Summary of the modification in the C++ version of ATPG program

Author: Bing-Chen (Benson) Wu

Date: 01/21/2018

1. This program is written by C++ (with some calls of C standard functions) and follows the C++11 standard. Please note that, in GCC, with only version 4.8 or newer can fully support the C++11 standard; otherwise, it might be failed to compile. The compiler used to compile the release version is *gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.4)*.
2. The ATPG is treated as a static object, `ATPG`, which contains all of the global functions defined in original C-version, including logic simulator, fault simulator, and PODEM. The full picture of `ATPG` class with all declarations of member functions and variables are defined in atpg.h, and the detail implementations are defined in following separated .cpp files:
   - atpg.cpp: Constructor of `ATPG` class.
   - tpgmain.cpp: Main function and some utilities.
   - input.cpp: Input processor and related functions.
   - level.cpp: Leveling routine and related functions.
   - sim.cpp: Logic simulator and related functions.
   - podem.cpp: PODEM and related functions.
   - test.cpp: Trigger of fault simulator and PODEM.
   - init_flist.cpp: Fault list builder, dummy gates builder, and fault coverage calculator.
   - faultsim.cpp: Parallel-fault event-driven fault simulator and related functions.
   - display.cpp: Information printers.
3. STL containers are widely used in this program. For example, `array<class T, size_t N>` is used to build the hash tables of wires and nodes, and `forward_list<class T>` is used to build the fault list. Basically, all of the data structures used in this program are built from STL containers, including the input/output wire list of a node and the driven/driving node list of a wire. By using STL containers, the declaration of variables can be more succinct. For example, there is no need to declare `nin` nor `nout` to indicate the number of input/output (driven/driving) wires (nodes). In addition, several operations of ATPG and simulators can be easily performed with STL containers, such as arranging decision tree, fault dropping, and leveling.
4. Some C++11 features can be found in this program:
   - Smart pointers (`unique_ptr<class T>`) is used to hold the dynamic objects. This feature would have more detail discussion latter in the item 5.
   - Range-based for loop would be used to access the elements in the STL container.
   - Lambda expression is used to state the condition of fault dropping, and it is able to count the number of detected faults simultaneously. By using lambda expression, fault dropping can be performed without hand-crafted loops while it can be easily performed by calling a `remove_if(Predicate pred)` function on `forward_list<class T>`, where the `pred` is a predicate function object that is implemented by the lambda expression here.
5. `NODE`s, `WIRE`s, and `FAULT`s are treated as dynamic objects and maintained by smart pointers

`unique_ptr<class T>` with aliases, `nptr_s`, `wptr_s`, and `fptr_s`, respectively. The deletion of the objects would be automatically maintained by these smart pointers to improve the safety of memory management. In this program, the smart pointers are mainly responsible for holding the dynamic objects, so any manipulation to the objects is done by traditional object pointers instead. In addition, the traditional object pointers, `NODE*`, `WIRE*`, and `FAULT*` also have aliases, `nptr`, `wptr`, and `fptr`, respectively.

6. The most of pass by pointer operations is replaced by pass by reference except when passing the wires, nodes, and faults.
7. The function augments are prefixed by the key word `const` if proper.
8. A makefile is provided with an additional option: direct. By calling this option, the whole pack of source codes would be directly compiled and linked into an executable file without preserving the intermediate object files.