Automated Source-code-based Testing of Object-Oriented Software

R. Gerlich[1], R. Gerlich[1], C. Dietrich[2]

Data Systems in Aerospace DASIA 2014

June 3rd, 2014, Warsaw, Poland

[1]Dr. Rainer Gerlich System and Software Engineering BSSE
Immenstaad, Germany
E-Mail: Rainer.Gerlich@bsse.biz
Ralf.Gerlich@bsse.biz

[2]Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR)
Bonn, Germany
E-Mail: Carsten.Dietrich@dlr.de

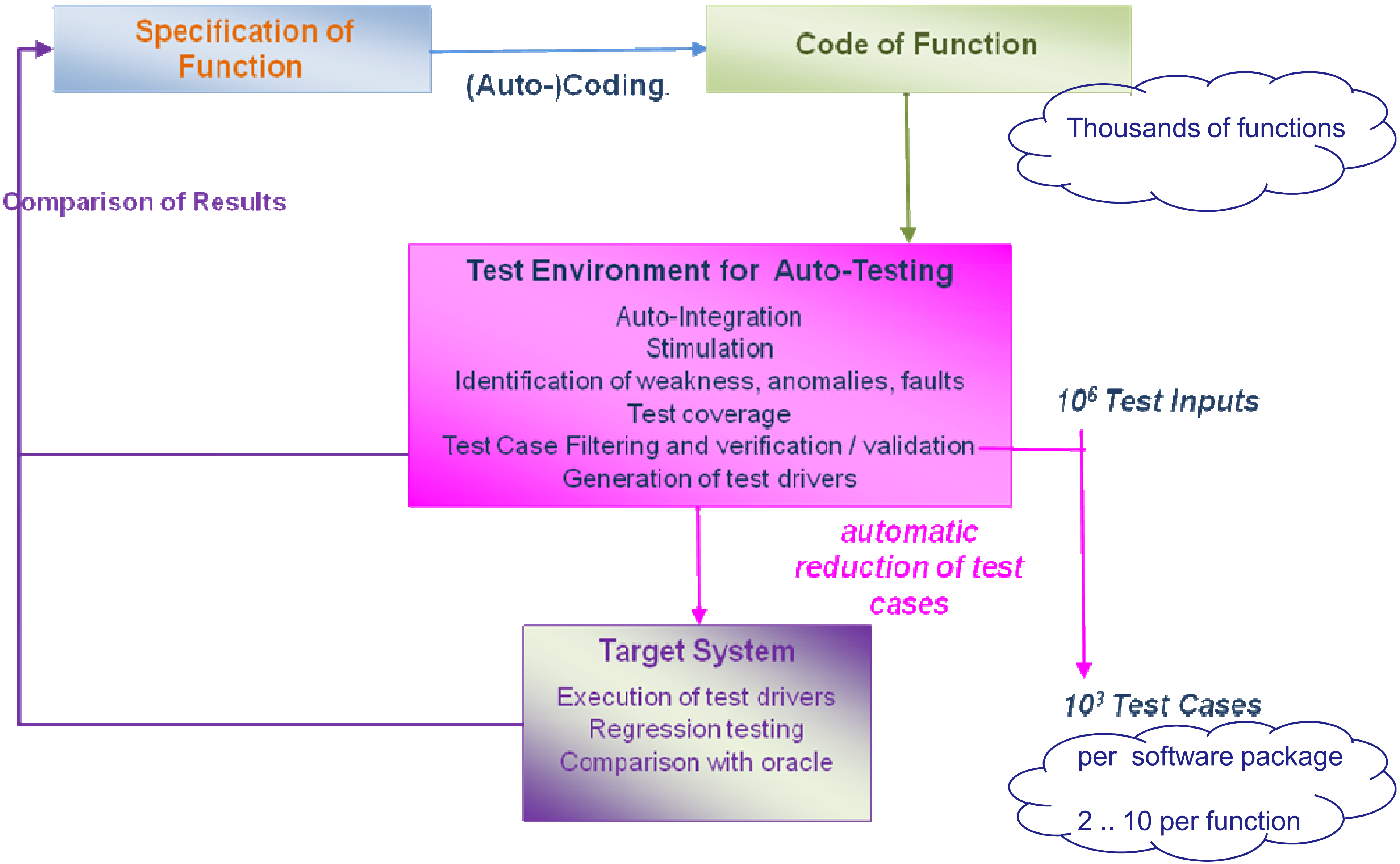
Contents

- Introduction to the FAST Process

  Fully / Flow-optimised Automated, Source-code-based Testing
- Advantages/Disadvantages of C++ in OBSW
- Challenges for automated test of OO-Software
- Consequences and Solutions
- Conclusions

The FAST Test Process

Principal Flow



Advantages of the FAST Process

BSSE System and Software Engineering

- Massive stimulation at low effort/cost
- Automatic Robustness Testing
- Increased probability of finding sporadic defects
- Post-Factum filtering of test-case-candidates instead of Pre-Factum guessing
- Critical Defects found after finalisation of ISVV!
- Testing may start even if system is incomplete

  ⇒ missing functions are derived from prototypes as active stubs

Advantages of the FAST Process

- Analysis yields
  - most faults found with FAST after standard test process are a matter of massive stimulation
  - if the number of automated stimuli is in the range of manually generated

  ⇒no significant difference in faults detected

  (as reported in literature)

C++ in On-Board Software: Pros

- Enforcement of Data Consistency/Fault Isolation
- Stricter type system than C (but nowhere near Ada)
- Templates/Template Libraries
- Implicit Initialisation of Variables (sometimes...)
- Paradigm largely matches UML

C++ in On-Board Software: Cons

- Class Model instead of Component Model
- Template Matching complicated (Turing-complete!)
- Overloading of syntactic elements impacts readability
- Implicit behaviour (copy operators, default constructors) may impact comprehensibility
- Many language elements are inconsistent with each other (e.g. type deduction, behaviour of STL elements, ...)

Basic OO Concepts

- Inheritance/Subtype Polymorphism
- Encapsulation/"Data Hiding"
- Dynamic Dispatch
- Abstraction

The Challenge

Generate stimuli close to operational profile, but properly balance that with robustness testing...

…without any formal information about the programmers‘ intent!

⇒ Heuristics!

Still need to find more of these for OO programming languages

Challenges for Automated Test Data Generation in Context of O-O

- Encapsulation/”Data Hiding”
- Subtype Polymorphism
- Dynamic Dispatch
- Testing Templates
- Stubbing of Constructors
- Use of Design Patterns
- Generation of Regression Test Suites
- Inaccessible Types and Methods

Challenge: Subtype Polymorphism

BSSE System and Software Engineering

```
class A {};

int foo(A* b);
```

Foo expects pointer to A

Substitution principle:
Any subclass of A is applicable as well.

⇒Consider all subclasses of A.

Subclasses may be declared in other compilation units!

Challenge: Dynamic Dispatch

Library:

```
class A {
public:
    virtual void foo();
    void bar() {
        …
        foo();
        …
    }
};
```

Application:

```
class B: public A {
public:
    void foo() {
    }
};
```

Overriding foo() also changes behaviour of bar().

No way to test libraries in isolation!
We can only test full applications!

Challenge: Use of Design Patterns


```
class Singleton {
private:
    Singleton() { … }
public:
    static Singleton* getInstance() {
        if (!instance) {
            instance = new Singleton();
        }
        return instance;
    }
private:
    static Singleton* instance;
};
```

Only constructor is hidden.

Get instance by calling getInstance()

How should the test tool know?

→Heuristics, Annotations

FAST
From Safety to Security

- so far: focus put on safety / embedded systems
- increasing interest in security, also in space area
- security may even be of higher interest for C++ software
- safety vs. security:
  - safety assumes benevolent environment
    Design-by-contract
  - security assumes malevolent environment
    attackers do not respect contracts
  - →“fuzzing”
  - FAST exactly addresses this point

Status so far

- Parsing and instrumenting C++
  - not for the faint of heart!
- Generating test data using constructors and assignment to public data members
- Handling and recording C++ exceptions
- Provision of stubs for missing functions / methods
- On-going tests with software intended for use on ISS

Application software

- Criticality Level C, OS Linux
- Uses templates from STL, Boost, LOKI
- Own templates
- 302 hpp-files, 229 cpp-files
- 53.200 physical lines of code
- 22.600 LOC
- 643 public methods
- 279 classes
- 364 class instantiations
- parsed, instrumented, executed

FAST and C++

- Near-term
  - Gaining advantages by massive stimulation
  - Fault detection driven by assuming remaining faults
- Mid-term
  - Better handling of information hiding
  - Native support for templates
- Long-term
  - Generation of regression test suites

Findings so far

BSSE System and Software Engineering

- Random test data more suited for covering the state space than regularly spaced data points
- Internal enforcement of consistency (by constructors, methods) reduces need for external definition of constraints
- Constraints for sequences of method invocations required (e.g. protocol state machines)

Conclusions

- C++ S/W may be easier to verify (e.g. due to enforcement of consistency)
- OO Features (e.g. dynamic dispatch) heavily increase effort for manual testing
- Only very limited pre-testing of libraries possible
- Increased effectiveness with massive stimulation
- ⇒suggests automatic test approach
- More research required for better heuristics
- Annotations may come from software models (→code generation)

ACKNOWLEDGMENT

This activity was supported by
DLR Space Agency
(Deutsches Zentrum fuer Luft- und Raumfahrt)
on behalf of
BMWi (German Federal Ministry of Economics and Energy)
Reference Number 50 RA 1120

Thank you for your attention!

Questions?

Backup

Inheritance


```
class A {
public:
    A(int aVal):val(aVal) {}
    int getVal() const { return val; }
protected:
    int val;
};

class B: public A {
public:
    B(int aVal): A(aVal) {}
    int getValTimesTwo() const { return 2*val; }
}

int foo() {
    B obj = B(10);
    return obj.getVal();
}
```

A has method getVal() and data member val.

B is subtype of A.

B *inherits* getVal() and val.

Subtype Polymorphism

```
class A { … };
class B: public A { … };

int foo(^* obj) { … };

int bar() {
    A* obj = new B;
    return foo(obj);
}
```

B is subtype of A.

obj has type A*, but may point to B.

foo expects A*, but B* may also be used.

Substitution Principle: Whenever an object of type T is expected, any object of any subtype T of S can be used.

Encapsulation / “Data Hiding“

BSSE System and Software Engineering

Class A encapsulates data and operations

Access to data controlled by well-defined interface (here: getter-function)

Subclasses may access protected members, but not private members.

A itself may access val.

But foo may not. val is *hidden*.

```
class A {
public:
    A(int aVal):val(aVal) {}
    int getVal() const { return val; }
protected:
    int val;
};

int foo(A* obj) {
    return obj->val; // error
}

int bar(A* obj) {
    return obj->getVal(); // OK
}

class B: public A {
public:
    B(int aVal): A(aVal) {}
    int getValTimesTwo() const { return 2*val; }
}
```

Dynamic Dispatch


```
class A {
public:
    int getSomething();
    virtual int calcSomething();
};

class B: public A {
public:
    int getSomething();
    virtual int calcSomething();
}

int foo(A* obj) {
    return obj->getSomething()+
           obj->calcSomething();
}

int bar() {
    A* obj = new B;
    return foo(obj);
}
```

B overrides *getSomething()* and *calcSomething()*

Always calls A::getSomething()

calls implementation depending on the actual type of obj at runtime.

Object passed to *foo* is actually of type B, although parameter has type A*.

Abstraction


```
class A {
public:
    virtual void doSomething() = 0;
};

class B: public A {
public:
    virtual void doSomething();
}

void foo(A* obj) {
    return obj->doSomething();
}

int bar() {
    A* obj = new B;
    return foo(obj);
}
```

A declares *doSomething()*, but does not provide an implementation.

B provides an implementation.

*obj is of abstract type, but may take value of concrete type.

Object passed to *foo* is actually of type B, although parameter has type A*.

Challenge: Regression Test Suites


```
class A;
class B {
public:
    B():val(0) {}
    int getVal() const { return val; }
    int calcVal(int x) {
        val=x*x+2*x+5;
    }
    bool operator==(const B& other) const {
        return (other.val % 7)==(val % 7);
    }
private:
    int val;
};
B* foo(A* obj);
enum verdict_t testFoo_123() {
    A* input_obj = new A(…);
    B* ret = foo(input_obj);
    B* refValue = new B();
    refValue->calcVal(???);
    if (*refValue==*ret) {
        return success;
    } else {
        return failed;
    }
}
```

We know how to do this…

Hrm…how do we get the state we had in the stimulation run?

Is this the right concept of equivalence for this test?

→Heuristics, Annotations, „Patching“ code (autom.)

Challenge: „Data Hiding“

```
class Stream {
public:
    Stream():readPtr(0),writePtr(0) {}
    void read(char* data, unsigned int size) {
        if (readPtr+size>writePtr) {
            /* error */
        }
        …
        readPtr+=size;
    }
    void write(const char* data, unsigned int size) {
        …
        writePtr+=size;
    }
protected:
    char* buffer;
    unsigned int readPtr;
    unsigned int writePtr;
};
```

Only default constructor present, ininitialises *readPtr*, *writePtr* to fixed value

We want to test *read()*

But *read()* requires *writePtr* to be different from 0

How do we automatically construct instance of *Stream* to test *read()*?

Execute a sequence of method calls after construction (→Heuristics, Protocol State Machines).

Construction Patterns

- Abstract Factory Pattern
- Factory Method Pattern
- Prototype Pattern
- Builder Pattern
- ...

❖ ⇒Extensive variability for the developer, increased complexity for the tool