Report on DSnP Final Project

蔡承佑　b03901078
b03901078@ntu.edu.tw

January 21, 2016

1 Data Structure

1.1 CirMgr: A class that manages the interface between the Cmd interface and the CirGate member

`_gateList`	Stores the `CirGate*` pointers in order of their lineNo, which is more convinient for circuit parsing.
`_idList`	A `vector<unsigned>` which maps ID to the index in _gateList

`_PINum`
`_PONum`
`_AIGNum`
`...`
	various numbers that stores the parameters of this circuit.

`_dfsList`	Stores the gate pointers that are in the dfs list. The dfs list is frequently used in this program, so it's better to store one.
`_FecList`	Stores the FEC groups. Its more detailed data structure is defined in **CirFec.h**.
`_simLog`	`ofstream*` to output the value of simulation.

1.2 CirGate: A base class that defines all variables that are needed in any kind of gates

`_lineNo`	The line number of the gate defined in the aig file.
`_id`	ID defined in the aig file.
`_ref`	A parameter for doing dfs.
`_value`	Values of simulation. Defaults to all zero.

Static Members

`_globalref`	Corresponding to `_ref`, used for dfs.
`_printOrder`	Record the number that a gate should print when calling `printGate()`.
`_indent`	A string consists of various spaces for indentation of `reportFanin()` and `reportFanout()`.
`USED`	A hidden parameter that stores whether the gate is in the dfs list in the 30^{th} bit of `_id`.

`INV`	A hidden parameter that stores whether fanin or fanout is inverted in the LSB.

And there are 5 types of other gates, so I define 5 other derived class to inherit the base class `CirGate`.
There are only minor difference among them for specific perpose.

Variable	classes that have it	Explanation
`_symbol`	`PI`, `PO`	Only PI and PO has symbol, other class don't need it.
`_fanin`	`PO`, `AIG`	Only these classes have fanin. Besides, `_fanin` in `PI` is a `size_t` but `size_t[2]` in `AIG`.
`_fanoutList`	`PI`, `AIG`, `Const`, `Undef`	It doesn't make sense to store fanouts in `PO`.
`_FecGroup`	`AIG`, `Const`	FEC groups consist only of AIG and CONST. PI and PO are not considered in this program.
`_var`	`PI`, `PO`, `AIG`, `Const`	Undefined gates won't be covered in the dfs list.

For common functions like `reportGate()`, `setFanout(size_t fo)`, `getTypeStr()`, they are defined as virtual function in the base class in order to call with `CirGate*` without cast.

1.3 FecGroupList: A wrapper class that stores all FEC groups with member function that updates the group list when new simulation is performed.

In fact, I've tried two data types for storing the groups. Here are their comparison.

	Advantage	Disadvantage
`queue`	• FIFO • More efficient when both `pop_front` and `push_back` are needed	• Not random access. • Must use `pop()` and `front()` to triverse. • `pop()` calls the destructor of the object. • Must copy the object when triverse since the reference one is destructed
`vector`	• No need to copy the object. Just use the reference one • The element is not destructed while triverse.	• $O(n)$ for `pop_front`. • Not a good choice when `pop_front` is frequently used.

And here's the result of a experiment:

	Reference Program	Implement with `queue`	Implement with `vector`
Period Time	0.71 seconds	0.83 seconds	6.04 seconds
Memory Used	4.473 MB	66.34 MB	66.21 MB

At last, I choose `queue` to implement `FecGroupList`.
And the two most important function in the class `FecGroupList` is as follow.

`update()`	This functions is called when new simulation values have been evaluated. Triverse all the current FEC groups and using a `HashMap<value,GateList>` to split the the gates with different value. Pop out the first FEC group from the queue and collect valid new FEC groups(size>1) to push into the queue.
`updateFecGroupInGates()`	As its name suggests, it update the `_FecGroup` in every gates. This function is called when all the simulation patterns are done. And it sort the elements in FEC groups in order of their ID.

2 Algorithm

2.1 CirSweep

Use the 30th bit of `CirGate::_id` to record whether the gate is used (in `_dfsList`). Triverse all valid gates in `_gateList` and check the bit. If it indicates that the gate is not in the `_dfsList`, delete it.
∗ Deleting a gate is like deleting a node in double linked list. Just pull out its fanins and fanouts.

2.2 CirOpt

Declare a new function `checkFaninType` in `class AIG` (only AIG need this function), and define a enum `FaninType` for the function to return which type of the combination of the fanins the AIG belongs to.
There are 5 types of fanin combinations:

1. Two fanins are definitely the same.
2. Two fanins are with the same ID but inverted.
3. One of fanin is CONST 0.
4. One of fanin is CONST 1.
5. None of above.

First identify which type the gates belongs to. The operation done to type 1 and type 4 are similar, and type 2 and type 3 are similar. In this function, the two gates to be merged is in series (a little bit different from the case in cirStrash and cirFraig) And there a inverting problem should be properly handled (When the inverting fanin merges a gates, the fanouts inherited by the merging gates should also invert)

2.3 CirStrash

Construct a `HashMap<AIGKey,AIG*>`, where `AIGKey` is a wrapper class which implements the `operator()` and `operator==` for the Hash Function. Here `operator()` is the sum of `(size_t)_fanin[0]` and `(size_t)_fanin[1]`. Use the HashMap to identify gates with the same fanins.

```
foreach(AIG,_dfsList)
  if (check(AIGKey(AIG),mergeGate))
    merge(mergeGate, AIG);
  else
    forceInsert(AIGKey(AIG),AIG*);
```

In this case, the `merge(g1,g2)` function transfers the fanouts of g2 to g1 and delete g2 from the `_fanoutList` of its fanins.

2.4 CirSim

1. Get the pattern whether from the file or randomly generate.
2. Pack it into a 32-bit `unsigned` and pass them into all PIs.
3. Call `AIG::evaluateValue()` to evaluate the value of the AIG.
4. Call `PO::fetchValue()` to fetch the value of its fanin gate.
5. Classify their FEC groups according their value just evaluated.
   (Method of classifying FEC groups have been mentioned above, with the data structure `FecGroupList`
6. Output the value of POs to `_simLog` if logFile is assigned.

2.5 CirFraig

After simulation, we've get some FEC groups. And in this part, we using a SAT engine to proof the equality of these FEC groups.

1. Set up the SatSolver.
2. Triverse the circuit from `_dfsList`, fetch a FEC and proof whether (`*this`⊕FEC) is SAT. If SAT, they're not equivalent. If UNSAT, they're equivalent and can be merged.
3. For gates that proved to be equalent, merge them.
4. Rebuild the dfs list.

At last the program crashes while rebuilding the dfs list. Maybe there's trouble when merging gates...