EECS 4302
Final Project

Winter 2020

A Compiler for the Verification of a Programming Language: User Manual

Last Name	First Name	Student Number	PRISM Login

Table of Contents

1 Input Programming Language

1.1 Overall Structure of a Program

```
class Account { /* Specification: Class Declaration */
  /* Section: Variable Declarations */
  declare {
    int old_amount;             /* Specification: Int Initialization */
    int amount = 0;
    int tmp;
    string name;                /* Specification: String Initialization */
    bool student = false;       /* Specification: Boolean Initialization */
    int[50] recentTransactions; /* Specification: Array Initialization */
  }

 /* Section: Class Invariants/Preconditions */
  assume {
    amount >= 0;
 }

 /* Section: Functions */
  func int increase(int x, int y) {
    require {                  /* Specification: Function Precondition */
       x > 0;
       y > 0;
    }
    old_amount = amount;       /* Specification: Function Body */
    amount = amount + x;
    amount = amount + y;

 /* Section: Loops */
    loop {
      from {                  /* Specification: Loop Start Condition/Initialization */
        tmp = 1;
        amount = 5;
        old_amount = amount;
      }
      until {                 /* Specification: Loop Exit Condition */
        tmp = 5;
      }
      invariant {             /* Specification: Loop Invariant */
        tmp > 0;
        amount >= old_amount;
      }
      do {                    /* Specification: Loop Body */
        old_amount = amount;
        amount = amount * 2;
        tmp = tmp + 1;
      }
      variant {          /* Specification: Loop Variant */
        5 - tmp;
      }
    }
    // loop end          /* Specification: Comments */

    if (amount > 50) {   /* Specification: If-Else Statement */
      amount = amount - 50;
    } else if (amount <= 50) {
      amount = amount + 50;
    } else {
      amount = amount - 1;
    }
    /* Section: Switch Statement */
    switch (amount) {
      case: 4 {         /* Specification: Switch Cases */
        student = true;
      }
      default: {        /* Specification: Default Case */
        student = false;
      }
    }

    ensure {            /* Specification: Function Postcondition */
      amount == (old_amount + x + y);
    }
  }
  // function end

  func void increaseAndCheck() {
    increase(20, 20);  /* Specification: Function Call */
    ensure {
      amount >= 0;
    }
  }

  assert {             /* Specification: Class Assertion */
    amount > 0;
  }
} // class end
```

Figure 1: Structure of Input Program

Figure 1 outlines what a typical input program looks like. The compiler supports many programming features. It supports variable initialization, assumptions (predicates), functions, loops, and assertions (class-level assertions). For variables, the types integer, string and boolean are supported, the values ‘true’ and ‘false’ for boolean, specifically. Every block of the program begins and ends with braces, e.g. `require { … }`. Each program must contain a class block and must contain a name for the class, otherwise the program will be empty. Note that it is possible to have an empty class, however, that is not really useful. Additionally, note that the above example is just one way of typing the program.

1.2 How a Specification is Written

1. 1.2.1 Specification: Preconditions (Function)

Preconditions are supported with the keyword `require` in functions. This specification appears in the function block in the program and is enclosed with braces. They act as preconditions for the function defined. It only accepts boolean expressions. This specification does not support the notion of pre-state, simply because it is a precondition and there is no reassignment of variables that happen in this block.

Example code snippet:

```
// function signature
require {
      x > 0; // assume x is an int variable
      y > 0 && z > 0 && y > z;
}
```

2. 1.2.2 Specification: Postconditions (Function)

Postconditions are supported with the keyword `ensure` in functions. Just like preconditions, this specification also appears in the function block and is also enclosed with braces. This specification *can* support pre-state but only with the help of extra variables. For example, to check the post state of a variable that had its value changed, we need to have another variable, *`old_variable`*, that will be checked against the new value.

Example code snippet:

```
// function signature
ensure {
      z == x + y; // e.g., increase function
      old_z < z; // assume old_z is initialized earlier
}
```

3. 1.2.3 Specification: Assumptions (Class)

Assumptions are supported with the keyword `assume` in classes. This specification appears in the class block and is also enclosed with braces. These assumptions act as predicates for the class. They are facts used to assist Alloy when trying to find counterexamples. This specification does not support the notion of pre-state since these are just predicates made in the class. There are no reassignments since it only accepts boolean expressions and therefore, we do not need to support pre-state in this specification.

Example code snippet:

```
class Person {
      declare {
            string name;
      }
      assume {
            string != “”;
      }
}
```

This snippet highlights the fact that each person is assumed to have a name, in other words, no person can have no name.

4. 1.2.4 Specification: Assertions (Class)

Assertions are supported with the keyword `assert` in classes. This specification appears in the class block and is enclosed with braces as well. These assertions are used as class invariants. This specification can support the notion of pre-state with the help of additional old variables. Assertions accept only boolean expressions.

Example code snippet:

```
class Person {
      declare {
            int numOfFriends;
      }
      assert {
            numOfFriends >= 0;
      }
}
```

5. 1.2.5 Specification: Classes

Classes are supported using the keyword `class` followed by the name of the class. Each class is enclosed by braces. This specification occurs in the beginning, the starting block for each class/module, they have to start with the keyword `class`. For example,

```
class Person {
      declare {
            string name;
      }
}
```

The above snippet shows how a Person's class can be constructed.

6. 1.2.6 Specification: Loop Invariants (Loops)

loop invariants occur within the body of a loop and are represented with the keyword **invariant**, loop invariants are boolean expressions that are assumed to hold true at the beginning of each loop iteration, loop invariants do not implement the old keyword as it is not seen as a necessary feature (users can store old values in temporary variables). In our hoare triple based compilation method, the invariant is logically implied by the statements that follow the loop conjuncted with the initialization block. the loop invariant is also used in conjunction with the exit condition of the loop to imply the post condition of the loop

example of a loop invariant:

```
invariant {
      i <= z;
      i > 0;
      result == x || result == y;
}
```

this loop invariant is extracted from our fibonacci example, here the invariant is verifying that the result must be equal to either x or y (the 2 variables used to store the numbers that will calculate the next fibonacci number, can be thought of as f_i-1 and f_i-2 for the upcoming iteration), the loop counter i must be greater than 0, and i must be less then or equal to z, (where z is the zth fibonacci number that is to be calculated and returned as the final result of the function)

7. 1.2.7 Specification: Loop Conditionals (Loops)

The Loop Conditional, represented by the keyword **until,** occurs within the body of each loop, Conditionals take the form of a series of boolean expressions, whose values must all evaluate to true for the loop to exit. The until block does not support an old keyword as it would not have much use in verifying the exit condition of a loop. The loop conditional along with the invariant are used to verify partial completeness of the loop in our Hoare triple compilation method. An example of an until block being used in action is shown below.

```
until {
	i >= z;
}
```

This is from our fibonacci example, where i can be considered as the loop counter, and the resulting fibonacci number we wish to return is the zth one, there is only one condition here, but its valid to have multiple expressions in the until block and each must evaluate to true for the loop to exit

8. 1.2.8 Specification: Loop Variant (Loops):

The Loop Variant is represented in our program by the keyword variant, they appear within loop bodies and take the form of integer expressions. The variant is expected to decrease after each loop iteration and must maintain a value greater than 0 throughout the execution of the loop. The variant block does not support the old keyword because in compilation the old values of any variables used within the variant are stored in order for the compiler to verify the decrementing property of the Loop variant, an example can be seen below:

```
variant {
	z - i;
}
```

The variant represented here is for the fibonacci sequence loop and displays i, the loop counter subtracted from z, the number which we want to get the fibonacci number for. Since i is getting incremented each iteration, the loop variant is constantly decreasing and stays above 0 based on the loop condition shown above

1.3 List of Advanced Programming Features

1.3.1 Feature 1: If-Else Statements

1.3.1.1 Syntax

The syntax for the If-Else statements is very easy to read. It is structured this way:

```
if (boolean expression) {
    …
}
else if (boolean exp) {
    …
} // can have as many as else-ifs as needed
else {
    …
}
```

1.3.1.2 Correct Input (Input 10)

This is a **snippet** of code from input 10:

```
class test {
    declare {
        string result;
    }
    func string smaller(int x, int y){
        require {
            x > 0;
            y > 0;
        }
        if(x < y){
            result = "x";
        }
        else if(x > y){
            result = "y";
        }
        else{
            result = "equal";
        }
}
```

1.3.1.3 Incorrect Input

This is a **snippet** of code of **input 12** but modified to be **incorrect**:

```
class test {
    declare {
        string result;
    }
    func string smaller(int x, int y){
        require {
            x > 0;
            y > 0;
        }
        if(x < y){
```

```
                    result = "x";
                }
                else if(x > y){
                    result = "y";
                }
                else{
                    result = "equal";
                }
                ensure {
                    result == "x" || result == "y" || result == 
"equal";
                }
            }

            assert {
                result == "x";
            }
        }
```

1.3.1.4 Transformation

When we translate the program, we accumulate post-condition from back to front, meaning the WP for each statement will become the post-condition for the previous statement. And if there's no previous statement, the precondition will imply the accumulated post-condition; So for loop, the way we calculate WP is as shown below:

WP(if B_1 **then** S_1 **else if** B_2 **then** S_2 …..**Else** S_0, **Q)**, will be translated into:

$(B_1 \Rightarrow wp(S_1, Q)) \wedge ((\neg B_1 \wedge B_2) \Rightarrow wp(S_2, Q)) \wedge \ldots(\neg(B_1 \vee B_2 \vee \ldots) \Rightarrow wp(S_0, Q))$

which is the weakest precondition for if-else statement to be satisfiable, which then will in turn be used as post-condition for earlier statement.

.

1.3.2 Feature 2: Switch Statements (switch):

1.3.2.1 Syntax

The syntax for switch statements are similar to that of switch statements in java or javascript except each case statement does not need to be followed by the break keyword and the case statements are followed by a pair of curly braces encasing the statements to be run for each case statement, each switch statement must also contain a default block which is the statements that run if none of the case statement conditions are true, case statements are evaluated sequentially

1.3.2.2 Correct Input

Here is an example of a switch statement being utilized in a correct program

```
declare{
    int result = 2;
}
func int example(int y){
    switch (y) {
        case: 3 {
            result = 4;
        }

        case: 5 {
            result = 7;
        }

        default: {
            result = 5;
        }
    }
    ensure {
        result > 0;
    }
}
```

1.3.2.3 Incorrect Input

Here is an example of a case statement being implemented in an incorrect manner

```
declare{
    int result = 2;
}
func int example(int y){
    switch (y) {
        case: 3 {
            result = 4;
        }

        case: 5 {
            result = 7;
        }

        default: {
            result = 5;
        }
    }
    ensure {
        result > 7;
    }
}
```

1.3.2.4 Transformation

The switch statement for specification purposes is processed very similarly to the processing of if statements, based on our WP model, each condition for the case statement is processed into an equivalence boolean between the switch variable and the case variable, and conjuncted with the negation of all the previous conditions and is used to logically imply the Weakest precondition of the case's body and the postcondition of the function

switch B **case** B_1 {S_1} **case** B_2 {S_2} …… **default**{S_k}

$((B{=}{=}B_1 \Rightarrow WP(S_1, Q) \land \neg(B{=}{=}B_1) \land B{=}{=}B_2 \Rightarrow WP(S_{2,}Q)$

$\land \ldots\ldots \neg(B{=}{=}B_1 \lor B{=}{=}B_2 \lor \cdots..) \Rightarrow WP(S_k, Q)).$

This ensures that any case whose condition is true has an effect on the WP of its statement and all the rest are False => PostCondition which can be disregarded as false implies anything. We can directly translate this into alloy by taking the negation of the above Hoare predicate to validate the correctness of the program in alloy.

We are able to support nested switch statements (as well as loops and if statements can be contained within case body) using a Parent interface which each of these constructs implements

1.3.3 Feature 3: Loops:

1.3.3.1 Syntax:

The Syntax in our loops contains 5 fundamental parts which each have a purpose in the verification of the loop, the 1st part is the **from** block which is used to initialize or redeclare variables that will be used throughout the for loop, the 2nd part is the **until** block which is used for listing the exit conditions of the loop, the 3rd part is the **invariant** block which contains the loop invariants, or the expressions which hold true at the beginning of each loop iteration, the 4th part is the **do** block, which contains all the statements to be executed within the loop, these statements include, other loops, if statements, switch statements, assignments etc. The final and 5th part of the loop is the **variant** block, this block contains the loop variant for the loop, an integer expression which decreases after every iteration and is always greater or equal to 0

1.3.3.2 Correct Input

Here's an example of a loop executed within a correct program:

```
declare{
int start;
int arr[10];
}

func void populateArr(){
    loop{
           from{
               start = 1;
           }

           until{
               start == 10;
           }

           invariant{
               start == 1 || arr[start - 1] == start;
           }

           do{
               arr[start - 1] = start;
               start = start + 1;
           }

           variant{
               size - start;
           }
    }
}
```

1.3.3.3 Incorrect Input

Here’s an example of a loop executed within an incorrect program:

```
declare{
int start;
int arr[10];
}

func void populateArr(){
    loop{
        from{
            start = 1;
        }

        until{
            start == 10;
        }

        invariant{
            start == 1 || arr[start - 1] == start;
        }

        do{
            arr[start - 1] = start;
            start = start + 1;
        }

        variant{
            start;
        }
    }
}
```

(The Program is incorrect because the variant is not decreasing over each iteration)

1.3.3.4 Transformation

Loops have the following structure in our program

```
Function(){
    require{Q}
    loop{from{Sinit} until {B} invariant {I} do {Sbody} variant
{V}}
    ensure{R}
}
```

to translate the loop into alloy, first we must come up with the Hoare Triple predicates which will be converted into alloy by the compiler. To do this, we compute the conjunction of the result of 5 hoare triples $\{Q\}\ S_{init}\ \{I\}$, $\{I \wedge \neg B\}\ S_{Body}\ \{I\}$, $I \wedge B \Rightarrow R$, $\{I \wedge \neg B\}\ S_{Body}\ \{V >= 0\}$, and $\{I \wedge \neg B\}\ S_{Body}\ \{V > V_0\}$, the result of the conjunction of the hoare triples is negated and inputted into alloy to verify correctness of the loop

However, if there are more statements before the loop, then this will just become the new postcondition for the previous statements. However, only the variables in $\{Q\}\ S_{init}\ \{I\}$(Q is just true in this case) will be replaced with assignments, because all the statements before the loop can be considered as the initialization step.

1.3.4 Feature 4: Arrays:

1.3.4.1 Syntax: The logic for arrays is slightly different from all the other primitive types, you can either declare it with a size or initialize it with a list of values. And the only 2 thing you can do to array is to get value of the specified index or change the value

1.3.4.2 Input:

Here's some example of an **incorrect** program:

```
declare{
    int[5] accounts;
    int[] ages = {16, 23, 12, 34, 25};
    int temp;
    int old;
}
```

```
func void swap (int x, int y){
        old = ages[y];
        temp = ages[x];
        ages[x] = ages[y];
        ages[y] = ages[x];      //note this line is incorrect
        ensure{
                old == ages[x] && temp == ages[y];
        }
}
```

running this in alloy will tell you its incorrect, but if you fix the swap(int x, int y), it will be **correct**:

```
func void swap (int x, int y){
        old = ages[y];
        temp = ages[x];
        ages[x] = ages[y];
        ages[y] = temp;         //now this one will be successfully verified
        ensure{
                old == ages[x] && temp == ages[y];
        }
}
```

1.3.4.3 Transformation:

when translating it into alloy, even though the user can give the array an initial value, it doesn't really make any sense to translate that. Because when verifying the program, it is only assumed that the precondition is true, we cannot assume any initial value at the start of function because the variable might have gone through arbitrarily many re-assignment steps. Therefore when initializing, any array will just be **seq Type**, without any initial value, that way the program will be verified correctly.

For array references, we can actually conveniently translate it directly as shown:

ages[5] is translated into n.ages[5]

Of course if we have an expression as the index (like ages[a + b]), that expression will be translated accordingly.

1.3.5 Feature 5: Function calls

1.3.5.1 Syntax:

when calling the function, you can just call it like function(x); However, we do not support assigning the return value of the function call, so if you want to say something like y = function(x); you would have to store the result of function into another variable and assign that variable to y

Note that it is really important to make the postcondition of the callee as strong as possible because the caller relies on that to be verified.

1.3.5.2 Input

Here's an example of a correct program using function calls (Simplified version of input-3.txt):

```
declare{
    int[9] arr;
    int tmp;
}

func void sort(){
    SWAP(0, 1);

    ensure{
        arr[0] <= arr[1];
    }
}

func void SWAP(int i, int j){
    require{
        i >= 0 && j >= 0;
    }

    if (arr[i] > arr[j]) {
        tmp = arr[i];
        arr[i] = arr[j];
        arr[j] = tmp;
    }

    ensure{
        arr[i] <= arr[j];
    }
}
```

However, if you remove the post condition for SWAP, the program will be incorrect because the caller relies on that.

1.3.5.3 Transformation

The way we verify function calls is to rely on the programmer to give us the postcondition and precondition, there are 3 things that we need to verify for function call: 1. the precondition of callee is implied. 2. The callee function itself is correct. 3. The post condition of the callee implies the postcondition of the caller, and we replace the occurrences of parameters in the pre and postcondition with the actual input parameter.

Note that this approach avoids having to go inside the callee but it relies heavily on the post condition given to us. So when giving input to our program, the programmer must make sure that the postcondition is correct and postcondition is as strong as possible.

However, by avoiding going inside the callee, we can support function recursions.

1.3.6 Feature 6: Class invariant.

1.3.6.1 Syntax: The syntax is being talked about in milestone 3, however the way we translate class invariant in milestone 3 is incorrect. It should be pre and post condition of all the function calls.

1.3.6.2 input:

Here's an correct example of class invariant

```
declare{
        int i = 9;
}

func void notChange(){
        i = i;
}

assert {
        i = 9;
}
```

In this case, class invariant will be plugged into the pre and post condition in the function. If you change i = i to i = i + 1, then the program will become incorrect!

1.3.6.3 Transformation

The transformation is solid compared to what we did in milestone 3. In logic, class invariants should be true before any function and should remain true after any function. We plug the invariant to all the pre and post conditions of all the functions and we verify all the functions. So in the case above, we will let alloy verify the hoare triple:

{i = 9} i := i {i = 9}

Which is correct.

2 Output Specification Language

```
open util/boolean

//=============test==============
some sig test {
        result:Int
}
```

```
fact testFact {
}

pred addCheck[x:Int, y:Int]{
      some n: test| not (((((n.result) >= (0)) and ((x) >= (0))) and
((y) >= (0))) => (((plus[x][y]) >= (0)) and ((plus[x][y]) =
(plus[x][y]))))
}

//The post-condition/class-invariant/loop-correctness of function
(add) is only valid when this is inconsistent.
run {
      some x: Int| some y: Int| addCheck[x, y]
} for 8 but 8 int, 2 Bool, exactly 32 String
```

Figure 2: Structure of Output Program

Figure 2 shows an example of what an output program can look like. Note that this figure is a different output from Figure 1 (which shows an input program). The first line is a library import that contains boolean values, true and false. The class name is written as a comment before the declaration of the class to help split multiple classes into a more readable output.

3 Examples

3.1 Example Input: Input-1.txt (Loop Correctness)

The input-1.txt demonstrates how a typical loop can be **successfully** verified, 5 hoare triples will be generated for alloy and alloy won't find counterexamples (The predicate will be inconsistent). And if you remove *x* = *x* - *1* in the loop body, the alloy will find counterexamples!

3.2 Example input: Input-2.txt (Conditional Sorting)

The input-2.txt demonstrates how conditional statements work, it's a real world scenario where you want to verify if a sorting algorithm can correctly sort the elements. The post condition is really simple, but it is not trivial to prove that the sorting is always correct. But if you compile it and run it in alloy, you will know that it is **correct**. Btw, if you remove any one of the lines in the algorithm, the alloy will immediately find counterexamples where the sorting would fail. (WARNING: the output program is long!)

3.3 Example input: Input-3.txt (Sorting an Array Using Function Calls)

The input-3.txt demonstrates how you would use function calls to sort an array using a sorting network. As can be seen, in this case there are 25 comparisons needed to sort a 9-elements array, it is really difficult to prove that the sorting will always work by hand. But if you run it in alloy, you know the sorting is actually **correct**. And if you remove even one comparison in the program, Alloy will immediately find a counterexample.

3.4 Example input: Input-4.txt (Switch statements)

This file demonstrates how switch statements are used. Similar to the common programming languages, the user can define as many cases as they want but they are required to define a default case, if none of the cases match.

3.5 Example input: Input-5.txt (Switch statements)

This file shows even more switch cases than the previous input file. It shows how we can now use switch cases instead of multiple if-else statements.

3.6 Example input: Input-6.txt (Modify array values)

This file shows how a program is **incorrect**. The function postcondition is obviously wrong since we change the value of the element at the index to empty. Therefore, Alloy finds a counterexample (an instance) and shows it to us. Therefore, the program is **incorrect**.

3.7 Example input: Input-7.txt (Recursion and Function Calls)

This file demonstrates the ability to call other functions inside a function. It also shows the ability to do recursive calls, however, the user **must** define the preconditions and postconditions since it is very difficult to guess and calculate the wp of an unknown precondition.

3.8 Example input: Input-8.txt

This file uses multiple specifications (such as arrays, switch, functions) to simulate a Stock market program.

3.9 Example input: Input-9.txt

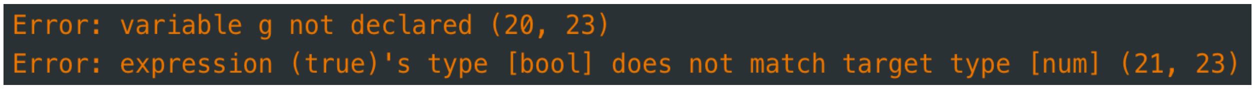
This file tests the negation of a boolean expression.

3.10 Example input: Input-10.txt

This file demonstrates the usage of if-else statements.

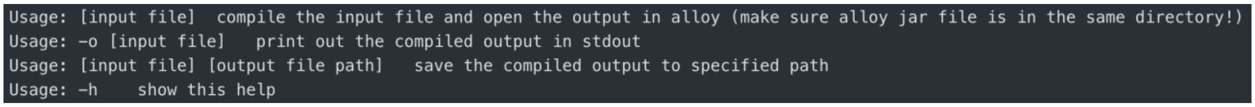
4 Miscellaneous Features.

- Semantic Check and Error Reporting
    - During parsing, out compiler will run some basic semantic checks for example if the variable is declared, does the expression match the target expression
    - The program will tell you what is the expected type of expression and which line does the error occur, eg:



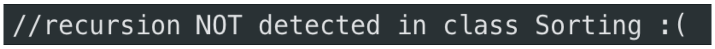
```
Error: variable g not declared (20, 23)
Error: expression (true)'s type [bool] does not match target type [num] (21, 23)
```

- Optional Commands/Arguments



```
Usage: [input file]  compile the input file and open the output in alloy (make sure alloy jar file is in the same directory!)
Usage: -o [input file]   print out the compiled output in stdout
Usage: [input file] [output file path]   save the compiled output to specified path
Usage: -h    show this help
```

-
    - When running the program, the user have 4 different commands options
    - *java -jar ProgramApp.jar [input file]*
        - this will compile the input file and open it in Alloy for the user.
    - *java -jar ProgramApp.jar -o* [*input file*]
        - this will print out the compiled output to stdout.
    - *java -jar ProgramApp.jar [input file] [output file]*
        - this will compile the input file and save the output in the output file.
    - *java -jar ProgramApp.jar -h*
        - this will simply print out the usage of the program.
- Recursion Detection
    - We also did recursion detection, it was originally used to avoid verifying recursion calls because supporting recursion is hard. But after some research we realized that by asking the programmer to specify the post-condition, we can support recursion. So this feature doesn’t really contribute to verification anymore.
    - After parsing the input, the program will build a function call graph and it will do a DFS on the graph to detect cycles, if there is a cycle, that means the input have recursions
    - The program will tell the user if there are any cycles in any classes:



```
//recursion NOT detected in class Sorting :(
```

5 Limitations

- Our Compiler does not support Array reassignments, only the individual elements of an array can be reassigned a value
- Our Compiler does not support user defined Class based type declarations for variables, variables can only be declared using the predefined types of string, int, and bool
- Functions in our compiler do not return values, instead users may declare global variables and assign the final result of the function to them.
- Our Compiler does not support Class inheritance
- Pre-conditions, Post-conditions, Assumptions and Assertions only support boolean expressions, if one wishes to run a function, loop or conditional within these, they must do this in a function and assign the corresponding values to variables which can be used within a boolean expression