

Introduction to the C/C++ to TTCN-3 mapping

Andreas Nyberg (Nokia/Helsinki) & Matti Kärki (VTT/Oulu)

TTCN-3 User Conference

Berlin, 31 May-2 June 2006

1

NOKIA
Connecting People

Contents

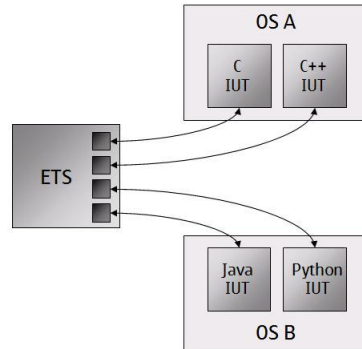
- Background
- Approach
- Language constructs covered by the mapping
- C mappings, “*pointer handling*”
- C++ mappings, “*object-orientation*”
- Conclusions

2 © 2005 Nokia T3UC2006.ppt // ANy

NOKIA
Connecting People

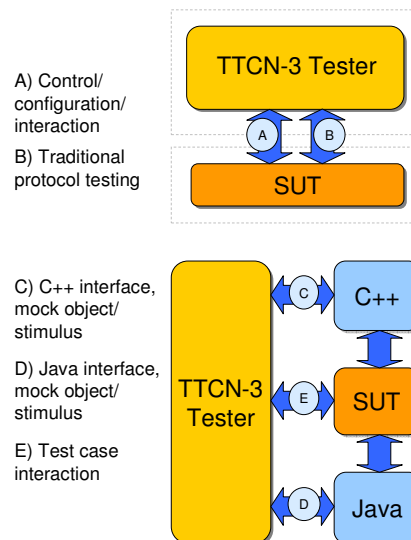
Background

- New test domains for TTCN-3 testing; based on procedural and object-oriented programming languages requires a mapping of the language under test in to TTCN-3
 - Procedure-based communication can be used for direct interfacing to software modules
 - Testing can be applied in an earlier phase (unit testing)
 - One test language for testing of SUTs in different programming languages, same test suite, same test case
 - Combining traditional TTCN-3 testers with additional direct software interaction, stimulus, mock objects
 - Controlling/configuring SUTs, preamble/postamble



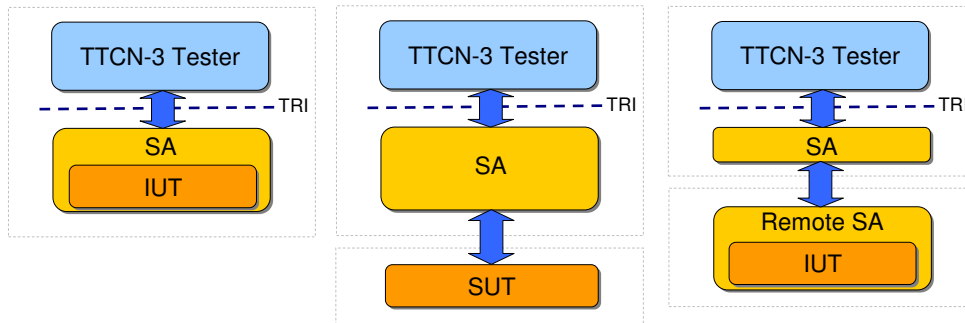
The need for a mapping

- Test architecture possibilities, many combinations
 - Internal interfaces
 - Many programming languages
- Callable interface needs to be represented in TTCN-3
 - Functions
 - Types
- Mapping must provide same operational semantics as mapped language



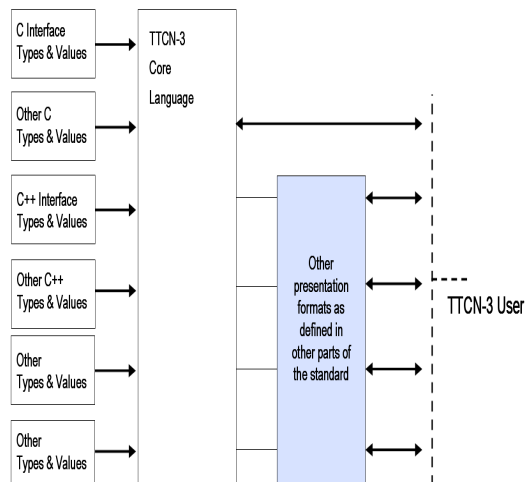
Historical background

- Two independent projects at Nokia/Helsinki and VTT/Oulu. Outcome was a combined mapping proposal, C/C++ to TTCN-3.
- Test system architectures to verify mappings with where based on remote SUT, remote SA and a 'tight' IUT, all had different ways of handling memory access.
- Procedure-based communication had not been frequently used. Immature TTCN-3 tools and non-existing or unsatisfying test system components.



Choice of mapping languages

- C and C++ is together the largest preferred programming language in the world, ~28% (www.tiobe.com/tpci.htm)
- C / C++ suitable for software module testing/mapping exploration; strong typing, OO-language, pointers
- C++ does not impose a single rooted inheritance hierarchy
- Neither C or C++ can be mapped in full in an un-ambiguous way



Known problematic issues

TTCN-3

Does **NOT** have

- Object-oriented concepts
- Overloading
- Generic constructs eq. C++ templates, Java generics

C/C++

- Mapping input
- Pointers
`int*`
- Pointer arithmetic
`ptr++`
- Variadic functions
`printf(const char *FORMAT, ...)`
- Type casting
`(int*)shortPtr`
- Address of value (`&`)

C++

- Inheritance
- Polymorphism
- Templates
- Standard library items
`string`, `vector<int>`

What is covered by the mappings

Covered C/C++

- Built-in types, structured types
- Enumerated types
- Functions (inline, extern, volatile)
- Global variables
- Pointer types
- Dereferencing/address of variable
- Typedef

Covered C++

- Encapsulation
- Inheritance
- Polymorphism
- Operators
- References

Proposed mappings/Normative (C/C++)

- Variadic parameter lists
- Templates
- Preprocessor directives

Not covered

- Address of variable of built-in types
- Type casting, built in types
- Access specifiers

Mapping input

- Preferred mapping input complete translation unit i.e. preprocessed source file
- Does not necessarily only have to be a preprocessed source file
 - Macro definitions (`#define MAX_SZ 1024, #define KEY "myKey"`)
- Items looked for in the source code are items of the interface under test and additional items useful for test suite implementations
 - Functions
 - Type definitions
 - Enumerated types
 - Global variables

Mapping examples

- Pointers
`int* intPtr;`
- Address of variable
`void* ptr = &myObject;`
- Encapsulation
`class A { /* attributes & methods */};`
- Inheritance
`class B: public A { /*...*/};`

Mapping C/C++ to TTCN-3

- Majority of the types can be mapped in a rather straight forward manner
 - `int`→`integer`
 - `float`→`float`
 - `struct`→`record`
 - `class`→`module`
 - `function`→`signature`

```
//int add( int x, int y );
type integer CInt ( c_CIntMin .. c_CIntMax );
signature add( in CInt x, in CInt y ) return CInt;
// . . .
pt_pb.call( add:{ 2,3 }, C_TIMEOUT ) {
  [] pt_pb.getreply( add:? value 5 ) { setverdict( pass ); }
  [] pt_pb.catch( timeout )          { setverdict( fail ); }
}
```

Mapping C to TTCN-3 (pointers)

- Usefulness of an address to a memory location (!NULL)?
- Minimally needed functionality, allocate, initialize, write, read, release
- Pointer arithmetic
- Dereferencing, `*ptr`
- Address of variable, `&inst`

```
// int*
type integer CPtr;
type CPtr CIntPtr;
type record of CInt CIntArr;
```

Mapping C to TTCN-3 contd. (pointer handling)

- External functions or signatures, depends on the test architecture
 - SUT and tester share the same memory space
 - Multiple SUTs
 - Both approaches are proposed in the mapping

```
// malloc, set, get, free
external function ef_MallocCInt (in CUnsignedInt p_Length)
    return CIntPtr;
external function ef_SetCInt (in CIntPtr p_Self,
                             in CIntArr p_CInts);
// . . .
signature s_MallocCInt (in CUnsignedInt p_Length)
    return CIntPtr;
signature s_SetCInt (in CIntPtr p_Self,
                    in CIntArr p_CInts);
// . . .
```

Mapping C to TTCN-3 contd. (pointer handling, by example)

```
// int* ptr = (int*)malloc( sizeof(int)*4 );
// ptr[0]=5;ptr[1]=4;ptr[2]=3;ptr[1]=2;
// useIntArr(ptr);
// if ( ptr[2]==c_oracle ) { ... }

var CIntPtr ptr; var CIntArr intArr;
ptr := f_MallocCInt(4); // allocate
f_SetCInt( ptr, {5,4,3,2} ); // initialize/dereferencing

pt.call( s_useIntArr:{ptr}, c_TIMEOUT ) { // use/write
    [] pt.getreply( s_useIntArr:? ) {}
    [] pt.catch( timeout ) {}
}
pt.call( s_GetCInt:{ptr,4}, c_TIMEOUT ) { // read
    [] pt.getreply( s_GetCInt:? ) -> value intArr { /* if(ptr[2]...*/ }
    // ...
}
```

Mapping C to TTCN-3 contd. (address of variable/object)

- Getting address of variable/object using operator & (not for built-in types)

```
// struct MyStruct obj = retObject();
// useObject( &obj );

type record MyStruct {
  CPtr m_this optional, // instead of operator &
  // attributes ...
}
type CPtr CMyStructPtr;
signature retObject() return MyStruct;
signature useObject( in CMyStructPtr ptr );

var MyStruct obj;
pt.call( retObject: {}, c_TIMEOUT ) {
  [] pt.getreply( retObject:? ) -> value obj {...} // ...
}
if ( obj.m_this != omit ) {
  pt.call( useObject: { obj.m_this }, 2.0 ) {...} // operator &
}
```

Mapping C++ to TTCN-3 (encapsulation)

- struct**, **union** and **class** provides encapsulation in C++
- A **module** provides encapsulation of mapped attributes and member functions in TTCN-3

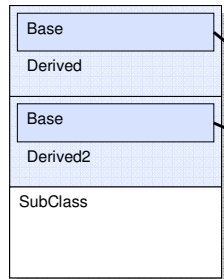
```
class MyClass {
  // Attributes
  . . .
  // Member functions
  void MyFunction();
};

module MyClass {
  type record MyClass_t {
    // Attributes
    . . .
  }
  // Member functions
  signature MyFunction(
    in CppPtr m_this );
  . . .
}
```

Mapping C++ to TTCN-3 contd. (inheritance)

- Inheritance the C++ way

```
class Base {};  
class Derived : public Base {};  
class Derived2 : public Base {};  
class SubClass :  
    public Derived, Derived2 {};
```



- Using TTCN-3 `import`

```
module CppBase { type record CppBase_t  
    {} }  
module CppDerived {  
    import from CppBase all;  
    type record CppDerived_t {  
        CppPtr m_this optional,  
        CppBase_t m_base  
    }  
}  
module CppDerived2 { /* as above */ }  
  
module SubClass {  
    import from CppDerived all;  
    import from CppDerived2 all;  
    type record CppDerived2_t {  
        CppPtr m_this optional,  
        CppDerived_t m_derived,  
        CppDerived2_t m_derived2,  
    }  
}
```

Conclusions

- Approaching new test domains, using the defined mappings we can:
 - directly interact with software interfaces in their native language, enabling a multitude of test system configurations
 - combine traditional TTCN-3 testers with direct internal interaction using procedure-based communication
- C and C++ are very ambiguous languages and there can be alternative mappings to almost everything
- Usage of mappings requires knowledge of mapping background
- Usage will require additional tools, code generator C/C++ -> TTCN-3
- Other OO mappings, different approaches might be needed due to the single inheritance tree model of C++
- YES, some testing might be easier using native language of SUT, TTCN-3 adds additional capabilities

QUESTIONS?

Mapping C++ to TTCN-3 contd. (inheritance, methods)

- Polymorphism, usage of virtual functions

```
class Root {  
    void Function();  
};  
  
class MySuperClass : Root {  
    virtual void VirtualFunction();  
    void MemberFunction();  
};  
  
class MySuperClass2 : Root {  
    void MemberFunction();  
};  
  
class MySubClass :  
    MySuperClass, MySuperClass2 {  
    virtual void VirtualFunction();  
    void MyFunction();  
};
```

- All inherited member functions are overridden

```
module CppMySubClass {  
    // . . .  
    signature s_VirtualFunction(  
        in MySubClassPtr p_this );  
    signature s_MyFunction (  
        in MySubClassPtr p_this );  
  
    // to resolve ambiguity  
    signature s_MySuperClass_MemberFunction  
        (in MySubClassPtr p_this);  
    signature s_MySuperClass2_MemberFunction  
        (in MySubClassPtr p_this);  
    signature s_Root_MySuperClass_Function  
        (in MySubClassPtr p_this);  
    signature s_Root_MySuperClass2_Function  
        (in MySubClassPtr p_this);  
    // . . .  
}  
  
// obj.Root::MySuperClass::Function();  
pt_pb.call(  
    s_Root_MySuperClass_Function:  
    {CppMySubClass.m_this} ) { /* . . . */ }
```

Mapping C++ to TTCN-3 contd. (polymorphism, overloading)

```
void print(int value);           signature s_print_int(
                                in CppInt p_value);

void print(const char* string);  signature s_print_constcharptr(
                                in CppCharPtr p_string);

void print(int value,           signature s_print_int_def(
                                int def=10);           in CppInt p_value);
                                signature s_print_int_defint(
                                in CppInt p_value,
                                in CppInt p_def);
```

Mapping C++ to TTCN-3 contd. (polymorphism, virtual)

```
// Base* base = getObj();
// base->print();

module Base {
  import from CppBuiltInTypes all;
  signature print( in CppPtr m_this ); // virtual function
}
module Derived {
  import from Base all;
  signature Base_print( in CppPtr m_this ); // Base::print();
  signature print( in CppPtr m_this ); // virtual function
}

var CppPtr objPtr := f_getObj();
// polymorphism resolved in SUT
pt.call( Derived.print:{objPtr}, 2.0 ) {
  [] pt.getreply( Derived.print:? ){/*...*/}
  [] pt.catch( timeout ) {}
}
```