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

Outline

1) Introduction to SuperTest

2) Functional Safety for Compilers
   - Types of Compiler Errors

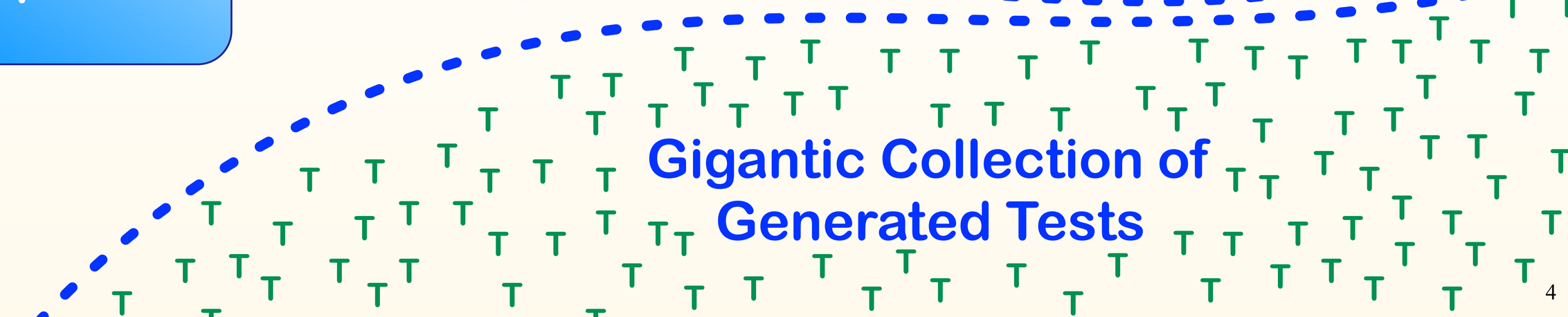
3) Optimizations

4) Conclusions

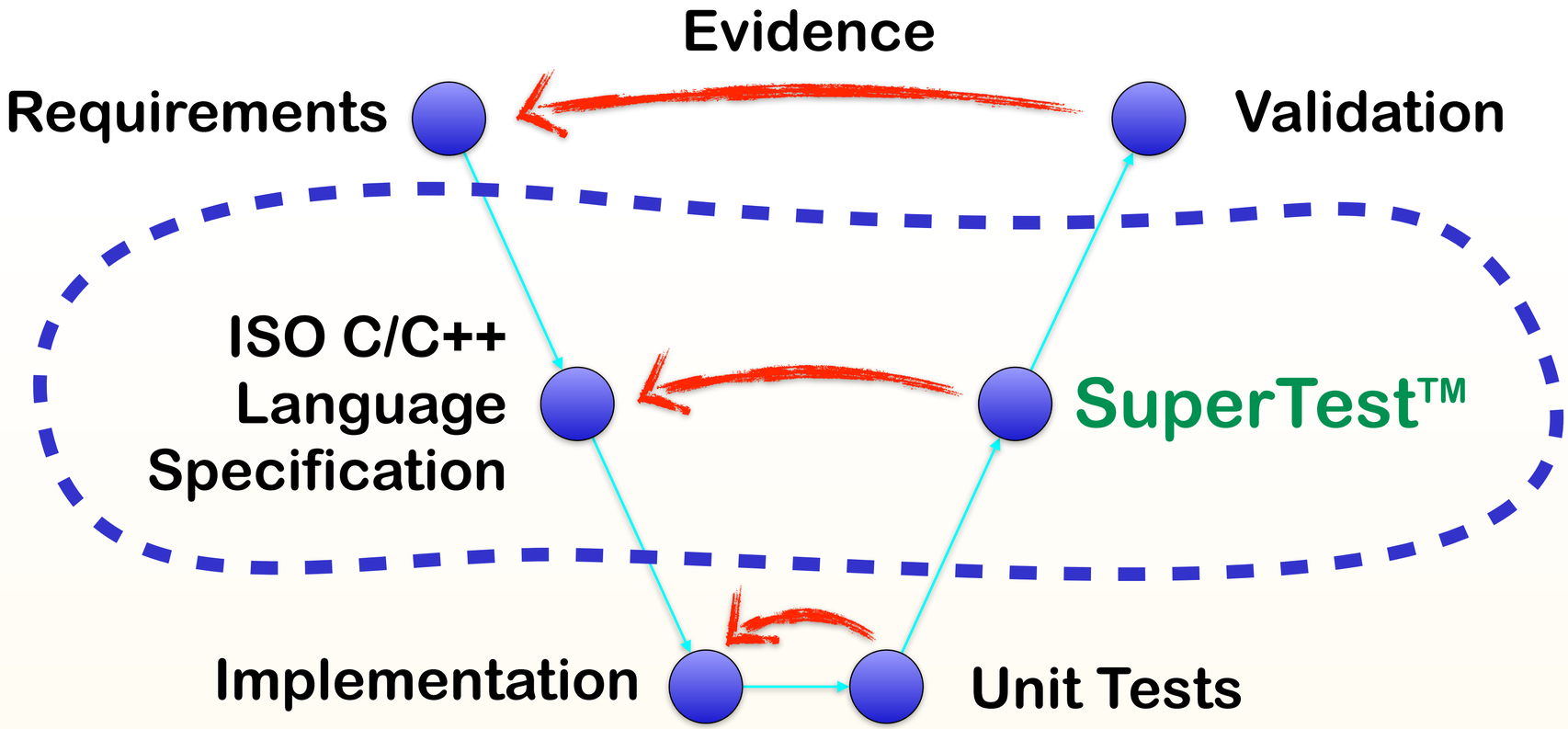
SuperTest

Test Driver

Test Reporter



Requirements Traceability



The V-Model

ISO C90 Examples

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./43/

Testing the Comma Operator

```
void ge( int *p ){

    *p = 2;
}
```

/* SuperTest/suite/3/3/17/t2.c */

```
int test_it( void ){

    int a, *p, r;

    p = &a;
    r = ( ge(p), a++, a+=3, a+=8, a+8 );

    return r == 22;
}
```

Non-Conformance and Diagnostics

```
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

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

int test_it( int i ){

     return i && X;
}
```

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

```
$ 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;
         ~ ^~
```

Outline

1) Introduction to SuperTest

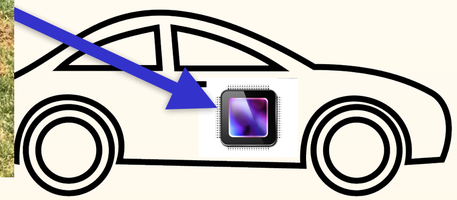
2) Functional Safety for Compilers
   - Types of Compiler Errors

3) Optimizations

4) Conclusions

Complexity and Importance

App Source Code



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

```
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

/* SuperTest/suite/Cxx14/7/1/5/xclangcrash.C */

2) Diagnostic Error

```
#include <stdio.h>

int test( void ){

    /* Not strictly conforming */
    return 3 ? : 7;
}
                              /*  SuperTest/suite/3/3/15/xspr6112.c  */
int main( void ){

    printf( “%d\n”, test() );
    return 0;
}
```

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 };
                                                  // GCC
    printf("Phone (650): %d\n", home.pf.phone); // OK:  650
    printf("Fax   (999): %d\n", home.pf.fax  ); // Error: 0
}
```

4) Really Bad Runtime Error

```
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

Coverage without Optimizations

```
int f( int n ){
    int total = 0;
    for(int i = 0; i < n; i++){
        total += i & n;
    }
    return total;
}
```

```
+:  push   rbp
+:  mov    rsp,rbp
+:  mov    edi,-0x4(rbp)
+:  movl   0x0,-0x8(rbp)
+:  movl   0x0,-0xc(rbp)
+:  mov    -0xc(rbp),eax
+:  cmp    -0x4(rbp),eax
+:  jge    0x40051b <f+0x3b>
+:  mov    -0xc(rbp),eax
+:  and    -0x4(rbp),eax
+:  add    -0x8(rbp),eax
+:  mov    eax,-0x8(rbp)
+:  mov    -0xc(rbp),eax
+:  add    0x1,eax
+:  mov    eax,-0xc(rbp)
+:  jmpq   0x4004f5 <f+0x15>
+:  mov    -0x8(rbp),eax
+:  pop    rbp
+:  retq
```

- Unit Testing: f(999)
- Full coverage at source code
- Compiled at -O0
- Full coverage at assembly level

Coverage with Optimizations


```
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

```
+:  test   %edi,%edi
v:  jle    0x400552 <loop+0x12>
+:  xor    %edx,%edx
+:  cmp    $0x7,%edi
>:  ja     0x400555 <loop+0x15>
-:  xor    %eax,%eax
-:  jmpq   0x400660 <loop+0x120>
-:  xor    %eax,%eax
-:  retq
+:  mov    %edi,%ecx
+:  and    $0xfffffff8,%ecx
+:  mov    $0x0,%eax
v:  je     0x400660 <loop+0x120>
+:  movd   %edi,%xmm0
+:  pshufd $0x0,%xmm0,%xmm0
+:  lea    -0x8(%rcx),%edx
+:  mov    %edx,%eax
+:  shr    $0x3,%eax
+:  bt     $0x3,%edx
>:  jb     0x4005aa <loop+0x6a>
-:  movdqa 0x17c(%rip),%xmm1
-:  pand   %xmm0,%xmm1
-:  movdqa 0x180(%rip),%xmm3
-:  pand   %xmm0,%xmm3
-:  movdqa 0x184(%rip),%xmm5
-:  mov    $0x8,%edx
-:  test   %eax,%eax
-:  jne    0x4005c0 <loop+0x80>
-:  jmpq   0x400637 <loop+0xf7>
+:  pxor   %xmm1,%xmm1
+:  movdqa 0x14a(%rip),%xmm5
+:  xor    %edx,%edx
+:  pxor   %xmm3,%xmm3
+:  test   %eax,%eax
v:  je     0x400637 <loop+0xf7>
+:  mov    %ecx,%eax
+:  sub    %edx,%eax
+:  movdqa 0x163(%rip),%xmm8
+:  movdqa 0x16a(%rip),%xmm9
+:  movdqa 0x172(%rip),%xmm6
+:  movdqa 0x17a(%rip),%xmm7
+:  nopw   %cs:0x0(%rax,%rax,1)
+:  movdqa %xmm5,%xmm2
+:  paddd  %xmm8,%xmm2
+:  movdqa %xmm5,%xmm4
+:  pand   %xmm0,%xmm4
+:  pand   %xmm0,%xmm2
+:  paddd  %xmm1,%xmm4
+:  paddd  %xmm3,%xmm2
+:  movdqa %xmm5,%xmm1
+:  paddd  %xmm9,%xmm1
+:  movdqa %xmm5,%xmm3
+:  paddd  %xmm6,%xmm3
+:  pand   %xmm0,%xmm1
+:  pand   %xmm0,%xmm3
+:  paddd  %xmm4,%xmm1
+:  paddd  %xmm2,%xmm3
+:  paddd  %xmm7,%xmm5
+:  add    $0xfffffff0,%eax
+:  jne    0x4005f0 <loop+0xb0>
+:  paddd  %xmm3,%xmm1
+:  pshufd $0x4e,%xmm1,%xmm0
+:  paddd  %xmm1,%xmm0
+:  pshufd $0xe5,%xmm0,%xmm1
+:  paddd  %xmm0,%xmm1
+:  movd   %xmm1,%eax
+:  cmp    %edi,%ecx
+:  mov    %ecx,%edx
v:  je     0x40066c <loop+0x12c>
+:  nopw   0x0(%rax,%rax,1)
+:  mov    %edx,%ecx
+:  and    %edi,%ecx
+:  add    %ecx,%eax
+:  inc    %edx
+:  cmp    %edx,%edi
+:  jne    0x400660 <loop+0x120>
+:  retq
```

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

Optimization Errors: Embedded ARM

```
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);
}
```

```
$ 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
```

```
$ 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

Outline

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@solidsands.nl
www.solidsands.nl