the state of the turtle using a **4-tuple**, $(\_\_turtleX, \_\_turtleY, \_\_turtleZ, \_\_turtleW)$. The first two represent the x,y coordinates of the turtle. Last one represents the angle in which the turtle is pointing. Third one represents the *penstate*, i.e., $penup(1)$ or $pendown(0)$.

- **Handling `goto` command** : Each Goto Command is coverted to two assignment statements.

$$goto(exprA, exprB) \Rightarrow \_\_turtleX = exprA, \ \ \_\_turtleY = exprB$$

- **Handling `left/right` statements** : Each left or right statement if converted to one assignment statements.

$$left\ expr \Rightarrow \_\_turtleW = (\_\_turtleW + expr)\%360$$

$$right\ expr \Rightarrow \_\_turtleW = (\_\_turtleW - expr)\%360$$

- **Handling `forward/backward` statements** : Each forward or backward statement is converted to two assignment statements - one each to update $\_\_turtleX$ and $\_\_turtleY$ respectively.

  $forward\ A \Rightarrow \_\_turtleX = \_\_turtleX + A * f(\_\_turtleW),\ \ \_\_turtleY = \_\_turtleY + A * g(\_\_turtleW)$

  $backward\ A \Rightarrow \_\_turtleX = \_\_turtleX - A * f(\_\_turtleW),\ \ \_\_turtleY = \_\_turtleY - A * g(\_\_turtleW)$

  Here f and g are some basic arithmetic functions which evaluate as follows -

$$f(x) = \begin{cases} 0 & \text{if } x = 90 \text{ or } 270 \\ 1 & \text{if } x = 0 \\ -1 & \text{if } x = 180 \end{cases} \qquad g(x) = \begin{cases} 0 & \text{if } x = 0 \text{ or } 180 \\ 1 & \text{if } x = 90 \\ -1 & \text{if } x = 270 \end{cases}$$

  Please refer to `ProofEngine/TurtleCommandsCompiler` for exact expressions. Also, there is one important detail, since `forward`,`backward` and `goto` commands are compiled to two statements, we add a `NoOp` in the AST with each of these commands when we construct it using `astGenPassSMTLIB` to reserve a spot for the extra statement.

- **Variable Renaming** : To capture the code flow, we need to rename the variables in SSA-like format, i.e., we have to convert our code into SSA-form except for the $\phi-$nodes. This can be easily done for a straight line code by maintaing a map of variables to their current live instance.

All the above points are sufficient to handle straight line code. Code with branches (if-else blocks) is handled as follows -

- **Variable Renaming for branches** : The variable renaming now can be done for each basic block. Each variable can be first renamed as $var \Rightarrow var\_\{bb\ id\}$. For transferring the value from the current basic block to its successors in CFG, we add assignment statements for each variable used in any of its successors - $var\_\{succ\ id\} = var\_\{bb\ id\}$ at the end of the basic block. Then we can rename each basic block the same way as we would rename a straight line code. Also note that, we start adding the statements in the reverse order of code flow, i.e., we start at the end node of the cfg and go upwards upto the start node. This ensures that in the start all variables used in the code are initialised and hence can be included later in the precondition. Hence, each basic block will assign values to all the variables used later in the code.

- **Handling If-Else blocks** : Each block of the form if ($condition$) $then$ $S1$ $else$ $S2$, (where $S1$ and $S2$ are list of assign statements) can be compiled to an equivalent SMTLIB `assert` statement as -

$$(assert\ (or\ (and\ condition\ S1)\ (and\ (not\ condition)\ S2)))$$

- **Handling If blocks** : If there is only a *then* part and no else part, then we insert an empty `else` node in the CFG. We handle this in the `astGenPassSMTLIB` where we handle IfConditionals as IfElseConditionals, with an empty list for else part.

- **Handling Loops** : If there is a loop in the code, its verification is done using the invariant as discussed in the previous sections. Hence, we extract the assume/precondition ($Pre(x)$), invariant_in ($Inv(x)$), invariant_out ($Inv(y)$), loop condition ($Guard(x)$), loop false condition ($\neg Guard(y)$), assert/postcondition ($Post(y)$) from the CFG. To ensure that correct variables

are used in invariant_out and loop false condition, we insert these in the "assert" node at the end of the CFG before renaming. Similarly, we ensure that same variables are used for loop condition and invariant_in. All the extra statements added in the start, i.e., in the "assume" node are included in the precondition, since they set the initial state of variables used by the loop body. Once all these are extracted, we can generate SMTLIB code for the loop body in the same way as we did for code without loops.

Code Flow

- **Generating modified IR and CFG** : We first build the AST from the parse tree using our modified builder `astGenPassSMTLIB`. Then we use the `ProofEngine/TurtleCommandsCompiler.py` to compile the turtle commands and generate a modified IR. We then build the CFG using `cfg/cfgBuilder.py`.
- **Checking the format of CFG** : We check if the cfg of format is as expected, for eg., the `assume` statement should be in the start, the `assert` statement should be at the end, etc.
- **Appending block ids to variables** : We extract the variables used in each basic block and rename them by appending block ids. In this step we also rename the variables :TURTLEX, :TURTLEY, :TURTLEPEN, :TURTLEANGLE, :REPCOUNTER if these are used by the user. These variables represent the turtle states $\_\_turtleX$, $\_\_turtleY$, $\_\_turtleZ$, $\_\_turtleW$ and the loop counter $\_\_rep\_counter\_1$ respectively.
- **Inserting conditions appropriately** : We insert invariant_out and loop false condition with the "assert" node and rename the variables with the corresponding block id.
- **Rename all the variables** : We rename each new instance of each variable by appending a number.
- **Add statements at the end of each block** : We process all the basic blocks in reverse order and add assignment statements for each variable used in any of its successors. Note that we assign only the first instance of the successor's variable, rest instances are created in the successor block itself.
- **Extract all conditions** : We extract all the conditions (precondition, invariant, postcondition, etc.). Each condition is represented by a list of statements.
- **Handle loop body/code body** : We traverse the loop body and convert each basic block to a tuple *("assign", instrList)* if it has only assign statements. If a basic block has a condition, then we separate that condition and compile the rest of the basic block same as before. Then we take that condition and create a tuple *("if-else", condition, then_instrList, else_instrList)*, where the *then_instrList* and *else_instrList* are obtained by processing the successors of the condition's basic block.
- **Convert Infix to Prefix** : We process all the statements in all the instruction lists and conditions obtained from previous step, and convert each of them from infix to prefix format.
- **Combine all statements** : We combine all the statements in the respective instruction lists to obtain the precondition, loop body/code body, postcondition, invariant, loop condition, etc. to generate a single SMTLIB statement for each of these.

- **Final SMTLIB code** : We use all these individual statement to write all the satisfiability check conditions (by negating all the conditions as discussed in previous sections).
- **Add variable declarations** : We extract all the variables used in the code generated and add all the variable declarations in the start of the code. We also add the **check-sat** and **get-model** conditions to check satisfiability and get the values if satisfiable.
- **Invoke the Z3 Solver** : The generated SMTLIB code is given to a python's z3 solver to check for satisfiability.
- **Parsing the output of Z3 Solver** : The output generated by Z3 solver is parsed and inspected to provide meaningful results to the user, displaying which condition verifications failed (if any) and display all the variable:value pairs in a proper format.

Description of files

We have added a new directory `ProofEngine` in the `ChironCore` directory. The `ProofEngine` directory contains the following files:

- `Infix_To_Prefix.py` : Constructs an AST for the given infix expression. The preorder traversal of this AST gives us the prefix notation.
- `TurtleCommandsCompiler.py` : Compiles all the turtle commands into assign statements in the IR.
- `TraverseCFG.py` : Performs all the processing on the CFG - checking CFG format, variable renaming, adding instructions, extracting conditions, etc.
- `Smtlib_Helper.py` : Contains all the helper functions needed to generate IR and CFG, and then generating the final SMTLIB code using the output of `TraverseCFG.py`.
- `Z3Integration.py` : Contains the function to invoke the z3 solver and process the output of z3 solver to display meaningful results.

Assumptions

- The code contains only one loop (single repeat statement). Hence nested loops and more than one loops are not allowed.
- The code does not contain nested if-else blocks. Note that any nested if-else block can be converted to an equivalent set of non-nested if else blocks. Hence this does not restrict the set of programs which can be written.
- We assume that the user will ensure that angle for `left/right` is a multiple of 90 or in other words, the turtle will move only at right angles.
- The code does not use the variables :TURTLEX, :TURTLEY, :TURTLEANGLE, :TURTLEPEN, :REPCOUNTER for other purposes, since they have a specific meaning and these variables are renamed internally. These are exposed to the user so that he/she can write conditions in terms of these variables for analysis code.

How to run

Open the terminal in the home directory of the git repo. Run the following commands:

```
cd ChironCore
python3 chiron.py -smt <path-to-src-program-file>
```

or one can use the `--smtlib` flag.

```
cd ChironCore
python3 chiron.py --smtlib <path-to-src-program-file>
```

Explanation of Testcases

- **cube.tl**- It calculates $(a-b)^3$ using two ways. First by multiplying with itself 3 times, i.e., $(a-b)*(a-b)*(a-b)$. Second by using the algebraic identity $(a-b)^3 = a^3 - 3a^2b + 3ab^2 - b^3$
- **cube_buggy.tl**- It also checks the cube identity, but it uses the wrong identity $(a-b)^3 = a^3 - 3a^2b - 3ab + b^3$(INCORRECT). This code's output gives the set of values on which the incorrect identity fails.
- **sort.tl**- It sorts three numbers x,y,z in ascending order and checks if these satisfy the condition `x <= y <= z`.
- **sort_buggy.tl**- It also sorts three numbers x,y,z in ascending order but it has a bug in it. The bug is that if the inputs are in descending order (i.e. if $x > y > z$) then the code returns the order (y, z, x) which is incorrect. This code's output indeed gives the set of values in descending order.
- **signum.tl**- It performs the signum function, i.e. returns the sign of a number.
- **signum_buggy.tl**- It also performs the signum function, but it has a bug, i.e. it assigns sign -1 to input 0. The code's Z3 ouput indeed gives this set of values.
- **leapyear.tl**- It compares two methods of checking whether a year is a leap year. The second method is a more intuitive way, written using only if-else statements and its correctness is ensured by the definition of leap year. The first method is rather non-intuitive and it checks leap year only using arithmetic assignment statements. We can prove that the two methods are equivalent.
- **arithmetic_progression.tl**- It computes the sum of first `n` terms of an AP with first term `a` and common difference `d` using a loop. It checks this value against the value obtained by the formula of sum of first `n` numbers of an AP: $s = (n*(2*a+(n-1)*d))/2$
- **home.tl**- This program proves that if the turtle starts in the inner box (home), it will never step outside the outer box (its playground). It can start from anywhere within the square box of (-150,-150) to (150,150). It will move in a spiral outward manner, for a total of 5 rounds. Anytime it crosses the outer boundary square of (-200,-200) to (200,200), we set the flag `outgone` to 1, and the postcondition fails. Depending on the starting coordinates, the code is satisfiable. See *home_src.tl* for source code.

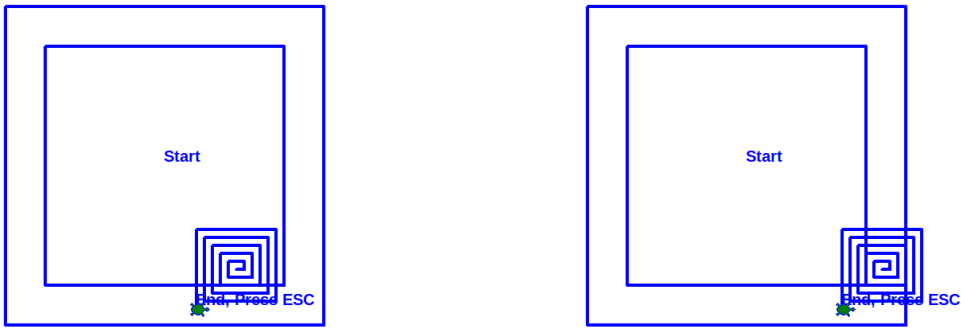

Figure 1: Turtle remains in v/s out of the playground

- **spiral.tl**- This program proves that if the turtle starts from (0,0) and moves in a spiral outward manner, then $\forall n$ (`n` is the number of steps), it will always remain in the square box (-n-1, -n-1) to (n+1), (n+1). See *spiral_src.tl* for source code.

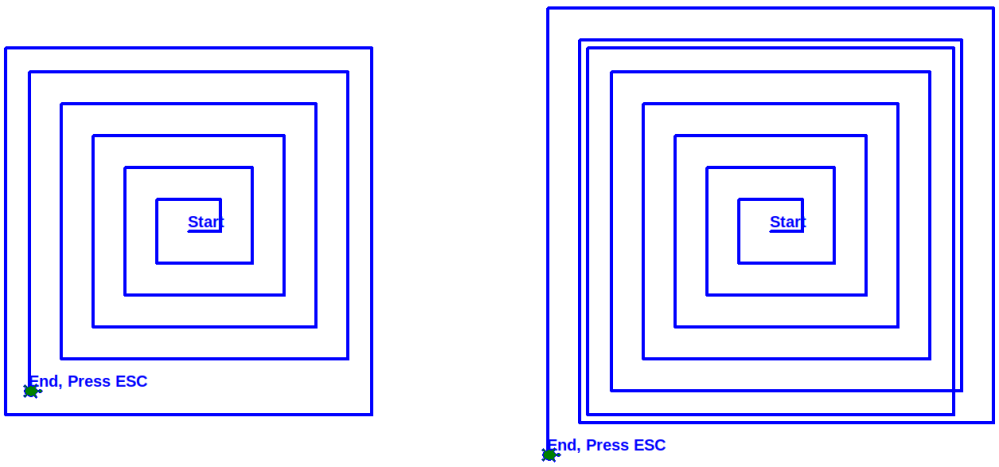

Figure 2: Turtle remains in v/s out of the outer box

- **river.tl**- This program checks that if the turtle moves in a certain manner, as mentioned in

the code, it will never enter the river, flowing on the line $y = x$, and hence it will be safe from crocodiles! See *river_src.tl* for source code.

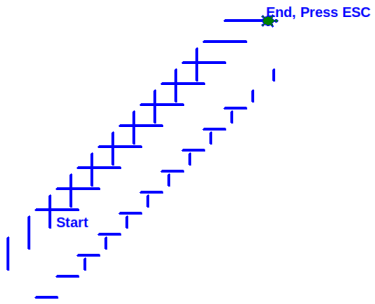

Figure 3: Turtle never goes into the river

- **park.tl**- This program checks that if the turtle starts anywhere from (-100,-100) to (100,100) and moves exactly 10 steps in a staircase right-up manner, it will surely reach the park, spread from (400,400) to (600,600). See *park_src.tl* for source code.

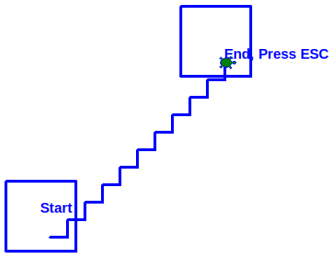

Figure 4: Turtle reaches the park in 10 steps starting from home

Kindly see the *run_images* folder inside the *examples* directory to see screenshot of the turtle execution for each of the test cases also having source codes along with the analysis codes.

What we learned?

This project gave us a good opportunity to learn about program verification. We also learnt about how to use z3 solver and how we can use it to prove interesting things. We learnt about deductive verification and how `forall` type of condtions can be proved. Other than that we got the experience to play around with various data structures used in compilers like the IR and CFG, and some idea about how to use an automated parser. We also learnt how an AST can be built and some cool things which we can do with it, like converting infix to prefix notation. Overall it was a great experience and we learnt a lot from it.

Scope for Improvement

- **Handling nested if-else blocks:** The current implementation does not handle nested if-else blocks. Handling these would make it easier to write source code.
- **Handling move statements:** The current implementation does not handle move statements for angles other than 90 degrees. Handling these would allow us to test programs with more interesting movement patterns of the turtle.

Acknowledgements

We would like to extend our heartfelt gratitude to our professor, Dr. Subhajit Roy, for his constant guidance and support throughout and also for suggesting us this project. We would also like to thank our TA Vikas Kushwaha Sir for his valuable feedback and suggestions. We would also like to thank Sumit Sir for his guidance about the Chiron Framework.

References

- AST Visitor Template: https://gist.github.com/jtpio/cb30bca7abeceae0234c9ef43eec28b4
- AST Documentation: https://docs.python.org/3/library/ast.htm#last.NodeVisitor.visit
- SMTLIB Documentation: https://smt-lib.org/
- NetworkX Documentation: https://networkx.org/documentation/stable/
- Z3 Online Demonstration: https://compsys-tools.ens-lyon.fr/z3/
- Z3 Playground: https://jfmc.github.io/z3-play/
- Python Regular Expressions: https://docs.python.org/3/library/re.html
- Python Official Website: https://www.python.org/