Testing and Qualification of Optimizing Compilers for Functional Safety

José Luis March Cabrelles, PhD
Solid Sands B.V.

- Based in Amsterdam, the Netherlands
- Founded in 2014
- The one-stop shop for C and C++ compiler and library testing, validation and safety services
- SuperTest
1) Introduction to SuperTest

2) Functional Safety for Compilers
   • Types of Compiler Errors

3) Optimizations

4) Conclusions
SuperTest

Test Driver

Test Reporter

Huge Collection of Hand-Written Tests

Gigantic Collection of Generated Tests
Requirements Traceability

The V-Model

ISO C/C++ Language Specification

Implementation

Unit Tests

Evidence

Validation

SuperTest™

Requirements Traceability
3.3.17 Comma operator

Syntax

expression:
assignment-expression
expression, assignment-expression

Semantics

The left operand of a comma operator is evaluated as a void expression; there is a sequence point after its evaluation. Then the right operand is evaluated; the result has its type and value.
Testing the Comma Operator

```c
void ge( int *p ){
    *p = 2;
}

int test_it( void ){
    int a, *p, r;
    p = &a;
    r = ( ge(p), a++, a+=3, a+=8, a+8 );
    return r == 22;
}
```

/* SuperTest-suite/3/3/17/t2.c */
3.3.13 Logical AND operator

Syntax

```
logical-AND-expression:
  inclusive-OR-expression
logical-AND-expression && inclusive-OR-expression
```

Constraints

Each of the operands shall have scalar type.

Semantics

The `&&` operator shall yield 1 if both of its operands compare unequal to 0, otherwise it yields 0. The result has type int.
Testing Operand Types

```c
struct x {
    int i;
} X;

int test_it( int i ){
    return i && X;
}
```

```bash
$ gcc -c x0.c
x0.c: In function 'test_it':
x0.c:6:11: error: invalid operands to binary && (have 'int' and 'struct x')
    return i && X;
       ^~
```

/* SuperTest/suite/3/3/13/x0.c */
Outline

1) Introduction to SuperTest

2) Functional Safety for Compilers
   • Types of Compiler Errors

3) Optimizations

4) Conclusions
Compiler Qualification

- ISO 26262, Part 8, Section 11
  - Confidence in the use of software tools

- Goal: Develop confidence in the compiler
  - Verification against language specification
  - Mitigations for compiler failures
  - Specific use case

- Established practice for the automotive industry
  - See also: Rail, Nuclear, Aviation, Medical
Compiler Errors

There are different types:

1) Compile Time Errors
2) Diagnostic Errors
3) Mitigable Runtime Errors
4) Really Bad Runtime Errors
1) Compile Time Error

```cpp
constexpr int function( int x )
{
    class A {
    public:
        /* Diagnostic Expected */
        constexpr A() : value(x) {}
        int value;
    };
    A a;
    return 0;
}

int main()
{
    constexpr int variable = function( 1 );
    return 0;
}
```

LLVM 3.9 Crash
2) Diagnostic Error

```c
#include <stdio.h>

int test( void ){
    /* Not strictly conforming */
    return 3 ? : 7;
}

int main( void ){
    printf( "%d\n", test() );
    return 0;
}

/* SuperTest/suite/3/3/15/xspr6112.c */
3) Mitigable Runtime Error

#include <stdio.h>

typedef struct { int phone; int fax; } Contact;
typedef struct { int addr; Contact pf; } House;

int main( void ){
    /* SuperTest/suite/C99/6/7/8/t7.c */

    Contact generic = { .phone = 998,
                        .fax = 999 };
    House home = { 501, .pf = generic,
                   .pf.phone = 650 };

    printf("Phone (650): %d\n", home.pf.phone); // OK: 650
    printf("Fax (999): %d\n", home.pf.fax ); // Error: 0
}
s[0] = 42;  /* SuperTest-suite/3/5/7/tspr2388.c */
*(sp[0]) = -1;  /* *(sp[0]) is an alias of s[0] */
printf("%d", s[0]);  /* Incorrectly prints 42 */

• **Optimization Error**

• No optimization specified and no option to turn this off

• It is not linked to a specific syntactical feature
Outline

1) Introduction to SuperTest

2) Functional Safety for Compilers
   • Types of Compiler Errors

3) Optimizations

4) Conclusions
How to Test Optimizations?

• Optimizations are non-functional requirements
  • Not even mentioned in the language specification

• Benchmarks: Not a good idea
  • Results not verified
  • Undefined Behavior
  • No different data models
  • Not all generated code is executed
int f( int n ){
    int total = 0;
    for(int i = 0; i < n; i++){
        total += i & n;
    }
    return total;
}

• Unit Testing: f(999)
• Full coverage at source code
• Compiled at -O0
• Full coverage at assembly level
int f(int n) {
    int total = 0;
    for(int i=0; i<n; i++) {
        total += i & n;
    }
    return total;
}

• Compile with -Ofast
• Unit Testing with f(999): About 80% coverage at assembly level
• Full structural coverage: 5 tests needed
• Full branch coverage: Not possible

Coverage with Optimizations
New Optimization Test Suite

• Maximum code and branch coverage for 3 compilers
• Based on typical optimizations and combinations
• Compute a verifiable result
• Free of Undefined Behavior for different data models
```
void loop( int *a, int *b ){
    for( int i = 0; i < 5; i++ ){
        if( a[i] <= 0 ){
            a[i] = 0;
        }else{
            a[i] = b[i];
        }
    }
}

void test_it(){
    print_values( "a before:" , a);
    print_values( "b before:" , b);
    loop(a, b);
    print_values( "a  after:" , a);
}
```

```bash
$ sh no_optimizations.sh
a before: 0 1 2 3 4
b before: 1 2 3 4 5
a after: 0 2 3 4 5
```

```bash
$ sh optimizations.sh
a before: 0 1 2 3 4
b before: 1 2 3 4 5
a after: 0 0 2 0 4
```

Really Bad
1) Introduction to SuperTest

2) Functional Safety for Compilers
   • Types of Compiler Errors

3) Optimizations

4) Conclusions
Conclusions

• No compiler is perfect

• Be aware of compiler weak points in safety-critical

• SuperTest is useful for compiler developers and users

• Verify test suites used by your compiler supplier
Thank You!

José Luis March Cabrelles
joseluis@soldsands.nl
www.solidsands.nl