LTL Algorithm

Code Structures

The codes consists of the following files:

- `Expr.cpp` : The definition of the expression tree. Also contains the definition of closure and elementary sets. Some conversion functions are also defined here.
- `Parser.cpp` : Use Pratt Parsing to parse the LTL formula. It is assumed that the input formula has enough parentheses to make the parsing unambiguous, so the parser does not need to handle the precedence of operators.
- `NBA.cpp` : The definition of the NBA(non-deterministic Buchi automaton) and GNBA(generalized NBA). The formula will first be converted to a GNBA and then to a NBA.
- `TS.hpp` : The definition of the transition system.
- `Product.cpp` : Calculate the product of NBA and TS.
- `NestedDFS.cpp` : The nested DFS algorithm.
- `SCCProcessor.cpp` : Calculate the strongly connected components of the product by Tarjan's algorithm.
- `Utils.hpp` : Defines the `failwith` macro for debugging.
- `main.cpp` : The main function of the program.

Algorithm

This code implements the LTL model checking algorithm which can be found in the book "Principles of Model Checking".

It first parses the LTL formula φ and converts $\neg\varphi$ into GNBA, and later converts into NBA $\mathcal{A}$ such that $L(\mathcal{A}) = L(\neg\varphi)$.

Then it calculates the product of the NBA and the transition system.

In the end, it uses the nested DFS algorithm to check whether a node in the accepting set is reachable from the initial states and is contained in a circle.

An alternating algorithm of nested DFS is Tarjan's algorithm. If a node in accepting set is reachable from the initial states and is in a SCC(strongly connected components) that contains a circle(that means the size of SCC >= 2, or the SCC contains a self-loop), then the formula is not satisfied. Both nested DFS and Tarjan's algorithm are implemented.

Data Structures

Expression Tree

Expr is a abstract class that represents the expression tree. The VarExpr, UnaryExpr, BinaryExpr class are derived from Expr. The VarExpr represents the atomic proposition, UnaryExpr represents the unary operator, and BinaryExpr represents the binary operator.

```
1 // Some code is omitted for brevity
2 enum class ExprType {
3    TRUE, VAR, NEG, CONJ, DISJ, IMPL, NEXT, ALWAYS, EVENTUALLY, UNTIL
4 };
5
6 class Expr {
```

```
 protected:
   ExprType type;
};

typedef std::shared_ptr<Expr> ExprPtr;

class VarExpr : public Expr {
 private:
   std::string var;
};

class UnaryExpr : public Expr {
 private:
   ExprPtr expr;
};

class BinaryExpr : public Expr {
 private:
   ExprPtr left;
   ExprPtr right;
};
```

Closure and Elementary Sets

ExprSet contains a vector of ExprPtr. It offers the functions to check if an expression is in the set(by comparing the syntax tree).

Closure is inherited from ExprSet. It represents a ExprSet that is closed under the negation operator. It also contains the primary expression and a map that maps evry expression to its negation.

Elemetary is also inherited from ExprSet. It represents a ExprSet that is elementary.

ElemetarySet is a class that contains a vector of Elementary.

```
// Some code is omitted for brevity
class ExprSet {
 protected :
   std::vector<ExprPtr> exprs;
 public:
   ExprSet() {}
   ExprSet(const ExprSet &exprset) : exprs(exprset.exprs) {}
   ExprSet copy() { return ExprSet(*this); }
   std::vector<ExprPtr>& get_exprs() { return exprs; }
   bool contains(ExprPtr expr) {
      for (auto &e : exprs) {
         if (ExprEqual(e, expr)) return true;
      }
      return false;
   }
   void add(ExprPtr expr) { exprs.push_back(expr); }
};

class Closure : public ExprSet {
 private:
   ExprPtr primary;
   std::map<ExprPtr, ExprPtr> negation;
};
```

```
24
25 class Elementary : public ExprSet {};
26
27 class ElementarySet {
28  private:
29    std::shared_ptr<Closure> closure;
30    std::vector<Elementary> elementaries;
31 };
```

NBA and GNBA

NBANode is the class for node in both NBA and GNBA. Since the NBA in this algorithm only contains the transitions that are labeled with the atomic propositions of elementary set, the node contains a set of atomic propositions.

NBA_Base is an abstract class that represents the NBA and GNBA. NBA and GNBA are derived from NBA_Base. The only difference between NBA and GNBA is the acceptance condition.

```
1  // Some code is omitted for brevity
2  class NBANode {
3   private:
4     int id;
5     int is_initial;
6     int is_accepting;
7     std::set<std::string> ap;
8     std::set<int> transition;
9  };
10
11 class NBA_base {
12  protected:
13    int node_count;
14    std::vector<int> initial;
15    std::set<std::string> aps;
16    std::vector<NBANodePtr> nodes;
17    std::map<int, NBANodePtr> node_map;
18 };
19
20 class NBA : public NBA_base {
21  private:
22    std::set<int> accepting;
23 };
24
25 class GNBA : public NBA_base {
26  private:
27    std::vector<std::set<int>> accepting;
28 };
```

Transition System

Actions are not recorded in the transition system since the algorithm only needs to know the set of atomic propositions of the state.

```
1  // Some code is omitted for brevity
2  class TSNode {
3   private:
4     int id;
```

```
    int is_initial;
    std::set<std::string> ap;
    std::set<int> transition;
};

typedef std::shared_ptr<TSNode> TSNodePtr;

class TS {
 private:
    int node_count;
    std::vector<int> initial;
    std::vector<TSNodePtr> nodes;
    std::set<std::string> ap;
};
```

Running the Code

The code has a `CMakeLists.txt` file. You can build and run the code by running the following commands:

```
mkdir build
cd build
cmake ..
make
make run
```

You can change the input file by changing the `ts_in_path` and `ltl_in_path` defined in the main function.