Problem 5: Latch Correspondence

2005 ICCAD/SIGDA CADathlon

1. Introduction

Formal verification tools have evolved from academic toy applications to full−fledged industrial deployment over the course of a decade. The invention of many clever algorithms and data structures have led to this success. One of them is van Eijk's algorithm that determines the maximal correspondence relation over signals in a sequential logic circuit. Once this relation is known it often precludes an expensive reachable state computation to prove equivalence of two designs or to prove a safety property.

2. Problem statement

2.1. Brief description

Your task is to determine the number of equivalence classes among the memory elements (latches or flip−flops) of a sequential circuit using the method as outlined in the paper. Note that this method is not complete: there might be equivalences that go by undiscovered!

2.2. Input/output specification

The circuit is represented as a number of next−state equations, one for each memory element. You may assume that the initial state (or reset value) of all memory elements is the logic value 0 (Boolean false). We do not consider the extended equivalence−module−inversion here! You may assume that the actually supplied input conforms to the format as specified below. A parser/reader for the input format will be provided. Also, the parse tree data structure and a simple relation data structure are provided. The output of your solution program merely consist of the number of equivalence classes.

2.2.1. Input

The next−state function (`NSFunction` in the syntax specification below) is a Boolean function that defines the data input of a memory element. A next−state function is expressed in terms of the current−state variables and the primary input variables. For simplicity, state variables will be denoted by upper−case letters; input variables are denoted by lower−case letters. The input file consists of 1 or more next−state equations each one terminated by a semicolon. For more details, consult the `parser.h` file.

Note that in a logic expression, the (optional) AND−operator '&' binds stronger than the OR−operator '+'. Both are commutative and associative. Note that the AND−operator is optional and may thus also be denoted by juxtaposition of Factors. The NOT−operator '!' has the highest precedence. The constants 0 and 1 denote the truth values false respectively true.

The input format is defined by the following Backus Naur Form syntax:

```
      Input       : NSFunctions .
      NSFunctions : NSFunction [ NSFunctions ] .
      NSFunction  : "@" S_Variable "=" Expression ";" .
      Expression  : Term [ "+" Expression ] .
      Term        : Factor [ [ "&" ] Term ] .
      Factor      : Primary | "!" Factor .
      Primary     : Constant | Variable | "(" Expression ")" .
      Constant    : "0" | "1" .
      Literal     : S_Variable | I_Variable .
      S_Variable  : "A" | "B" | "C" | ... | 'Z' .
      I_Variable  : "a" | "b" | "c" | ... | 'z' .
```

2.2.1.1. Example input

Here is a typical input file for your program.

```
      # Example input
      @ A = !C;
      @ B = !x;
      @ C = A & B;
      @ D = C & x;

      @ E = F;
      @ F = !(x + F);
      @ G = E & x;
```

2.2.2. Output

You are required to compute the number of equivalence classes of corresponding latches as determined by the method presented in the paper. The output therefore is a single non−negative integer number. Print this for instance using the format descriptor "%d\n".
Note: output should be to standard output (stdout in C, or use cout in C++).

2.2.2.1. Example output

This is the output for the input example given earlier.

```
      7
```

3. Additional information

Work diligently and shoot for obvious and easy solutions. Use simple data structures; for instance for storing the equivalence relation you could opt for a square bit matrix (as already is provided). Efficiency of your program is in this case of lesser importance; make sure it works first. Look at the supplied source files; they include useful material. Look at and run the tests and check their output.

A recommended approach is to start by understanding the available parser, the expression tree data structure and the BDD package interface.

4. References

- Detection of Equivalent State Variables in Finite State Machine Verification, C.A.J. van Eijk, and J.A.G. Jess, Workshop notes of the 1995 IWLS, 1995, pp. 3.35-3.44