The program currently consists 6 classes that does the parsing and the encoding tasks.

Main.cpp

Main class does the parsing by separating input lines using space delimiter and saves the tokens. After, these tokens are mapped to either main function's vector(mainTokenArray) or thread and function's vector (tokenArray). Also, thread, function and block starting program counters are saved in that part.
In the first parsing part, function parameters are saved as well.
After the categorization is completed, the *generateGlobalPart* function is called.

- *GenerateGlobalPart:* This part reads the lines in the LLVM files global part in order to create global variables. Using the instructions' grammar, it saves the variables, their types etc., checks if there is any mutex/condition variables, gets threads and finally creatas store instructions for these global variables using their initial values.

After these global variables, mutexes. threads etc. are saved, **declarationGenerator**'s execute function is called. That function generates the declaration of these variables in SMT-LIB format. After global part is complete, *generateMainPart* function is called.

- *GenerateMainPart:* This part reads the lines in the main function which is related mostly to thread operations. Store, load etc. types are NOT handled in this part. PTHREAD's create, join operations are handled in that function. Also, non-functional instruction calls are handled in here as well.

After main part, *initializationPart* function is called.

- *InitalizationPart:* This creates the first assert commands in SMT-LIB file. Creates the formula for threads and functions' initial program counters and copies the initial values of global variables to a new variable with "_0" suffix. When a global variable changes with a store operation, the suffix is incremented by 1.

Now, the initial scheduling part begins, using *schedulePart* function.

- *SchedulePart:* That function takes a "isFirst" boolean parameter, that states if it's the initial scheduling or not. That effects the paranthesis count in the scheduling part because the initial part contains opening paranthesis for each execution step. Then, the scheduling mechanism is added: If Tx is selected, other ones shouldn't and vice versa, using the thread maps. Also, for functions, call_<function_name> variable is added to schedule.

After initial scheduling part, the conversion operation begins. Main part takes the tokens from the tokenArray, line by line, and using the grammar, sends it to **instructionGenerator** object to do the conversion operation. In main part, the thread/function change operations are handled i.e. when the instruction is now belong to a new thread, the change is read and the converted formula is mapped to the new thread.

Other than that, the next available program counter trick is done here. That means, if an operation is not a branch operation, it means the next program counter can only have one value, so only that operation is added to possible program counter set. If it's a branch operation, both of the possible two outcomes are saved.

Before starting to generate the formula, *generateAllowedPcMap* function is called. As we mentioned in last parsing part, we have a possible Program counter map for each step. That means, in the first step, only the first instructions of the threads are encoded. Because, it is impossible to reach other instructions. In each step, every possible program counter is recorded while the conversion operation is done, so impossible to reach instructions are not encoded in that step in order to decrease redundancy in the formula.

- *generateAllowedPcMap:* This function uses that saved program counter values and now maps the encoded instructions to possible program counters. First, every thread/function's initial program counter is taken and the matching encoded instructions are saved to a map. This operation continues for each thread until all possible program counters/encoded instructions are mapped.

After allowed instructions are mapped according to step count/execution flow, now the formula is created. For each step, possible encodings are taken from the map and connected.

Also, the conversions are done using only 1st step, (current step = 0, next step =1) because of that, for each step, replace function is called.

- *UpdateToBound:* Takes the encoding and updates the step numbers according to current step. Initially, all encodings have 0 for current step and 1 for next step. These numbers are replaced for threads, global variables, mutexes etc.

In the end, fail condition is added, using the saved program counters for possible fail checks.

This is the summary for the task of **main** class. Helper classes and their tasks are explained below.

mainPartGenerator

Because some calls from the main part is different from the regular threads and functions, this class is needed to handle these instructions. It has a pcCounter as a field which is updated according to the current step.

- *pcType:* Creates the encoding of program counter update formula. Takes the program counter value as parameter and returns the encoding.
- *storeType:* For store type operation. Creates the encoding accordingly depending on the variable type (value, local variable, global variable)
- *callType:* Used for thread call operations. Trivial.
- *bitCastType:* Usually, bitcast operation is done for thread parameters. This function gets the source, length of the source, target and length of the target as parameters. Using the difference between these lengths, creates the encoding. If source is bigger, “extract”, otherwise “zero_extend” term is used.

globalStore

Very simple class with only *globalStorePart* function. Takes global variable name and its value as parameters. Using the globalTypeMap, creates the assignment encoding.

declarationGenerator

This class is used for creating variable declarations and definitions in the formula. Only execute functions are called from outside classes and it handles the operation inside. Both global and local variable declarations are done in this class.

- *initialize:* The initial declarations for the formula such as set-logic, set-option, I8 to BitVec 8 etc.

- *generateGlobalDeclarations:* Declaration of global elements in the program such as global variables, threads, mutexes, condition variables and malloc control variables. These functions are trivial, they iterate through these global elements’ maps and creates the definitions.

- *generateLocalDeclarations:* In the parsing part, all variables are mapped with their types in the variableTypeMap. This function iterates this map and using the key-value pairs, generates the definitions.

Utils

This class has some utility functions and mostly static maps for holding usable information.

- *findLog:* an integer value is passed to the function and it returns its (logarithm in base 2)+1 value as string. +1 is because of the bitvector size. For example, 9 can be represented in 4 bits, but integer division results in 3.

- *myReplaceAll:* a string, a source and a target is provided as parameters. This function replaces all “source” substrings in the given the string to “target” substring.

Example: myReplaceAll(“alicanal”, “al”, “ka”) will result in “kaicanka”.

- *replaceWithI:* Replaces the integer prefix “i” in types; i8, i16 etc. with uppercase, “I”.

- *myToString:* Converts the provided integer to string.

- *split:* Splits the given string using “ “ (one character whitespace) delimiter and returns the resulting tokens.

- *insertTemp:* Sometimes, the variables in LLVM representation, do not have tmp prefix, and they are only numbers. In order to prevent grammar errors in SMT-LIB, tmp prefix is added to these kind of variables. If it already contains some character prefix, tmp is not added.

Utils Instance Variables:

- *variableTypeMap:* Contains local variables and their types.

- *globalTypeMap:* Contains global variables and their types.
- *globalMap:* Contains global variables and their program counters where a store operation is done.
- *mutexSet:* Contains mutex variables.
- *condVarSet:* Contains condition variables.
- *threadMap:* Contains threads and their starting program counters.
- *functionPcMap:* Contains function names and their starting program counters.
- *threadFunctionMap:* Contains threads and the function names that are called from that respecting thread.
- *blockMap:* Contains block names and their starting program counters.
- *globalStoreCount:* Contains global variables and how many times a store operation is done on respecting global variable. This store count is not used anymore.
- *initialValueMap:* Used for global variables and their initial value.
- *condVariableThreadMap:* Map contains the relation between condition variables and threads, i.e. which condition variable is signalled from which thread.
- *mallocMap:* Contains the global variable which a malloc operation is done, and size of the memory allocated and size of the target.
- *mallocPcMap:* Contains program counters where a malloc operation is done and the variable which the operation is done.
- *targetSizeMap:* Used for strcpy operations. Contains the variable and the size of the copy.
- *functionParameterMap:* Contains function names mapped to variable name and types, which are the corresponding parameters of that function.
- *returnMap:* Contains function names and their return types.
- *defineSet:* When there are arrays, their size is converted to corresponding SMT-LIB bitvector size. For example, an 56 element integer array will have 8X8 two dimension bitvector array in SMT-LIB, and for easier representation, a new decl is used, named Array56xI8. If more than one variables have the same type, SMT-LIB gives an error, saying it's already defined. This set holds all array definitions to prevent this type of collisions.
- *threadLastPcMap:* Contains threads and their ending program counters.
- *localPointerMap:* Sometimes, a global variable is assigned to local variable before modifying it. However, in SMT-LIB, aliasing or pointer logic is not possible, so we

save all these assignments, and when the local pointer is used again, the program replaces that with the real target. This map contains these relations.

- *arraySizeMap:* When a store or load operation is done on an array, in SMT-LIB, the type of array and type of elements are needed. This map contains the array names and their sizes to use it when needed.

instructionGenerator

This is the class which all encoding is done. Tokens are passed from the main class to this class and using the instructions, the conversion is done. First thing done is adding an “and” to the encoding if the instruction is NOT one of the following: branch, block name, call, unreachable or return. Then, by checking the instruction, the conversion is done. The encoding functions are explained below:

- ***compare****:* If the third token in the line is “icmp”, that means this is a compare instruction. First token is the target boolean element and its type is usually bool. The function takes following parameters in following order:

result variable, type of the comparison (equal, greater, smaller etc.), first element to be compared, second element to be compared and type of the result (sometimes the result is not boolean).

The required checks are: if the second parameter is null or not and if the parameters are local variables or immediate values. Using these checks, the encoding is done. In the encoding, use bveq, bvslt or bvsgt according to the passed type and put parmeters in the order. Another thing to check is, if comparison is done with a negative integer, we need to add bvneg keyword in front of it.

(= <result_param> (<bvsgt|bvslt|bveq|=> <param1> <param2>))

An exceptional condition is, if one of the compared parameters is “null”, then the saved last pointer’s global variable is used and init_<global_variable> is checked.

- conditional brach*:* If the second token is “i1”, that means there is a branch operation using that variable. The function takes following parameters in following order:

the variable which the done branch using upon, first label to jump if the variable is true, second label to jump if the variable is false.

Sometimes, the parameters are not local variables, instead, they are directly true or false. There are checks for that condition in the function.

The encoding is following: (or (and <condition variable> (= nextpc <pc_of_label1>))(and (not <condition variable>) (= nextpc <pc_of_label2)))

- unconditional branch*:* If the second token is label, then this is an unconditional branch operation. The function only takes the label as the parameter. This is a trivial encoding, just get the pc of the label and use that formula: (= nextpc <pc_of_label>)

- allocate: If the third token is “alloca”, that means this is an allocation operation. Allocation operation does not have any encoding, only thing to do is getting the element and its type for future usage.

- mutexLock*:* If the 6th token in the line is
"@pthread_mutex_lock(%union.pthread_mutex_t*"
or "@pthread_mutex_lock(%struct._opaque_pthread_mutex_t*", that means this is a mutex lock operation.

The function only takes the mutex as the parameter. Encoding is trivial, just make the variable false for the next step. Also, the chosen mutex is saved and a boolean variable called isLock is assigned to true. These variables help to correctly form the next state global variable/mutex assignments. If there isn’t a lock/unlock operation, in the end, we state that these mutexes will stay the same for the next step. However, if there is a lock or unlock operation, we do not write the formula for staying the same.

- mutexUnlock*:* Very similar to mutexLock, now instead making the variable false, we make it true for the next step.

- conditionVariableWait*:* If the 6th token contains "pthread_cond_wait", that means it is a wait operation of PTHREAD.

The function, *waitType* takes two parameters: First parameter is the condition variable and the second parameter is the mutex to be locked. First, we check if the mutex is locked or not, and it should be locked. Then, we make the condition variable true for the next step. In my encoding, if there is a condition variable related to a thread, in the scheduling, the condition variable is checked if it’s true or not. If it’s true, that means a wait operation is done on that condition variable so that thread cannot be selected until a signal operation is done.

- conditionVariabeSignal*:* Similar to wait operation, now the condition variable is converted to false. Also, for signaling operation, there is no need to check if the related mutex is locked or not.

- malloc*:* If the 5th token contains malloc in it, that means it is a malloc operation. Briefly, if there is a malloc operation on a variable, that variable is saved in the parsing stage. And,

there is boolean variables for all possible malloc target variables. These variables get true or false value non-deterministically. That means, malloc might fail or work successfully.
The global boolean variables are in mallocPcMap, with program counter as a key.
The boolean variable, named <init_<name_of_the_global_variable>> is assigned to true or false for the next step.
- objectSize*:* if the 5th token contains "@llvm.objectsize", that means this is an object size instruction. Normally, LLVM uses this instruction to check if malloc is successful or not. However, as explained in malloc part, we are using init_<global_var_name> variables to check the condition of the malloc operation. As a result, this is only a placeholder formula, where the target is checked if it is equal to one or not (in LLVM, if not, that means malloc is unsuccessful, so we are making that check always pass and instead do the check using the init_<global_variable> variable). As stated in the comments, it might change in the future.
- memCopy*:* If the 5th token contains "@__memcpy_chk", that means this is a memory copy operation. Actually, in the benchmarks, this operation is only done for string copy, so we are following that tradition as well, using it like a strcpy operation. However, if required, it might change for supporting memcpy operations as well.
The function takes three parameters:
Source of the string copy operation, target of the string copy operation and size of the elements to be copied. Also, it's important to state that, LLVM does not directly copy the strings between global pointers, instead, the global pointer is assigned to a temporary local variable and then, copy operation is done on that local variable. In order to mirror it on the formula, we save every pointer load operation on a variable, and that variable holds the actual global target. When there is a string copy operation, that actual global variable is passed to the function and it is used in the encoding.
On our encoding, all characters are represented by their ascii values converted to bitvector values. The operation to be done on the SMT-LIB is store to save, and select to get elements from the source array.. So, every character is copied to a SMT-LIB array using the store and select operation.
foreach character in the string:
(= target<target_name><next_step_count> (store
target<target_name><current_step_count> <index> (select <source> <index>)))

***- phi**:* Phi operation is used in LLVM to check the source of the branch operation. Assume there are two blocks which jumps to a same target block. Phi operation checks where was the source of the jump, first or the second block.
In the encoding, we compare the last program counter with block program counters. If the last program counter is smaller than the value of the program counter associated with block that has larger program counter, that means it is done from the first block and vice versa.
The parameters are:
Target of the phi instruciton, first value to be compared, first label to be compared, second value to be compared, second label to be compared and the result type.
(or (and (bvslt <thread+previous_pc> <larger_of_block_pcs>)(= target value))
(and (not (bvslt <thread+previous_pc> <larger_of_block_pcs>)(= target value2)))
***- add**:* If the third token is “add”, that means this is an addition operation. We need to check, if the addition operation is done using local variables or immediate values. After the check, the tokens we need to pass are:
target of the addition, first value to add, second value to add, and type of the added values (i8, i16, i32 etc.)
We cannot use + operator because we use bitvectors as representatives of our values. So, “bvadd” term is used in the encoding.
(= <target> (bvadd (<value1> <value2>)))
***- multiply**:* Very similar to add operation. Now, the operation is multiplication.
***- trunc**:* Trunc operation is used for removing some part of the variable, in other words, truncation operation. The third token should be “trunc” in the LLVM instruciton. We first get source length and the target length from the instruction. After that, the difference, in other words, length of the part which will be truncated is calculated. The parameters to the function are:
Target of the operation, variable which will be truncated, length of the truncation part.
(= <target> ((_ extract <length> 0) <source>))
***- zext**:* Zext, in other words zero extension, can be considered as reverse operation of trunc. Now, we are adding zeros to the variable and making its size bigger. The parameters to be passed are:
Target of the operation, original length of the source variable, source variable, final desired length.

Also, the local-global variable exchange is required for that function as well. In LLVM, the operation is usually done on temporary local variables so we need to use the real global target to modify.

***- and**:* Logical and operation. Third token of the instruction should be “and”. The parameters to be passed:

Target of the operation, first value to be compared, second value to be compared, type of the values to be and’ed. If variables are boolean variables, we can use “and” keyword in SMT-LIB, however, in other cases, we should use “bvand” term.

(= <target> (and|bvand <value1> <value2>))

***- or**:* Similar to and operation.

***- Bitcast**:* This combination of zero extend and sign extend operations. When a type is casted to another type, this instruction is used. Third token should be “bitcast”. The function takes following parameters:

Target of the operation, the source length, the target length, source variable, type of the instruction (bitcast or sign extension).

If it is a bitcast operation, the keyword should be used is “zero_extend”. Otherwise, “sign_extend” keyword is used.

(= ((_ zero_extend|sign_extend <extension_length>) <source_param>))

- getElementPointer: This for either getting an element from an array or from a struct. The type can be found by checking the 5th token and checking if it starts with ‘[‘ character. If that’s the case, that means it is for getting an element from an array, otherwise from a struct. Array version is relatively simpler. The function takes the following parameters:

Target of the operation, source array, first pointer, target index. Select operation is used in SMT-LIB to get elements from bitvector arrays.

(= <target> (select <source_array> ((_ extract <log2_of_array_size> <first_pointer>) <source>)))

Struct type function takes all tokens as parameters because there are a lot of things to consider.

First if check is to see, if the target is a function parameter, which is actually a global variable passed from another function. Then, we need to get the offset. In my encoding, structs are similar to arrays, where the elements of the struct are separated using offsets. The offset length is the 10th token in the instruction. Also, after getting the element, it is saved to a global variable because, usually that element is assigned to a temporary local

variable and modified later. In order to mirror it in SMT-LIB encoding, we are saving all get pointer operation targets in global variables, so if the local target is modified, we can do that operation directly to the real target element.
Currently, there isn't any encoding for that type of instruction because, as we stated, usually the target is assigned to local temporary variable and modified later. Instead of this assignment, we just save the operation by mapping the local variable to the actual target, and when a modification is done on that local variable, we are doing the same operation on the real target. Because, in LLVM, that operation on the local variable is effecting the global variable or the real target as well, however this is not the case for SMT-LIB. So, there is no need to do that temporary assignment.
- store*:* If the first token is "store", then this is a store operation.
Store function takes the following parameters:
Value to be stored, target of the store operation, if there is align then the length of the alignment, is store operation done on a pointer and all tokens in that line.
Store is complex mostly because of the pointers and parameters should be considered carefully.
For example, sometimes store operation is done directly on a "getelementpointer" instruciton. That means, the element isn't directly provided, instead, it is specified with a element pointer and getelementpointer instruction. We can check it by looking if the target contains "get" keyword. If that's the case, then the actual target is 10th token and it's usually an array so we need to use "store" keyword to store in an bitvector array in SMT-LIB.
Also, another check should be added if the stored value is a local variable or an immediate value.
If it is a pointer, that is checked if the target is in globalPointerMap or not, then, we get the actual target from the globalPointerMap and do the store operation on that variable.
If there was a getElementPointer operation previously, as stated on that instruction's description, then the store operation should be done on that element, that means, instead of the temporary local variable, the store operation is done which the local variable points. The actual target is stored on a global variable when a getElementPointer operation happens, so we can get the actual variable from that global variable.
Also, another local pointer check is added as well for the real target assignment in the SMT-LIB. In other cases, the store encoding is trivial, just assign the value to the target.

Another thing to consider is that, as we stated, at the end of each step, global variables, mutexes, condition variables etc. are updated for the next step. If there isn't any change on these variables, we can just say: (= <global_variable><pc+1> <global_variable><pc>), stating the variable will stay same for the next step. However, similar to mutexes, if there is a store operation on a global variable, it makes the "isStore" boolean variable true, so that global variable won't be stated as staying the same for the next step.

- *load*: If the third token is "load", then it is a load operation. Load is also complex like store, because of the similar concerns. The function takes the following parameters:

Target of the load operation, source that will be assigned to the target, length of the alignment if there is an alignment, if the load operation is done on a pointer (boolean).

First, it is checked if the source value is an immediate value or a local variable. If it is an immediate value, operation is trivial, assign that value to the target.

If not, then, similar to store operation, the boolean variable, which holds if there was a previous getPointerArray operation is checked. If that's the case, then the actual variable in that array is used instead. "select" keyword can be used to get elements from bitvector arrays in SMT-LIB.

If not, the value is checked if it's a local or a global pointer. If so, the actual target is taken from the maps, which holds the pointer - target mappings (Utils::localPointerMap and globalPointerMap).

In case of no pointer, normal assignment encoding is used: just assign the target to the source value.

- *call*: Call operation is complex when a parameter is passed. If the function do not remove any global variables or do not return anything, it is not required to write any encoding.

In other case, the function takes the following parameters:

Target of the function (currently unused because return is handled in other function), name of the function, start index: starting index of the parameters in the token array, token array.

First we assign all the parameters to their values from the called function using Utils::functionParameterMap.

After parameters are assigned, the boolean variable in the encoding, call_<function_name> is made true. With that boolean variable, we are making the calling thread unable to work before that called function is finished, as required. After the function's return statement or last program counter is reached, the calling thread can be selected by the scheduler again. (= call_<function_name><next_step_counter> true)

I’m not sure if the call encoding works 100% correct so we need more tests on this.

After the required encodings for instructions are done, we are adding the lastStorePart, which assigns global variables, mutexes, condition variables, malloc global boolean variables etc. for the next step. If they are not changed, the encoding will state that, they will have the same value for the next step. In other case, the changed variable won’t be written in that encoding.

- *return**:*** If the first token is “ret”, then it means it is a return statement. If this is a valid function, then we will make call_<function_name> false, so the scheduler won’t choose that function for the next step and the caller thread can continue its execution and can be chosen by the scheduler.
Also, if the returned value is not void, the value will be assigned to the variable on the caller thread accordingly.