May 19, 2018 at 02:31

1. Intro. This shortish program inputs a Boolean circuit in GraphBase format, and generates data by which my SAT solvers are supposed to find a test pattern for a given single-stuck-at fault. (I hacked it from the simpler program SAT-GATES.)

The command line contains the name of the circuit file (e.g., `foo-wires.gb`, produced from `foo.gb` by GATES-TO-WIRES), and the name of the fault to be investigated (e.g., `0X1#2`).

```
#include <stdlib.h>
#include <string.h>
#include "gb_graph.h"
#include "gb_gates.h"
#include "gb_save.h"
  main(int argc, char *argv[])
  {
    register int j, k;
    register Graph *g;
    register Vertex *u, *v, *w, *fault;
    register Arc *a;
    ⟨Process the command line 2⟩;
    ⟨Check for improper wire names 3*⟩;
    ⟨Locate the faulty wire 4⟩;
    ⟨Emit clauses for the wires 5*⟩;
    ⟨Emit clauses for the outputs 12⟩;
  }
```

2. ⟨Process the command line 2⟩ ≡

```
  if (argc ≠ 3) {
    fprintf(stderr, "Usage:␣%s␣foo-wires.gb␣fault\n", argv[0]);
    exit(−1);
  }
  g = restore_graph(argv[1]);
  if (¬g) {
    fprintf(stderr, "I␣couldn't␣reconstruct␣graph␣%s!\n", argv[1]);
    exit(−2);
  }
  if (argv[2][0] ≠ '0' ∧ argv[2][0] ≠ '1') {
    fprintf(stderr, "The␣fault␣name␣should␣begin␣with␣0␣or␣1!\n");
    exit(−3);
  }
```

This code is used in section 1.

3* If necessary, I could rename vertices whose name is empty or too long. But I don't want to bother with that unless it proves to be necessary.

```
#define prime_char '\''
#define sharp_char '#'

⟨ Check for improper wire names 3* ⟩ ≡
  for (j = 0, v = g→vertices; v < g→vertices + g→n; v++) {
    ⟨ Shorten the name if necessary 13* ⟩;
    for (k = 0; v→name[k]; k++)
      if (v→name[k] < '!' ∨ v→name[k] > '~') break;
    if (v→name[0] ≡ '~' ∨ k ≡ 0 ∨ v→name[k]) {
      fprintf(stderr, "Sorry,␣the␣wire␣name␣`%s'␣is␣illegal!\n", v→name);
      j = 1;
    } else if (k > 8) {
      fprintf(stderr, "Sorry,␣the␣wire␣name␣`%s'␣is␣too␣long!\n", v→name);
      j = 1;
    } else if (v→name[k − 1] ≡ prime_char ∨ v→name[k − 1] ≡ sharp_char) {
      fprintf(stderr, "Sorry,␣I␣don't␣like␣the␣last␣character␣of␣the␣wire␣name␣`%s'!\n",
          v→name);
      j = 1;
    } else if (v→name[0] ≡ '_' ∧ v→name[1] ≡ '_') {
      fprintf(stderr, "Sorry,␣I'm␣reserving␣wire␣names␣that␣begin␣with␣`__'!\n");
      j = 1;
    }
  }
  if (j) exit(−3);
```

This code is used in section 1.

4.

```
⟨ Locate the faulty wire 4 ⟩ ≡
  for (v = g→vertices, fault = Λ; v < g→vertices + g→n; v++) {
    if (strcmp(v→name, argv[2] + 1) ≡ 0) {
      fault = v; break;
    }
  }
  if (¬fault) {
    fprintf(stderr, "Sorry,␣I␣can't␣find␣a␣wire␣named␣`%s'!\n", argv[2] + 1);
    exit(−4);
  }
```

This code is used in section 1.

5* A wire is "tarnished" if it is the faulty wire or if one of its operands is tarnished. Tarnished wires *g* are represented by three variables in the output, namely *g* and *g*`'` and *g*`#`; variable *g*`'` denotes the value computed when the fault is present, while *g*`#` denotes a wire on the "active path" from the fault location to an output.

Untarnished wires implicitly have *g*`'` identical to *g* and *g*`#` false.

When *g*`#` is true we ensure that $g \neq g$`'`.

Wires that fan out from a common source actually share the same name, except for their "active" variables. For example, the three wires `Q`, `Q#1`, `Q#2` generate at most five variables `Q`, `Q'`, `Q#`, `Q#1#`, `Q#2#`, not nine. The base variable name is called the wire's *ename*.

```
#define tarnished x.I
#define ename u.S
```

⟨Emit clauses for the wires 5*⟩ ≡

```
printf("~␣sat-gates-stuck-namekludge␣%s␣%s\n", argv[1], argv[2]);
for (v = g→vertices; v < g→vertices + g→n; v++) {
  v→tarnished = 0;    /* innocent until proved guilty */
  switch (v→typ) {
  case 'I': break;
  case '~': ⟨Handle a NOT gate 6⟩; break;
  case '&': ⟨Handle an AND gate 7⟩; break;
  case '|': ⟨Handle an OR gate 8⟩; break;
  case '^': ⟨Handle an XOR gate 9⟩; break;
  case 'F': ⟨Handle a fanout gate 10⟩; break;
  default: fprintf(stderr, "Sorry,␣I␣don't␣know␣to␣handle␣type␣'%c'␣(wire␣%s)!\n", (int)
        v→typ, v→name);
    exit(−666);
  }
  if (v→typ ≠ 'F') {
    v→ename = v→name;
    if (v→tarnished) {
      printf("~%s%c␣~%s␣~%s%c\n", v→name, sharp_char, v→name, v→name, prime_char);
      printf("~%s%c␣%s␣%s%c\n", v→name, sharp_char, v→name, v→name, prime_char);
    }
  }
  if (v ≡ fault) ⟨Initiate the fault scenario 11⟩;
}
```

This code is used in section 1.

6. ⟨ Handle a NOT gate 6 ⟩ ≡

```
if (v→arcs ≡ Λ ∨ v→arcs→next ≠ Λ) {
  fprintf(stderr, "The␣NOT␣gate␣%s␣should␣have␣only␣one␣argument!\n", v→name);
  exit(−10);
}
u = v→arcs→tip;
printf("%s␣%s\n", v→name, u→ename);
printf("~%s␣~%s\n", v→name, u→ename);
if (u→tarnished) {
  v→tarnished = 1;
  printf("%s%c␣%s%c\n", v→name, prime_char, u→ename, prime_char);
  printf("~%s%c␣~%s%c\n", v→name, prime_char, u→ename, prime_char);
  printf("~%s%c␣%s%c\n", u→name, sharp_char, v→name, sharp_char);
}
```

This code is used in section 5*.

7. ⟨ Handle an AND gate 7 ⟩ ≡

```
for (a = v→arcs; a; a = a→next) {
  u = a→tip;
  printf("~%s␣%s\n", v→name, u→ename);
  if (u→tarnished) {
    v→tarnished = 1;
    printf("~%s%c␣%s%c\n", u→name, sharp_char, v→name, sharp_char);
  }
}
printf("%s", v→name);
for (a = v→arcs; a; a = a→next) printf("␣~%s", a→tip→ename);
printf("\n");
if (v→tarnished) {
  for (a = v→arcs; a; a = a→next) {
    u = a→tip;
    if (u→tarnished) printf("~%s%c␣%s%c\n", v→name, prime_char, u→ename, prime_char);
    else printf("~%s%c␣%s\n", v→name, prime_char, u→ename);
  }
  printf("%s%c", v→name, prime_char);
  for (a = v→arcs; a; a = a→next) {
    u = a→tip;
    if (u→tarnished) printf("␣~%s%c", u→ename, prime_char);
    else printf("␣~%s", u→ename);
  }
  printf("\n");
}
```

This code is used in section 5*.

8. ⟨ Handle an OR gate 8 ⟩ ≡

```
for (a = v→arcs; a; a = a→next) {
  u = a→tip;
  printf("%s␣~%s\n", v→name, u→ename);
  if (u→tarnished) {
    v→tarnished = 1;
    printf("~%s%c␣%s%c\n", u→name, sharp_char, v→name, sharp_char);
  }
}
printf("~%s", v→name);
for (a = v→arcs; a; a = a→next) printf("␣%s", a→tip→ename);
printf("\n");
if (v→tarnished) {
  for (a = v→arcs; a; a = a→next) {
    u = a→tip;
    if (u→tarnished) printf("%s%c␣~%s%c\n", v→name, prime_char, u→ename, prime_char);
    else printf("%s%c␣~%s\n", v→name, prime_char, u→ename);
  }
  printf("~%s%c", v→name, prime_char);
  for (a = v→arcs; a; a = a→next) {
    u = a→tip;
    if (u→tarnished) printf("␣%s%c", u→ename, prime_char);
    else printf("␣%s", u→ename);
  }
  printf("\n");
}
```

This code is used in section 5*.

9. I could handle XORs of any length. But I don't want to do that unless it's important, because it would involve generating new names for intermediate gates.

```
⟨ Handle an XOR gate 9 ⟩ ≡
  for (k = 0, a = v→arcs; a; a = a→next) k++;
  if (k ≠ 2) {
    fprintf(stderr, "Sorry,␣I␣do␣XOR␣only␣of␣two␣operands,␣not␣%d␣(gate␣%s)!\n", k, v→name);
    exit(−5);
  }
  u = v→arcs→tip, w = v→arcs→next→tip;
  printf("~%s␣%s␣%s\n", v→name, u→ename, w→ename);
  printf("~%s␣~%s␣~%s\n", v→name, u→ename, w→ename);
  printf("%s␣~%s␣%s\n", v→name, u→ename, w→ename);
  printf("%s␣%s␣~%s\n", v→name, u→ename, w→ename);
  if (u→tarnished) {
    v→tarnished = 1;
    printf("~%s%c␣%s%c\n", u→name, sharp_char, v→name, sharp_char);
    if (w→tarnished) {
      printf("~%s%c␣%s%c\n", w→name, sharp_char, v→name, sharp_char);
      printf("~%s%c␣~%s%c\n", u→name, sharp_char, w→name, sharp_char);
      printf("~%s%c␣%s%c␣%s%c\n", v→name, prime_char, u→ename, prime_char, w→ename, prime_char);
      printf("~%s%c␣~%s%c␣~%s%c\n", v→name, prime_char, u→ename, prime_char, w→ename, prime_char);
      printf("%s%c␣~%s%c␣%s%c\n", v→name, prime_char, u→ename, prime_char, w→ename, prime_char);
      printf("%s%c␣%s%c␣~%s%c\n", v→name, prime_char, u→ename, prime_char, w→ename, prime_char);
    } else {
      printf("~%s%c␣%s%c␣%s\n", v→name, prime_char, u→ename, prime_char, w→ename);
      printf("~%s%c␣~%s%c␣~%s\n", v→name, prime_char, u→ename, prime_char, w→ename);
      printf("%s%c␣~%s%c␣%s\n", v→name, prime_char, u→ename, prime_char, w→ename);
      printf("%s%c␣%s%c␣~%s\n", v→name, prime_char, u→ename, prime_char, w→ename);
    }
  } else if (w→tarnished) {
    v→tarnished = 1;
    printf("~%s%c␣%s%c\n", w→name, sharp_char, v→name, sharp_char);
    printf("~%s%c␣%s%c␣%s\n", v→name, prime_char, w→ename, prime_char, u→ename);
    printf("~%s%c␣~%s%c␣~%s\n", v→name, prime_char, w→ename, prime_char, u→ename);
    printf("%s%c␣~%s%c␣%s\n", v→name, prime_char, w→ename, prime_char, u→ename);
    printf("%s%c␣%s%c␣~%s\n", v→name, prime_char, w→ename, prime_char, u→ename);
  }
```

This code is used in section 5*.

10. ⟨ Handle a fanout gate 10 ⟩ ≡

```
if (¬v→arcs ∨ v→arcs→next) {
   fprintf(stderr, "Eh?␣A␣fanout␣gate␣should␣have␣a␣unique␣parent!\n");
   exit(−6);
}
u = v→arcs→tip;
v→ename = u→ename;
v→tarnished = u→tarnished;
if ((v − 1)→typ ≡ 'F' ∧ (v − 1)→arcs→tip ≡ u) {
   if (v→tarnished)
      printf("~%s%c␣%s%c␣%s%c\n", u→name, sharp_char, (v − 1)→name, sharp_char, v→name, sharp_char);
} else if ((v + 1)→typ ≠ 'F' ∨ (v + 1)→arcs→tip ≠ u) {
   fprintf(stderr, "Eh?␣Fanout␣gates␣should␣occur␣in␣pairs!\n");
   exit(−7);
}
```

This code is used in section 5*.

11. ⟨ Initiate the fault scenario 11 ⟩ ≡

```
{
   v→tarnished = 1;
   printf("%s%s%c\n", argv[2][0] ≡ '0' ? "~" : "", v→ename, prime_char);
   printf("%s%s\n", argv[2][0] ≡ '0' ? "" : "~", v→ename);
   printf("%s%c\n", v→name, sharp_char);
}
```

This code is used in section 5*.

12. Here we conclude by emitting $k + 2$ clauses to force an active path, if there are k tarnished outputs. The first and last of these clauses can obviously be simplified; but we let the solver do that.

(I could have simply output a single clause of length k. But I prefer to stick to 3SAT.)

⟨ Emit clauses for the outputs 12 ⟩ ≡

```
printf("__0\n");      /* auxiliary variables begin with "__" */
for (k = 0, a = g→outs; a; a = a→next) {
   u = a→tip;
   if (u→tarnished) {
      printf("%s%c␣~__%d␣__%d\n", u→name, sharp_char, k, k + 1);
      k++;
   }
}
printf("~__%d\n", k);
```

This code is used in section 1.

13*. Here a name like C34:13#19 becomes C341319. In general I change all numbers to two digits, and delete the colons and sharp signs.

⟨ Shorten the name if necessary 13* ⟩ ≡

```
if (v→name[0] < 'A' ∨ v→name[0] > 'Z') {
  fprintf(stderr, "Vertex␣name␣%s␣didn't␣start␣with␣a␣code␣letter!\n", v→name);
  j = 1;
}
{
  int i1, i2, i3;
  register i, d;
  for (i = 1, d = 0; v→name[i] ≥ '0' ∧ v→name[i] ≤ '9'; i++) d = 10 * d + v→name[i] − '0';
  if (d > 99) {
    fprintf(stderr, "Vertex␣name␣%s␣has␣a␣number␣>␣99!\n", v→name);
    j = 1;
  }
  i1 = d;
  if (v→name[i] ≠ ':') goto okay;
  for (i++, d = 0; v→name[i] ≥ '0' ∧ v→name[i] ≤ '9'; i++) d = 10 * d + v→name[i] − '0';
  if (d > 99) {
    fprintf(stderr, "Vertex␣name␣%s␣has␣a␣number␣>␣99!\n", v→name);
    j = 1;
  }
  i2 = d;
  if (v→name[i] ≠ '#') goto okay;
  for (i++, d = 0; v→name[i] ≥ '0' ∧ v→name[i] ≤ '9'; i++) d = 10 * d + v→name[i] − '0';
  if (d > 99) {
    fprintf(stderr, "Vertex␣name␣%s␣has␣a␣number␣>␣99!\n", v→name);
    j = 1;
  }
  i3 = d;
  if (v→name[i]) {
    fprintf(stderr, "Vertex␣name␣%s␣has␣unexpected␣structure!\n", v→name);
    j = 1;
  } else if (i < 8) goto okay;
  else sprintf(v→name + 1, "%02d%02d%02d", i1, i2, i3);
okay: ;
}
```

This code is used in section 3*.

14* Index.

The following sections were changed by the change file: 3, 5, 13, 14.

SAT-GATES-STUCK-NAMEKLUDGE