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

Ralf Gerlich, Rainer Gerlich
Dr. Rainer Gerlich System and Software Engineering
BSSE
Immenstaad, Germany

e-mail: Ralf.Gerlich@bsse.biz,
Rainer.Gerlich@bsse.biz

Carsten Dietrich
Deutsches Zentrum für Luft- und Raumfahrt e.V.
(DLR)
Bonn, Germany

e-mail: Carsten.Dietrich@dlr.de

***Abstract*—With the advent of languages such as C++ and Java in mission- and safety-critical space on-board software, new challenges for testing and specifically automated testing arise. In this paper we discuss some of these challenges, consequences and solutions based on an experiment in automated source-code-based testing for C++.**

Keywords-automatic test data generation, C++, unit testing, robustness testing, testability, Java, object orientation, safety, security

I. INTRODUCTION

Automatic source-code-based testing is a method for software testing in which the procedures and functions of a software package are automatically stimulated based on the interfaces visible in the source code.

Fully automatic generation of the test environment, test stimuli, proper instrumentation of the source code and automatic reporting of observations in condensed form allows massive stimulation of software with millions of stimuli, while keeping the number of reported anomalies sufficiently low, thereby increasing the probability of occurrence and detection for sporadic faults.

Due to automatic instrumentation of the software package under test and the test environment, functional faults can be detected even if no application-specific oracle is available to individually check test stimuli and the corresponding software reaction for compliance with the specification[1].

The effectiveness and efficiency of this approach is strongly dependent on the approach selected for generation of input stimuli.

While on the one hand side, the profile of the stimuli generated should be as close as possible to the operational profile in order to have representative evidence of the software working under operational conditions, robustness testing by way of injecting invalid values or values normally not or rarely occurring during operation should not be left out.

It is therefore important to properly balance these two aspects when generating stimuli. However, it can be very difficult or even impossible for a test tool to infer the programmer's intent or the requirements regarding valid and invalid values from the source code, except if sufficient annotations for this explicit purpose are present.

However, domain-specific heuristics may be applied in order to tune the generation of stimuli towards the most probable intent, neither under- nor overemphasizing robustness testing.

A considerable body of experience on all of these aspects in the space-software domain exists for purely imperative languages such as Ada[2] and C[1][3], but with the advent of languages such as C++ and Java in space software, an extension towards object-orientation is necessary.

Many of the issues discussed in this paper in context of automated testing of C++ software are also valid for conventional, manual testing of C++ source code. Although there are some issues for automated testing, the discussion shows that the complexity of testing C++ software demands automation, e.g. to tackle the large number of combinations significantly increased by O-O concepts such as overloading and dynamic dispatch, and to obtain sufficient visibility on what is really executed.

The paper is structured as follows: First we define the aspects of object orientation which will be subject of the discussion in this paper, and discuss some advantages, disadvantages and issues of object-orientation in general and of C++ in particular in context of on-board space software.

This is followed by an exposition of challenges with regard to automated source-code-based testing that we have encountered so far, followed by a presentation of solutions and/or consequences of these challenges.

Then we present the results of applying the current tool version to representative space on-board software written in C++, followed by our conclusions from the analysis and experience.

II. THE FAST PROCESS

The FAST (Fully/Flow-optimised Source-code-base Testing) process was presented in detail already in[1] in context of C software. Therefore only the most important features as of relevance for testing of object oriented software shall be discussed here.

Massive stimulation: As the process takes the information required for generation of stimulation data and of the test environment from the source code and other machine-readable information – as far as available, a huge number of stimuli can be injected into a FUT (Function-Under-Test).

Fault injection: The stimuli may represent either valid or invalid data.

Observation of properties: When executing a FUT the process can observe the properties, especially anomalies, and record them using given algorithms.

Filtering of Information: The observed information is filtered according to given criteria, thereby reducing the amount of information to be evaluated manually.

Generation of test drivers for regression testing: According to given criteria the process can select certain tuples of input and output data for regression testing and generate the test drivers and their execution environment.

Massive stimulation in context of fault injection significantly raises the probability of fault activation. Information filtering reduces the amount of information, and provides lists on unique anomaly patterns. Therefore sporadic faults have been detected although the software-under-test passed already the verification procedures as required by the standards for a given criticality level.

A recent publication[4] on automation in context of Java suggests that automation does not bring an advantage when the number of injected stimuli is limited to the order of magnitude which can be reached by manual generation of stimuli. This is not surprising as the generated stimuli are equivalent to the manually generated ones – more or less.

An issue arises in context of "design-by-contract", when the information on the contracts is not available in machine-readable form. Then a FUT may be exposed to invalid data, i.e. to data which may not occur under normal conditions of system operation, while the test automaton considers such data as valid as they comply with the prototype of the FUT. Such events are called "false positives" as they suggest a fault while it is not – provided that the related contract is always fulfilled at time of operation.

In on-board software the concept "design-of-contract" is applied in many cases, and it is a major base to achieve safe operations, thereby relying on a cooperative user of a FUT.

However, in context of security such a user is not cooperative and the so-called false positives are just the holes through which a hacker can penetrate into the system, i.e. it is desirable to know about.

C++ may be applied to applications where security is of higher interest. Due to use of UML and design patterns allowing a higher degree of reuse, C++ is increasingly used in the space domain.

Both trends make it reasonable to extend the FAST process from C to C++.

III. Object Orientation Context

In this paper we will mainly discuss object-orientation based on the following features, which are present in Java and C++:

- Subtype polymorphism/inheritance,
- encapsulation ("data hiding"),
- abstraction, and
- dynamic dispatch.

Subtype polymorphism is the notion that a datatype S – the subtype – is related to some datatype T – the supertype – by a notion of substitutability, i.e. wherever an element of type T is applicable, an element of type S can be used as well. This applies specifically to function and procedure parameters.

An example is shown in *Fig. III-1*. Here a class *A* and a subclass *B* of Class *A* are declared. The function *foo* accepts pointers to instances of *A* as parameter. Due to the substitution principle, instances of class *B* may be used in any place where an instance of *A* is expected, so that *foo* also has to accept instances of *B*.

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

Fig. III-1Subtype Polymorphism

Encapsulation typically means both the bundling of methods and object data, as well as the concept of hiding the object state from program elements outside the object class or subclasses. The latter concept is also known as "data hiding".

An example for encapsulation and data hiding is shown in *Fig. III-2*. Here a class *A* is declared, containing data – in the form of data member *val* – and operations – in the form of a constructor accepting a single integer parameter and a so-called getter method *getVal* returning the value of *val*.

The member val is declared to be protected, meaning that it is accessible only from inside the class or – under certain circumstances – its subclasses.

```
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;
  }
}
```

Fig. III-2 Encapsulation and Data Hiding

Therefore, val cannot be accessed directly from the function foo. However, the getter-function getVal is declared public, so it is accessible from outside the class, so that the function bar can use it.

Access from subclasses is shown in the presented subclass *B* of *A*, where *val* is accessed directly inside the member method *getValTimesTwo*.

Direct access by subclasses can be avoided by declaring members to be *private*. In that case, the member can only be accessed by the declaring class itself.

Abstraction is the representation of an idea or concept without specification of its concrete implementation. In most object oriented languages this is supported by specification of methods by interface only, without providing an implementation. Classes containing such interface-declarations – called "pure virtual" methods in C++ – are called "abstract classes". The actual implementations are instead provided by concrete subclasses.

An example for abstraction is shown in *Fig. III-3*. Here, Class A declares a "pure virtual" method *doSomething* – indicated by the keyword *virtual* and the "= 0" after the method prototype. Function *foo* is working on instances of Class *A* and as *A* declares the method, *foo* can make use of that declaration.

However, it is not possible to create instances of *A* directly. For this, a concrete class implementing *doSomething* is required. In the example, this class is represented by Class *B*, a subclass of Class *A*. The lack of "= 0" after the prototype in Class *B* indicates that *B* actually implements *doSomething*. Therefore the function *bar* can create an instance of B and pass it to *foo*.

```
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);
}
```

Fig. III-3 Abstraction

The function *foo* itself, however, is completely unaware of what concretisation of *A* it will be passed. The developer has to ensure that the implementation in Class *B* actually implements the guarantees regarding its behaviour that *foo* expects to hold.

Dynamic dispatch is the process by which the implementation of a polymorphic operation is selected at runtime. This is necessary as due to the substitution principle, a variable of object type A may at runtime hold an object of a subclass B of A, where the implementation of the respective operation in B may differ from that in A.

An example is shown in *Fig. III-4*. Here Class *A* declares two member methods, *getSomething* and *calcSomething*. The latter is declared *virtual*, indicating that it is subject to dynamic dispatch, while the former is not.

Class *B* is a subclass of *A*, providing its own implementations of these two functions. Again, *calcSomething* is subject to dynamic dispatch, not primarily because it is declared *virtual*, but because it overrides a method in A that is declared *virtual*. The method *calcSomething* in Class *A* is said to "inherit" the virtual attribute from the method with the same signature in Class *A*.

Whenever a method is invoked on an instance of a class, the method of invocation depends on whether the method is declared to be subject to dynamic dispatch or not.

```
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);
}
```

Fig. III-4 Dynamic Dispatch

In the latter case, the implementation to be invoked is determined from the type of the expression it is invoked on. This type is determined at compile time. In the example, the call to *getSomething* in function *foo* always resolves to the implementation of *getSomething* in Class *A*, and never to the implementation in Class *B*, even though *bar* actually passes an instance of *B* to *foo*.

If the method is declared to be subject to dynamic dispatch, the implementation to be used is determined at runtime based on the type of the actual instance referred to. As a consequence, the call to *calcSomething* will be resolved to the implementation of that method in Class *B* when *foo* is called with an instance of B, such as, e.g., from function *bar*.

IV. C++ IN SPACE ON-BOARD SOFTWARE

Object-oriented languages in general and C++ in particular have some advantages and some disadvantages regarding the implementation of space on-board software compared to imperative languages in general and C in particular.

For example, encapsulation and inheritance allow for stronger modularisation, increase of software reuse and provide a very basic mechanism for fault isolation.

Classes can enforce data consistency by shielding data from direct access and allowing access only through method interfaces ensuring that invariants hold when control is passed back to the caller. Verification of this enforcement mainly has to consider the contents of a class. However, as behaviour may be modified by sub-classing, verification still has to be repeated for each new subclass.

Also, the object-oriented model coincides with that of many wide-spread modelling concepts such as UML, which may ease the transition from model to implementation.

Object-oriented paradigms are well-suited for application software. This is not only true for graphical user interfaces and similar application concepts, which are usually not part of space on-board software, but also, for example, for data processing applications. The existence of a large number of class and template libraries for this purpose is evidence for this[5][6][7].

However, for the hardware interface level of a space on-board system usually a component-based architecture is more suited due to the fact that the hardware of a spacecraft does not change during the mission. This is different from, e.g., a desktop computer, where peripheral hardware may be added and removed multiple times during the lifetime of the system or even during a single workday.

Thus, mapping the component-based architecture for the hardware interface level to an object-oriented implementation language will either degrade to an imperative approach – e.g. using only an imperative subset of C++ – or at least feel forced.

Consider, for example, the implementation of a driver for a serial interface. If it is implemented as a class, with each object representing one serial interface, the number of instances is clearly limited and fixed for a given hardware setup.

However, one basic assumption of a class-based object-oriented system is that the number of instances and their lifetime are indefinite and unknown at the time the class definition is being compiled.

For this reason alone the compiler will introduce an element of indirection, addressing object data and possibly object methods indirectly via an implicit object pointer ("this" in C++ and Java).

This indirection – and the use of function pointers – could in some cases conflict with industry standards for critical software, such as DO178 which disallows dynamic objects.

It should be noted that the Singleton Pattern[8] does not get rid of the indirection, but instead only places the instance and its instantiation under the control of the class itself.

In C++ the declaration of all methods and data in the class as static would allow the developer to get rid of the indirection. However, now the management of instances would be left to the developer, just as it was in Ada and C. The resulting class would very much resemble an Ada package. The only difference from a simple C implementation is the use of the class as a distinct namespace and the possibility of data-hiding.

C++ as a language has several specific advantages over its historical predecessor C, in addition to those introduced by its object-oriented nature.

C++ has a stricter type system in place and allows for enforcement of stricter type checking in various situations, which is also needed for proper overload resolution.

For example, while enumeration types were implicitly mapped to integer types in C, they are their own type category in C++, albeit with an implicit type conversion to integer.

Also, by declaring a parameter with type "reference to array of T" it is possible to avoid the usual implicit degradation of an array to "pointer to T". Otherwise this degradation leads to loss of information about the original array, such as its size.

Templates allow quite type-strict generic implementations of containers and algorithms, for example. Partial specialisation is a powerful tool for optimisation and even for small amounts of automatic code generation.

However, the principles for finding the matching specialisation for a given instance of a template define a Turing-complete language. In other words: Partial specialisation and matching of templates in C++ provides a whole programming language which is executed at runtime.

While all these features allow for many interesting applications[9], as a consequence the code may be very difficult to review, to verify and test.

For example, the symbols "<" and ">" are overloaded in the context of declarations and uses of templates, which impacts readability due to syntactic ambiguities.

Consider the term "T<a<b,c>::d>", which is clearly an instance of template "T". However it is not as clear what the arguments are.

It is possible to read the only argument being an enumeration constant "d" declared in the template instance "a<b,c>".

An alternative interpretation would be two arguments, one being the boolean result of the comparison "a<b" of two constants "a" and "b", and the second being the boolean result of the comparison "c>::d" of two constants "c" and "d", where "d" is explicitly stated to be declared in the root namespace.

Without knowing what "a", "b", "c" and "d" are it is not possible for the reader to determine the actual syntactic structure of the term.

To make matters worse, the current C++ standard[10] adds ">>" into the mix of overloaded symbols. This is relevant when the last parameter to a template instance is itself a template instance specification, such as in

"S<T<U> >"

here written in the way required by older C++ standards, with a space in between the two closing angle brackets. The new standard also allows to write this as "S<T<U>>", without a space between the angle brackets.

However, an advantage is the implicit combination of declaration and initialisation of variables: Whenever a class-type variable is declared, it is initialised either by an explicit initialiser or constructor call, or by an implicit call to the default constructor.

This default constructor may under specific circumstances be a so-called defaulted default constructor which initialises all member elements to their default values according to their type.

Finally, function and operator overloading may help in making the code more readable by adopting a mathematics-like notation, e.g., for quaternion, vector or matrix operations.

However, many of the implicit elements of C++ semantics may also lead to less comprehensible code.

For example, assignments to variables of class type may implicitly call a copy-constructor or operator, as may return statements with objects of class type.

Constructors accepting a single argument of a given type may be used as implicit conversion operators, except if they are marked as “explicit”.

Further, the comprehension of overloaded operators depends on the reader performing the overload resolution properly, while in C the name of the function to be called is unique. Although in case of functions with static linkage, there could be more than one function of the same name in an application.

Consequently, regarding critical software and related standards, a subset of C++ and proper design patterns should be defined.

V. Test Challenges in Object-Oriented Languages

Some typical features of object-oriented languages present specific challenges to testing in general and automatic generation of input data for test and stimulation in particular challenging.

A. Encapsulation/Data Hiding

In case of C++ encapsulation in the sense of data hiding is achieved by declaring object members as protected or private, the difference between the two being whether the members are visible to subclasses or not.

While in Ada and C, all possible values/states of a record (Ada) or structure/union (C) could be generated by recursively filling the fields of the record, structure or union with values of the appropriate type, encapsulation implies that this is not generally possible for objects of class type.

```
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;
};
```

Fig. V-1 Challenge Data Hiding

Clearly one could simply modify the source code automatically in such a way to remove the encapsulation for sake of stimulation.

However, besides simply hiding the object state and its representation from object users, encapsulation is also typically used to ensure a consistent representation of the state of an object.

Consider, for example, an object, the state of which is represented internally by two fields of integer type. The specification defines that, while any of both fields may be negative, their sum must be positive. Consequently, constructors and methods must not produce inconsistent object states.

Now consider the case where both fields are randomly initialised without considering the constraint: In about half of the cases, the values selected would not satisfy the constraint given in the specification.

While the values not satisfying the constraint are of interest in terms of robustness testing – which includes testing for robustness against invalid input data –, it is typically not desirable to spend about half of the stimuli on robustness testing alone.

Such constraints are not known to the test tool, except if provided explicitly in a form usable for automatic generation of applicable data, e.g. in any machine-comprehensible form, which – so far – is rarely the case. They also cannot be extracted from the source code in general due to Rice's Theorem, an extension of the Halting Problem, which states that for any non-trivial property there is no algorithm that can determine whether a given program has that property. In this context “non-trivial” means that there is at least one program that has the property and at least one that does not.

It is therefore reasonable to use the declared constructors for generating objects for stimulation with what according to the implementation should be consistent states. Only specifically for robustness testing the concepts of consistency as implemented in the software should be ignored.

However, it is not guaranteed that all possible states of the object can be reached this way. One simple example of this are objects implementing a finite state machine which always starts in the same initial state after construction. Except if the state machine is degenerate, there must be more than this one initial state, and by construction these other states cannot be reached from calling the constructor alone.

The only other way of manipulating the state is by using the methods of the object.

In case of pure finite state machines theory dictates that the number of transitions required to reach any given, reachable state is finite. However, although the state space may be finite, it may still be too large.

Thus the construction of an object requires invocation of the constructor followed by a sequence of method invocations, the length of which is unknown without additional information.

To further complicate the issue, the construction of objects may be recursive in that the constructor itself or the methods invoked afterwards may require data of class type as parameter.

For an example, consider *Fig. V-1*. The class declared in this example represents a simplified stream or rather, a first-in-first-out-buffer (FIFO). Initially the FIFO is empty, i.e. no data has been written to the stream and – consequently – no data has yet been read from the FIFO.

Wanting to test the method read, one would have to fulfil the condition that there are at least as many bytes available in the stream as shall be read. Clearly, this is not the case directly after construction for any case in which the parameter size passed to read is greater than zero.

There is no constructor that would allow the stream to be initialised to any state other than the empty state. The only way to create a pre-filled object is to construct one with the default constructor and to call write with the appropriate parameters. However, for all practical purposes, without annotations a software tool can only guess this connection.

B. Subtype Polymorphism

Subtype polymorphism is not specific to object-oriented languages. For example, it is possible to use a value of type "unsigned short" for a parameter of type "unsigned int" in C. In Ada it is possible to declare subtypes of scalar types, e.g. with reduced value range.

However, neither Ada nor C allow declaration of subtypes of record (Ada) or struct/union (C) kind.

While in the subtyping scheme of Ada, the total range of values is given by the topmost scalar supertype and subtypes can only select a subrange of this total range, every subclass S of a superclass T extends the set of objects applicable as values of type S.

This is true even if S does not introduce any new object fields, as S may introduce variants of implementations for the methods of the class. Thus the objects of type S may behave differently from the objects of type T.

Consequently, when generating an object of type T, also all subclasses of T have to be considered for testing.

C. Dynamic Dispatch

Due to dynamic dispatch, subclasses may modify the behaviour of superclass methods, even of those that are not overridden.

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

Fig. V-2 Library Class A

As an example, consider Class *A* declared in *Fig. V-2*. This class declares a method *foo* subject to dynamic dispatch and a method *bar* that uses *foo*. Whether *bar* is subject to dynamic dispatch or not is not relevant for this example.

Assume that Class *A* is declared in a class library used by an application and the application declares Class *B* shown in *Fig. V-3*. Class *B* overrides *foo*. As *foo* is subject to dynamic dispatch, an invocation to *bar* on an instance of Class B may show different behaviour than for instances of Class A itself.

No amount of verification on the class library can ensure that the guarantees associated with Class *A* still hold in the context of the application, except if derivation of subclasses by the application is prohibited and properly checked.

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

Fig. V-3 Application Class B

D. Templates

A challenge more specific to C++ as a language is template programming. The combination of partial and complete specialisation with the pattern-matching provided by the C++ template mechanism is in itself a Turing-complete language with programs being executed at compile-time by the compiler.

This specific feature of C++ has led to a large set of generic template libraries, including the Standard Template Library defined by the C++ standard itself, containing definitions and implementations of various generic container classes and algorithms. Other examples are the Boost[5] and LOKI[7] libraries or the Computational Geometry Algorithms Library (CGAL)[6].

The specifically practical challenge regarding information extraction from source code lies in the effort required for implementation of an appropriate parser which is able to properly understand template declarations and to apply pattern matching as specified in the standard on template instantiations.

However, a more conceptual issue arises for templates, similar to that resulting from dynamic dispatch and subtype polymorphism: The issue of determining the candidate types for type parameters.

While for parameters of class type in functions and methods it is straightforward to determine the candidate subtypes from the subclass relationship explicitly specified by a language construct specifically designed for this purpose, it is difficult or even impossible to determine which types would be eligible for a type parameter of a template.

The template may place constraints on the interface and behaviour of the type parameter. For example, CGAL provides a template to define what is called a kernel, providing operations for defining and manipulating polyhedra with a given type of vertex, edge and face. Instances of the polyhedron-template require a structure as type parameter which mainly contains type definitions for the vertex, edge and face types. The names of these types are fixed. Further, there is a set of operations that are required to be possible with objects of these types.

If one tries to instantiate this template using a type parameter that does not satisfy these constraints, compilation errors may occur, but do not necessarily occur. Depending on how the compiler instantiates the templates, errors may only occur when a method of the template instance is used which – when instantiated with the given template parameters – is semantically incorrect.

Besides the complexity of implementing an appropriate parser and semantic analyser, it would seem straightforward to simply try all types present in the application or library as type parameters wherever type parameters are required.

However, template instances themselves are types and would therefore be eligible themselves as type candidates under such a scheme. The set of candidates to be evaluated could therefore be infinite in the presence of templates.

E. Stubbing of Constructors

Automated testing may require stubbing of functions and – in the object-oriented case – classes. The reasons may vary.

In early stages of implementation there may be parts of the software that are not yet implemented. There also have been cases where only the interface declarations but not the implementation – neither in source- nor in object-code – was available. A useful strategy for approaching the automated test of a large software package is also to exclude part of the implementation and limit the actual test to a subset of the package.

With object-oriented languages such as C++ it may be necessary to also stub constructors. These are special functions that are intended to set up an object for its initial state, possibly depending on input parameters.

Let us assume for a moment that we are talking about stubbing a default constructor, i.e. a constructor that does not require any parameters.

In that case the constructor to be generated needs to ensure that inherited portions of the object are initialised by calling a constructor of each superclass, and that fields introduced by the class itself are properly set up.

In C++ this is done using initialisation lists, which are placed immediately before the actual body of the constructor. The rationale for this separation from the actual body is that the object should be in a defined state already upon entering the body.

For example, to call the constructor of a superclass, the name of that superclass is given, followed by the parameters to the constructor in parentheses. Similarly, to initialise a field, the name of the field is given, followed either by the value for the field or by parameters to the constructor if the field is of class type.

As any class may have more than one declared constructor, each of which may lead the object to be initialised in a different state, representative testing would require to ensure that the constructor being called is actually the constructor that is to be called in the final implementation. However there may be no information available on which constructor this is.

In other situations where definitive information is not available, automated source-code-based testing falls back to randomness or iteration over the whole set of possible alternatives.

In this case, however, it is only possible to select one constructor statically at the time of generation of the stub.

Now let us consider the more complex case, where the constructor to be stubbed receives arguments.

Usually, these arguments are used to initialise the fields of the object. This initialisation is not necessarily direct in that the value of a parameter is directly written to an object field.

Instead, intermediate calculations may be carried out and the parameters or the results of these calculations may even be used as parameters for further constructor calls, e.g. for super-class or field constructors.

The stub generator cannot know the correct way of transforming the parameters into inputs for the constructors called.

F. Use of Design Patterns

Some design patterns require special handling in test data generation and in testing. One prominent example is the Singleton Pattern[8], shown in *Fig. V-4*. The purpose of the singleton pattern is to ensure that only a single instance of a given class exists in a given context. For example, there may be a single instance per thread or a single instance for the whole application. The singleton instance provides control over the single-instance-criterion and simple access to the appropriate instance.

Use of the Singleton Pattern may be criticised from an architectural and design point of view – e.g. for increasing coupling, an attribute unwanted in object-oriented design, or for making construction order less predictable – the challenge for testing does not arise from the design issues, but rather from the form how the Singleton Pattern is often implemented.

In order to enforce the single-instance-rule, construction of other instances has to be prohibited. This is usually achieved by hiding all constructors of the respective class and introducing a static class-method – e.g. named *getInstance()* – to return the single instance. The hidden constructors are visible to that class method, so it is able to construct an instance, but no part of code outside the class can do so.

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

Fig. V-4 Singleton Pattern

Notably, in most languages there is no explicit language support to distinguish such construction methods from other, non-construction methods. They are not constructors in the meaning that term has in the language, as a visible constructor could be used to violate the single-instance-rule. They have to be declared and defined in the same form as any class-method is expressed, although they are introduced

with a very specific intent, and this intent is important for test data generation.

A test data generator may at some point find that the FUT or some other function which has to be called for test preparation requires an instance of the singleton class to work on. All of the relevant constructors of that class are hidden, so the data generator has no possibility to construct an instance in the classical way.

A similar situation may arise with the Abstract Factory Pattern, the Factory Method Pattern, the Prototype Pattern or the Builder Pattern. All of these share the feature of introducing an additional level of indirection in creation of objects, with the actual mechanism of construction not being apparent from the syntax for a program analysing the code.

G. Generation of Regression Test Suites

The process flow of execution of a single regression test cases consists of the following steps:

1. Construction of the input data
2. Invocation of the function-under-test
3. Comparison of the actual output to the expected output
4. Recording of the input and output data for repetition of the test

As in the FAST-process applied by the authors the test generator shall require no additional information about the application, the generator also has no knowledge about the expected output. Instead, test stimuli prompting an interesting response by the function-under-test are collected and regression test drivers are generated based on the input stimulus and the actual output observed. To allow testing against a specification, the observed outputs have to be manually verified regarding the contents of the specification. Otherwise, the regression test can only be used to observe changes in the behaviour of the software between versions.

When generating regression test suites from automatically generated test data, the first two steps in the regression test process are straight-forward to handle. After all, the input data has been constructed before and the way it was constructed is known and can be transformed into code that repeats this process. The generation of an invocation expression or statement also poses no special implementation challenges.

However, the third step is much more complicated. The approach so far for C has been to iterate over the structure of the observed output data and generate matching statements comparing the output observed in the test data generation run to the output observed during the execution of the test driver.

But this is not a mechanism for logical comparison in any case. For example, in case of a container type such as a tree or a hash set, equality is defined in terms of objects contained in the container rather than of the actual values of record elements used to represent the set.

A very much simplified example is given in *Fig. V-5*. Here, a function *foo* expects an instance of class A and returns an instance of class B.

The objective of the regression test is to

1. generate the instance of A in just the same way as it was generated during the stimulation run,
2. pass it to *foo*,
3. record the instance of B returned, and
4. compare the state of that instance to the state of the instance returned during the stimulation run.

```
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 = foot(input_obj);
  B* refValue = new B();
  refValue->calcVal(???);
  if (*refValue==*ret)
     return success;
  else
     return failure;
}
```

Fig. V-5 Regression Testing

As previously explained, Steps 1 to 3 are straightforward to implement. Step 4 is the difficult part of the process.

The instance of B was generated by the function-under-test in the stimulation run, so the way its state was achieved is not known to the test-tool.

The state variables are hidden from the outside, and the behaviour of the function *setVal* is clearly not useful to re-establish the state, as it does not set *val* directly, but rather indirectly via some calculations. In the practical case, these calculations may be arbitrarily difficult to reverse. Thus it is not possible to use *setVal* to re-establish the previously observed state.

Further, it is unclear what notion of equivalence or equality shall be used to compare the newly observed to the reference value. In this case, the comparison operator was redefined to represent a notion of equivalence, not equality. It may be that in this specific case, this notion of equivalence is not applicable, but rather direct equality or another concept of equivalence was intended to be used.

So neither is there an obvious way of providing a copy of an output object to use for logical comparison using appropriate user-provided comparison methods, nor is it generally possible to at least implement a structural comparison due to the presence of hidden data.

H. Inaccessible Types and Methods

Classes may contain declarations for methods and types – including other classes – that are marked to be hidden. In

Java and C++, these may be declared as private or protected, and thus are inaccessible to any code or declarations outside designated portions of the software.

These private methods are often internal helper functions used to add better structure and separation of concerns to the implementation of the functionality specified for the public interface of the class.

As such, they may violate invariants defined on the state of the class, as long as these violations are resolved before execution leaves the domain of the class again.

Depending on the test strategy, tests of these internal methods may be desired or not.

In a black-box strategy, the main focus is on testing whether a class performs as specified when using its public interface. The internal methods may be seen as implementation detail that is invisible to the user anyway.

In a white-box strategy, individual tests of these internal methods may be desirable.

For example, any faults being observed in a test of the public interface may come from defects either in the public method or in any of the internal methods called from there directly or indirectly. Localisation of the defect may thus be difficult. Testing the internal methods bottom-up instead may reduce the localisation effort for any defects to be found.

VI. Consequences and Solutions

A. Encapsulation/Data Hiding

There seems to be no solution which does not risk to either systematically exclude a significant portion of the state space of an object or to produce large numbers of duplicate inputs or long sequences of idempotent or otherwise superfluous method invocations, without at the same time requiring additional information that is to be supplied manually in a formal manner usable by a software tool.

For example, one could specifically mark methods that have no effect on the object. These would not be considered when generating a sequence of method calls intended to manipulate the state of the object.

In C++, these can be marked by using the qualifier *const* after the method prototype. However, it is not unusual to find a similar method with the same name and without *const*-qualifier in the same class.

This is often the case in getter-methods that provide access to a sub-object by passing a reference to that object to the caller. In the const-qualified case, the referenced object is const-qualified itself and thus cannot be modified by the caller. In the non-const-qualified case, the const-qualifier is also missing on the referenced object, and thus the caller can modify that object.

Still, the getter-method itself does not affect the state of the object, but instead only returns a reference to part of it for the caller to potentially modify.

Another way of restricting the set of sequences of method calls would be specification of a protocol state machine. Such a machine would formally specify the order in which the methods may be called in the nominal case, and may even specify constraints on the values the methods may be passed. This form is often used in model-based testing.

B. Subtype Polymorphism/Dynamic Dispatch

As a consequence of dynamic dispatch in combination with subtype polymorphism, it is not possible to use evidence collected in testing a class library to deduce the reliability of that class library when used in an application.

An application may use a class library, for example, by deriving its own subclasses from classes in the library. In some cases a class provided by the library may be abstract, but no concrete subclass may be provided in addition. In that case, the application is even forced to provide its own subclass to be able to make use of the functionality.

There the set of all subclasses of a class cannot be known – neither in the automatic nor in the manual case, as would be required for representatively generating objects of class type.

This is not necessarily different from C, as dynamic dispatch can be implemented in C as well using function pointers.

However, in object-oriented languages dynamic dispatch and subtype polymorphism typically are important parts of the language. They may even play a crucial part in the decision for an object-oriented over an imperative language.

Thus any attempt to prohibit the use of these features to avoid the difficulties in testing would very probably be subject to strong criticism.

As a minor consequence of subtype polymorphism and dispatch, a unit actually is a complete class, including all methods that are inherited from superclasses. The change of behaviour may affect inherited methods, so that these have to be re-tested in context of any subclass. However, that is not a consequence specific to automatic test data generation.

C. Templates

To solve the issue of candidates for template type parameters, considering only those concrete instances used in the application seems to be the most straightforward solution.

This also fits the approach of only considering complete applications. The template instances used in the application can be considered to be unique classes, as if they were defined without the use of templates.

D. Use of Design Patterns

Short of requiring manual annotations, the main possibility of handling construction patterns seems to be the use of heuristics.

The patterns most relevant to automatic test data generation seem to be construction patterns. These have in common that there is some method or function that provides an instance of the relevant class.

There are some issues with this heuristic. First of all, it includes getter methods. After all, the existence of the Abstract Factory Pattern requires that a tool also considers object methods as possible constructors. However, getter methods may return an object of the desired type, but that

object might have been supplied earlier to the constructor of the object on which the getter method would be invoked.

One option would therefore to exclude all methods that directly or indirectly require an instance of the respective type for invocation. This way, however, one might exclude implementations of the Factory Pattern that use another instance of the respective type as a template for the object to be created. This issue is similar to the problem of achieving different states of an object by calling a constructor and a sequence of methods, as discussed in Sect. VI.A, and therefore the same solution approaches may apply.

As another complication, the return value of a function might not be the only way to pass outputs to the caller. Parameters passed by-reference or even more complex encapsulations of outputs may be another possibility.

For example, a function in C++ may declare a parameter to be a reference to a pointer to class T, which may be used to pass a newly created instance of T to the caller. However, without additional information a tool cannot distinguish whether the parameter is bidirectional (input-output) or output only. Thus, the decision on whether the function requires an instance of the relevant type as input cannot be made.

Further, at least theoretically it is possible that the instance is passed to the caller encapsulated in another object, e.g. when multiple objects are created at once. This may be necessary, e.g., due to structural invariants such as a group of objects each having a link to each other. Obviously, any sequence of generating these objects would breach the invariant at least temporarily. The reason for use of the Abstract Factory Pattern may be that this temporary violation of the invariant shall be kept hidden from the caller.

Thus, any of these approaches to automatic identification of constructor methods that are not constructors in the language sense will necessarily be a heuristic and most probably exclude some of such constructor methods from consideration.

E. Inaccessible Methods and Types

As already mentioned in Sect. V.H, this problem might be solved by simply stimulating only on public interfaces and assuming that the internal methods are tested in that context as well. The degree of coverage achieved is measured in context of the FAST process and thus gives a feedback on what was tested.

However, if the test strategy required testing of internal methods as well, there are at least three options:

- declaring the relevant functions from the test environment to be friends of the relevant class (possible in C++),
- declaring the test functions as part of the relevant class,
- modifying the class declarations so that the relevant elements are public.

All of these options require modifications to the source code for the purpose of the test. These modifications may be temporary in that they are only present during test, but the software product as delivered stays unchanged. Fortunately, it is possible to automate these modifications.

The first alternative is possible at least in C++. Declaring a function to be a friend of a class allows that function access to private and protected members of that class, thereby removing the limitation for the test environment.

In languages which do not provide a feature comparable to friend-declarations, the other alternatives may apply. A function that itself is a member of a class also has access to all the other members of the same class, independent of whether they are private, protected or public.

The third alternative achieves the same, but makes the relevant members accessible even to functions outside the class scope.

At least in C++, this rather drastic modification is not expected to change the behaviour of the software. Such a change could only occur if due to the increased accessibility, overload resolution – i.e. the selection of the operator, method or function to be called from a set of candidates of the same name – would now result in different callees being selected in parts of the code.

However, at least in C++, visibility and accessibility are separate concepts: All members of a class are *visible* at all points of the code, and as such are all considered in overload resolution. Once overload resolution has resulted in a single remaining candidate member, accessibility is checked. Therefore, the set of candidates is not changed by changing the visibility of the members.

The same may apply for other languages.

F. Conclusions

The solutions discussed above – requiring manual intervention – are not a matter of test automation and of the FAST process, but of manual, traditional testing as well. However, in context of such a high degree of automation manual intervention is not the preferred solution, but in some cases a constraint which cannot be avoided – at least currently. The challenge for the FAST process is to find and propose more efficient solutions.

VII. IMPLEMENTATION AND TESTS

The issues as discussed above in Ch. V require a staggered approach to extend the FAST process from C to C++ under consideration of what typical space software needs and how it could best benefit from the features of the FAST process.

A. Steps of Implementation

Therefore the extension of the FAST process from C to C++ shall be performed in the course of the following steps.

In a first step – very close to finalisation – the basic features shall be supported as required for identification of sporadic faults complementing the standard test process:

- Massive stimulation
- Fault injection
- Stubbing of missing functions / methods including constructors and destructors

- Provision of filtered lists on anomalies pointing to potential faults

with the following limitations:

- Support of visible, public methods only,
- No parsing of templates.

In the next steps shall be supported

- information hiding and encapsulation
- templates.
- procedures to step through the state space, and finally
- regression testing.

The effective schedule for implementation of next features will be driven by the needs of the application software.

B. *Implementation*

In its current shape the extension of the FAST process to support C++ software does – in addition to the features already supported for C

- extract information from C++ source code according to the C++ Standard 201112 without supporting templates
- instrument the source code for coverage analysis, fault injection, supporting the later extension to cover encapsulation,
- generate stubs for missing methods including instantation of a concrete subclass if a class is abstract, recursively considering superclasses and subclasses
- generate test stimluli for C / C++ types including calls of associated constructors,
- call methods including random selection of overloaded methods at run-time.

Based on the extracted information more context information is derived, addressing correlation of information across compilation units, because generation of the test environment requires more information than compiler and linker need. E.g. for an abstract (super-)class concrete methods may be provided in separate compilation units, i.e. in the compilation unit of the super-class nothing it known about its sub-classes. To test the methods of the super-class all concrete methods of its sub-classes must be executed.

C. *Tests*

Tests have been applied to a software package intended for use for an experiment on the International Space Station ISS. The application shall run under Linux. Its properties are:

- 302 hpp-files
- 229 cpp-files
- 643 public methods
- 53200 physical lines
- 22600 LOC

Out of these 643 methods 135 – provided in 20 files respectively abstract classes – require more derived context information on concrete subclasses to test them. Tab. VII-1 shows quantities on classes as occurring in the parsed application files. The second colum refers to classes, class templates, structures and unions defined in the set of application files, the third colums to all files considered. Latter set of files refers to the application files plus the h-files from Boost and Loki and some h-files from Linux – as required by the application.

Item	Application Files Only	All Files
Classes	279	325
Structures	55	176
Unions	0	15
Class Templates	364	469
Total	**698**	**985**

Tab. VII-1: Profile of Classes

The GNU compiler g++ 4.7.2 has been applied for compilation and testing of the software.

The test generation process has been applied to this software package and the 643 methods.

Evaluation of the test results has just been started, mainly focusing on verifying correctness of the test process and report generation, taking the results for improvement of the test process, and minimisation of false alarms. When this phase is finished, analysis of the report for issues in the software under test will start.

VIII. CONCLUSIONS

The issues for implementation of an automated test process which takes its information from source code or other machine-readable information – if provided – are much more challenging for object-oriented languages like C++ or Java than for imperative languages like C or Ada. Such an automated process requires much more information than what is needed for compilation and linking the source code to get an executable test environment. Information from different compilation units may have to be combined to get a full overview on the correlations.

The power of C++ or of an object oriented language respectively implies that the mapping between a statement in source code onto assembler code is much more complex and extensive than in C. E.g. in C a *return* may be covered by a short sequence of assembler instructions, while in C++ a cascade of operations may be executed, including implicit calls to destructors and copy/move operators..

Further, the dynamics related to sub-classing, overloading and dispatching generates a huge number of combinations which have to be considered in advance when building the test environment.

However, what is a big challenge for implementation of such an automated process, also applies to manual test preparation. The huge number of cases to be considered suggest that a test engineer only can achieve a very small coverage figure w.r.t. what should be covered, especially regarding overloading and dispatching.

In consequence, the high dynamics of object oriented software and the huge number of possible execution paths which come on top of the huge number already of imperative programming – suggest that an automated process is required such as FAST to achieve a sufficient degree of coverage. Although such an automated process never will perfectly reach the required coverage, it will reach more than what can be achieved when manually building the test environment.

An automaton can extract information from the source code much more efficiently than an engineer and correlate and synthesize it to what is required for generation of the test environment, which results in derivation of more combinations for testing.

The current status of implementation shows that the FAST process can be applied to object-oriented languages like C++, too, and it forms the base for further extension towards more C++ features. In its current status the FAST process can support fault identification in C++, especially tackling sporadic faults, as it did in the past for C software, for quantities of source code as produced in real projects.

ACKNOWLEDGMENT

The activities as referenced in this paper were supported by DLR Space Agency (Deutsches Zentrum fuer Luft- und Raumfahrt) on behalf of BMWi (German Federal Ministry of Economics and Technology) under reference number FKZ 50 RA 1120.

REFERENCES

[1] Ralf Gerlich, Rainer Gerlich, Marek Prochazka, Kenneth Kvinnesland, Bengt Solheimdal Johansen: A Case Study on Automated Source-Code-Based Testing Methods, Proceedings of the DAta Systems In Aerospace Conference 2013 (DASIA 2013).

[2] R. Gerlich, R.Gerlich, T. Boll, K. Ludwig, Ph. Chevalley, N. Langmead: Software Diversity by Automation, Proceedings of the DAta Systems In Aerospace Conference 2005 (DASIA 2005).

[3] R. Gerlich, R. Gerlich, C. Dietrich: Fault Identification Strategies, Proceedings of the DAta Systems In Aerospace Conference 2009 (DASIA 2009).

[4] G.Fraser, M.Staats, P.McMinn, A.Arcuri, F.Padberg: Does Automated White-Box Test Generation Really Help Software Testers? Proceedings of the 2013 International Symposium on Software Testing and Analysis, pp. 291-301

[5] Boost C++ Libraries, http://www.boost.org/

[6] Computational Geometry Algorithms Library, http://www.cgal.org/

[7] Loki, http://loki-lib.sourceforge.net/

[8] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns, Addison-Wesley, 1995.

[9] Andrei Alexandrescu: Modern C++ Design: Generic Programming and Design Patterns Applied, Addison-Wesley, 2001.

[10] ISO/IEC 14882:2011 – Programming languages – C++, 2011