ACRE: An Automated Aspect Creator for Testing C++ Applications

Etienne Duclos*, Sébastien Le Digabel†, Yann-Gaël Guéhéneuc*, and Bram Adams*
*Department of Computer and Software Engineering
†GERAD and Department of Mathematical and Industrial Engineering
École Polytechnique de Montréal, Québec, Canada
(etienne.duclos, yann-gael.gueheneuc, bram.adams)@polymtl.ca, sebastien.le.digabel@gerad.ca

***Abstract*—We present ACRE, an Automated aspeCt cREator, to use aspect-oriented programming (AOP) to perform memory, invariant and interferences testing for software programs written in C++. ACRE allows developers without knowledge in AOP to use aspects to test their programs without modifying the behavior of their source code. ACRE uses a domain-specific language (DSL), which statements testers insert into the source code like comments to describe the aspects to be used. The presence of DSL statements in the code does not modify the program's compilation and behavior. ACRE parses the DSL statements and automatically generates appropriate aspects that are then weaved into the source code to identify bugs due to memory leaks, incorrect algorithm implementation, or interference among threads. Thanks to the use of aspects and ACRE, testers can add or remove tests easily. Using an aspect generated by ACRE, we find a memory leak in a complex C++ software program, NOMAD, used in both industry and research. We also verify a crucial mathematical point of the algorithm behind NOMAD and collect data to find possible interference bugs, in NOMAD.**

***Keywords*-AOP, C++, NOMAD, interference bug pattern, memory testing, invariant testing**

I. Introduction

Software programs have become increasingly important, with human lives depending on them. They must therefore be as reliable as possible. Moreover, software maintenance, which involves the correction of bugs, represents an important part of the costs of the development cycle [1]. Testing is a means to reduce these costs and the risk of failure.

Programs written in C++ present different challenges to testers when compared to programs written in Java. The absence of garbage collection and the use of pointers in C++, although useful, increase the complexity of dealing with memory and, therefore, the probability of bugs that involve memory leaks [2]. Techniques have been developed to test for null-pointer exceptions, buffer overflows, and memory leaks [3], [4]. The use of pointers in C++ can also have an impact on invariant testing. Testers may have to test classes to verify that attributes are never nulls. Techniques have been proposed for class invariant testing [5], [6]. However, most of these techniques require developers to modify the source code of their programs for memory and invariant testing. Interference detection is another extra challenge for C++ as interference bugs may occur where unprotected shared variables are used. Many approaches exist to detect interference bugs [7], [8], but none of them allow a complete detection of every possible bug.

To avoid modifying the source code of programs under test, researchers can use various technologies, including aspect-oriented programming (AOP). AOP is based on the concept of separation of concerns and its creators claim that it may be useful for testers [9]. Researchers have been using AOP for unit and integration testing, focusing on programs written in Java or embedded programs. However, their use of AOP required either to modify the programs under test and–or to generate the aspects manually in an ad-hoc manner.

We propose a new approach to use AOP for testing: the use of automatically-generated aspects to test C++ programs. We develop ACRE, an Automated aspeCt cREator, that automatically generates testing aspects from the source code of programs under test. The generated aspects can be used for memory, invariant, and interference testing. We build ACRE atop a domain-specific language (DSL) that we use to describe the testing aspects within the source code. An important characteristic of our DSL is that, despite the inclusion of DSL statements in the source code, the code still compiles as before and its behavior does not change. Moreover, thanks to the DSL, testers do not need expertise in AOP. Indeed, we propose three types of pre-defined testing aspects to be embedded in C++ source code. We show that ACRE, thanks to the three types of aspects, allows (1) finding a difficult memory leak in a C++ program, NOMAD [10], (2) verifying a crucial mathematical point inside NOMAD, and (3) finding possible interference bugs in NOMAD.

The remaining of the paper is organised as follows: Section II introduces aspect-oriented programming. Section III reviews relevant work on testing AOP and using AOP codes, and on memory management in C++, invariant testing, and interference bugs. Section IV presents ACRE and our DSL, used for the automatic creation of the testing aspects. Section V describes our goals, perspectives, and research questions, and also talks about NOMAD, the tested program. Section VI reports and discussed our results. Section VII reports the threats that affect our approach and study while Section VIII concludes and sketches future work.

II. Background

This section introduces aspect-oriented programming.

Aspect-oriented programming (AOP) is a technology based on the concept of separation of concerns used with object-oriented programming (OOP). Separation of concerns aims to increase the modularity of a program: each module should have its own specific task and should not address the tasks of other modules. A basic example of concern is the logging of method called in a program. With OOP, developers must modify all methods to add logging code. With AOP, developers write an aspect that essentially says "log each time that we enter a method" and they do not need to modify the original methods as illustrated in Listing 1. The two major AOP languages[1] are AspectJ for Java and AspectC++ for C++.

Listing 1. Example of aspect

```
#include <stdio.h>
#include <stdlib.h>

    aspect loggingAspect{
        public:
          pointcut logMethods() =
              call("% %::%(...)");
          advice logMethods() : before() {
                printf("> Enter: %s\n",
                      JoinPoint::signature());
          }
    };
```

ACRE uses AspectC++ to generates aspects for C++. The main constructs provided by AspectC++ are *aspect*, *advice*, *pointcut*, and *join point* [11]. These constructs allow AspectC++ programs to access the underlying AspectC++ execution model, which is out of the scope of this paper[1]. An *aspect* is the equivalent of a class. It may contain pointcuts and advice declarations as well as attributes and methods. An *advice* links join points with actions. For example, the advice presented in Listing 1 prints a message before each call to a method. An advice can also add a new attribute to all the classes accessed by a join point. A *pointcut* is a set of join points corresponding to the same expression. This expression is a string containing a search pattern. In Listing 1, the expression is *call*("%% :: %(...)") and the pointcut is named logMethods. The advice runs before each call to a method (%::%) containing any parameter (...). Finally, a *join point* describes a point in the source code (depending on execution model of the programming language) where aspects can be woven. With the current implementation of AspectC++, a join point may refer to a method entry/exit, an attribute read/write, a type (class, struct, or union), or an object instanciation/access. In Listing 1, the join point is a call to a method. Advices can be executed *before*, *after*, or in place of (*around*) a join point. In AspectC++, developers can assign priorities to advices to order their execution.

Aspects are concretely declared in *.ah* files and woven into the C++ source code using a weaver called *ag++*, which weaves the aspects into the original source code at compilation time, producing a new source code that is immediately built into an executable file. Thus, ag++ replaces the regular compiler g++. Including the *.ah* file in the C++ source code may be required by AspectC++, if a static attribute of the aspect has to be initialised for example. However this inclusion could be automated.

III. Related Work

This section present relevant works on the use of aspect-oriented programming in testing, on memory management in C++, invariant testing, and interference bugs.

A. AOP in Testing

Li and Xie [12] claim that aspects make good stubs and drivers. Another advantage of AOP for unit testing is that testers do not have to modify the source code. This advantage was also discussed by Metsä et al. [13], who compared aspects with macros and test interfaces. However, they concluded that AOP should not be the only technique used for unit and integration tests. ACRE helps testers to use aspects for unit and integration testing.

For unit testing Xu et al. [14] created JAOUT, a framework and an aspect-oriented test description language to use AOP with JUnit. Knauber and Schneider [15] used JUnit and AspectJ to test variable cross-cutting concerns such as security, transaction management, and error handling in software product lines. D'Amorim et al. [16] proposed an approach that allows to verify dynamically that the execution of a program is correct. AOP is used for invariant testing during the execution of the tested programs. These approaches focused on Java programs. ACRE allows the use of AOP to test C++ programs and their particularities, like explicit use of pointers which may lead to null pointer exceptions, memory leaks or buffer overflows.

Pesonen et al. [17], [18], and Metsä et al. [19], from Nokia corporation, used aspects to test embedded C++ programs in Symbian OS. They chose aspects because of the features of embedded programs, such as limited resources. Similarly to Li and Xie [12], they first used aspects as stubs and drivers, building test harnesses for functions [17]. They then enhanced integration testing using aspects. They also tested the hardware components of their programs with aspects [18]. They focused on the device-specific parts of the code and checked, using aspects, that their programs worked even when some devices were not present or not activated. They highlighted five areas of functional testing where aspects can be useful [19]: memory, performance, robustness, reliability, and coverage. They concluded that aspects increase the overall test coverage of their programs.

[1]http://www.eclipse.org/aspectj/, http://www.aspectc.org/, http://www.aosd-europe.net/deliverables/d12.pdf

However, they focused only on embedded programs. With ACRE, we show that aspects can also be useful to test non-embedded C++ programs.

For Metsä et al. [19] and Mahrenholz et al. [20], there are two main problems when using aspects for testing. The first is the lack of tools, or interfaces for existing tools, that use aspects. The second is that weaving aspects into the code requires an extra step in the building process. ACRE can overcome these two problems because we provide a DSL and because the testing aspects are automatically generated from the source code extended with DSL statements describing the tests.

Farhat et al. [21] claimed that one disadvantage of using aspects for testing non-functional requirements is that AOP is a new technology that is not well known. ACRE overcomes this disadvantage because developers and testers can generate and use testing aspects even if they do not know AspectC++. That is because the DSL "hides" from the developers the use of AspectC++ and of its constructs.

Table I summarises and compares the different approaches to using AOP for testing programs. It shows that only our approach allows the automatic generation of aspects used to test C++ programs with various kind of tests, including memory, invariant and interference testing.

Who	Language			Generation		Tests	
	Java	emb. C++	C++	auto.	man.	unit. & int.	all
[12]	✓				✓	✓	
[14]	✓			✓		✓	
[15]	✓				✓	✓	
[16]	✓			✓			✓
[13]		✓			✓	✓	
[17]		✓			✓		✓
[18]		✓			✓		✓
[19]		✓			✓		✓
us			✓	✓			✓

Table I
COMPARISON OF THE DIFFERENT APPROACHES

As explained by Ceccato et al. [22], testing aspects and their integration in object-oriented source code is also challenging. Kumar et al. [23] analysed four approaches used to detect different faults due to the presence of aspects, such as incorrect changes in control flows, false postconditions or false state invariants. They highlighted the difficulties brought by the presence of aspects, and found that some faults, as incorrect changes in polymorphic calls, can not be found using the actual testing techniques. Xie and Zhao [24] proposed an approach that allows the use of automatic testing for aspects in AspectJ using existing testing tools for Java code. They however concluded that there are still some problems, concerning the evaluation of such tests as no benchmark exists, or for the development of a systematic approach for integration testing. Sokenou and Herrmann [25] used aspects to test other aspects. They found that aspects-oriented techniques are good to test aspect-oriented systems, but they require specificities for the aspect language used for the test that not all of aspect languages possess.

B. Memory in C++

Two major differences between Java and C++ are the absence of a garbage collector and the use of pointers in the latter, which force developers to be careful about memory management [26] and increase the risk of memory leaks. Gati [27] used AOP to log all the memory allocations and de-allocations of a C++ program, but had some problems dealing with templates. Mcheick et al. [28] tracked memory under and overflows using AOP.

Sioud [29] is one of many researchers to have implemented garbage collection for C++. He chose to use AOP instead of classical OOP to implement a reference-counting algorithm: one aspect increments a counter each time the *new* function is called and another decreases the same counter each time the *delete* function is called. At the end of the program, the counter is analysed. If it is zero, then all created objects have been deleted and all memory freed, else there is a memory leak.

Let us consider a simple example to illustrate Sioud's implementation: a program with two classes *A* and *B*. Listing 2 shows the code of this toy program. In the `main` function, we instantiate one object of class *A* and two of class *B*. At the end, we delete the object instantiated from A and one of the two objects instantiated from B. We have a memory leak because one object is not deleted and the counter indicates 1. This basic example will be considered throughout this paper.

However, a limitation of Sioud's approach is that it focuses only on the methods *new* and *delete* while objects can be instantiated and freed without using these functions (using functions *free* and *malloc* for example), so memory leaks may be missed. To overcome this limitation, we focus not only on the *new* and *delete* methods but directly on constructors and destructors. This allows us to deal with object-level memory.

C. Invariant Testing

Invariant testing allows to verify that a property of a method, a class, or an algorithm is always true. To test invariants, we can use assertions (in C++ or Java) or dedicated methods. Class invariants concern fields. We can have methods that verify that a field is never null or that each time a field is modified, it stays within a given range. For methods and algorithms, invariant testing can help verifying that the result of a method or an algorithm is correct, for example that a table is always sorted after a *sort* method, no matter the order of the elements in the providing table. Invariant testing can then be used to verify that the tested program runs correctly: if at a certain point an invariant is false, then there is an error in the program.

Listing 2. Source code of toy program

```
#include <stdio.h>
#include <stdlib.h>
#include "A.hpp"
#include "B.hpp"

int main(void) {
    A* a = new A();
    B* b = new B();
    B* c = new B();
    a->foo();
    a->moo();
    c->zoo();
    delete(a);
    b->roo();
    b->koo();
    delete(b);
    c->joo();
    return 0;
}
```

Invariant testing can be executed dynamically, using programs like Daikon [6]. They can also be used to detect buffer overflows [5], using program instrumentation and mutation. However, these methods require behavior modifications in the source code of the tested programs. Our approach allows invariant testing without modifying the source code behavior of the tested programs.

D. Interference Bug

Interference bugs happen when at least two threads of a same program try to write in the same unprotected variable in a very close time. It is a very common bug, but it is also one of the most difficult to remove [30]. Many approaches exist to test multi-threaded programs and try to find interference bugs, e.g. [7], [31], [32], [33]. Among them, CHESS [8] can detect the most bugs. However, it requires a complete enumeration of all possible schedules. Bhattacharya et al. [34] proposed a mathematical approach to help these tools find interference bugs. Their approach relies on the exact times when threads write in a variable and we show in Subsection VI-C that ACRE can provide these times.

Figures 1 represents the expected behavior of a multi-threaded program: a master thread has three slaves and each one access sequentially a shared variable, reading its value and then writing a new value. Figure 2 represents the same program with an unexpected behavior: a delay has been injected in Slave 1 just before the writing event. Write events of both Slave 1 and Slave 2 happen in a very closed time: there is a risk of having an interference bug.

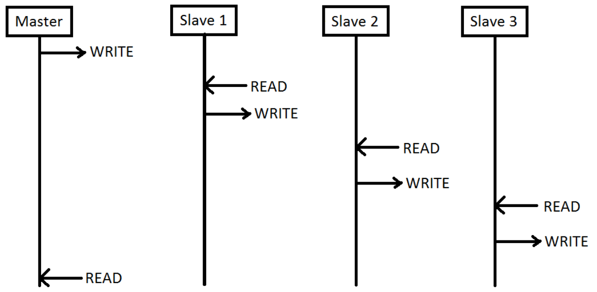

Figure 1. Normal behavior of a program, from [34]

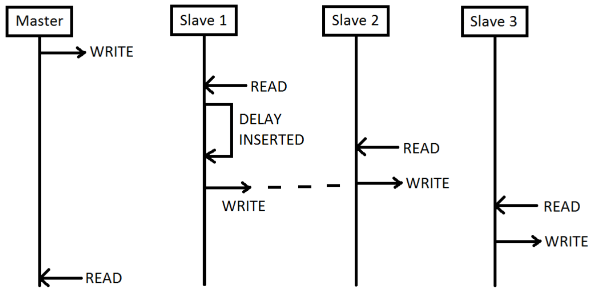

Figure 2. Behavior of a program with an injected delay, from [34]

IV. Approach

Our goal is to generate aspects that can be used for memory, invariant, and interference testing directly from the source code of a C++ program. We thus create a DSL, detailed below, to describe the tests. The DSL does not change the compilation or behavior of the program under test. We then create ACRE, a program that reads the DSL statements and generates the corresponding aspects.

A. Domain Specific Language

As there are no annotations in C++ in opposition to Java, we create our own DSL to describe the tests to be woven into C++ source code at compile time. We choose to embed the DSL statements into the C++ source code to ease the development of tests: developers/testers would write tests using this DSL as they would write comments for their code. We ensure that the source code compiles despite the addition of the DSL code while the DSL statements must also be identifiable. We therefore require the DSL lines describing a test aspect to start with the "////" symbol. All the DSL lines defining one test aspect must be consecutive and at least one non-DSL line must separate two aspect definitions. The first DSL line for a definition contains the name of the aspect and is defined via: *//// name: AspectName*. The second line contains the type of the aspect in the form: *//// type: AspectType*. ACRE currently supports four types of aspects: *counter*, *logging*, *timing*, and *checking*. Each type has its own characteristics, detailed below.

A *counter* aspect is used for memory testing. It first adds a static attribute to the class under test, indicated in the aspect definition. It then instruments the constructor and destructor of this class to increment the attribute when the constructor is called and to decrement it when the destructor is called. Finally, after the execution of the main method of the program, it prints the value of this attribute. In addition to the first two mandatory lines, the definition of a counter may include two optional lines. The first contains the name of the class to count: *//// className: ClassName*. If this line does not appear in the definition, ACRE uses the name of the file where the aspect definition appears as the class name. The second optional line contains the namespace to which the chosen class belongs: *//// namespace: NamespaceName*, which is used by AspectC++ to identify the classes that are in a given namespace. If this line is not present ACRE will search in the file for a namespace definition. If none is found, the namespace is left undefined. Listing 3 shows an example of the DSL code to create a counter. This code helps to find memory leaks in real programs as discussed in Section VI.

Listing 3. DSL used to generate counter aspect
```
//// name: EVPCG
//// type: counter
```

A *checking* aspect can be used for invariant testing. It contains just one advice declaration. After the first two mandatory lines, the third line must contain the name of a method and when the aspect must be executed (before, after, or instead of the method). The format of this line is: *//// before / after / around: [returnType__][className::]MethodName[(arguments)]*. The class name may contain the namespace and the class name, separated by "*::*". After these three lines, the developer describes the aspect. Five descriptions lines are available and can be combined.

The first line allows the creation of a variable: *//// create: type && VariableName*. If developers want to use the constructor, for example *A a(x)*, then the attribute *VariableName* must be the name of the variable followed by parentheses containing the required arguments. In the previous example, the DSL line is *//// create: A && a(x)*. For initialisation, *VariableName* must contain the complete line required to initialise the variable, in addition to its name. For example, for *int i = 0*, the line is *//// create: int && i=0*. The second description is used to give a value to a variable using a method: *//// store: type && howToGet*. The attribute *howToGet* must have the form *a.b()* or $a \rightarrow b()$, where *b()* is a method returning the variable. The next lines are: *//// do: if(condition)* and*//// do: endIf*. These lines allow the developers to create *if* statements. The final possibility is: *//// do: line*, which contains the lines that the developer wants to use without modification. Listing 4 shows an example of the DSL code required to create a checking aspect, as discussed in Section VI.

Listing 4. DSL used to generate checking aspect
```
//// name: Deltas
//// type: checking
//// after: iteration
//// store: NOMAD::Signature * &&
     this->_p.get_signature()
//// store: int && signature->get_n()
//// store: NOMAD::Mesh &&
     signature->mesh()
//// store: int && mesh.get_mesh_index()
//// create: NOMAD::Point && delta_m(n)
//// create: NOMAD::Point && delta_p(n)
//// do: mesh.get_delta_p(delta_p,
                    mesh_index)
//// do: mesh.get_delta_m(delta_m,
                    mesh_index)
//// do: for-i-0-n
//// do: if (delta_m[i] > delta_p[i])
//// do: printf("Error")
//// do: endIf
//// do: endFor
```

A *timing* aspect is used for interference testing. It gives the access times in read and–or write of the selected attribute. The third line must be *//// time : second / nanosecond*, to indicate if the obtained times are in seconds or nanoseconds. Then, the description must contain at least one of the three following lines: *//// attribute : AttributeName [&& set && get]*, *//// set : SetterSignature* et *//// get : GetterSignature*. They precise which attribute is concerned by the aspect. *SetterSignature* and *GetterSignature* are the complete signature of the setter/getter method. *AttributeName* is the name of the field. If this line is used, ACRE considers that the attribute belongs to the class where the description is found, and if the attributes *set* and *get* are provided, ACRE considers that the setter is *set_AttributeName(...)* while the getter is *get_AttributeName(...)*. If the concerned methods signatures are not these ones then the line *//// attribute : AttributeName [&& set && get]* must not be used. Listing 5 shows an example of the DSL code required to create a timing aspect, as discussed in Section VI.

Listing 5. DSL used to generate timing aspect
```
//// name: TimingAspect
//// type: timing
//// time : nanosecond
//// attribute : mesh_index && set && get
```

A *logging* aspect is used to obtain a trace of the program and allows developers to print a statement each time the program enters or leaves a method. After the first two mandatory lines, a third mandatory line specifies when the aspect should be called (when the method is called or executed) and the name of the associated method: *//// method && call / execution && MethodName*. MethodName must contain the complete signature of a method if the developer uses the aspect for a single method or a search pattern similar to a signature, i.e. a return type, a class name, a method name, and two parentheses containing the types of the arguments separated by commas (if any). After this line, there must be one or more lines of the form: *//// when: before / after*, which indicate when to print the statement:

before or after the method. Listing 6 shows an example of the DSL code required to create the logging aspect shown in Listing 1.

Listing 6. DSL lines to generate logging aspect
```
//// name: loggingAspect
//// type: logging
//// methods && call && % %::%(...)
//// when: before
```

Table II proposes a synthesis of the DSL syntax.

Table II
DSL SYNTHESIS.

Type	Line	Description
All	//// name : AspectName	Name of the generated aspect
	//// type : AspectType	Type of the generated aspect
Counter	//// className : ClassName	Class concerned by the aspect
	//// namespace : Name	Namespace of the class
Logging	//// method && call / execution && MethodName	Method concerned by the aspect
	//// when : before / after	Aspect executes before or after the method
Timming	//// time : second / nanosecond	Time unit
	//// attribute : AttributeName [&& set && get]	Attribute concerned by the aspect
	//// set : SetterSignature	Setter of the attribute
	//// get : GetterSignature	Getter of the attribute
Checking	//// before / after /around : [return_type__][className::] MethodName[(arguments)]	The aspect is executed before or after the concerned method
	//// create : type && VariableName	Create a type variable
	//// store : type && howToGet	Give a value to a variable
	//// do : if(condition)	Create an if bloc
	//// do : endIf	
	//// do : while(condition)	Create a while bloc
	//// do : endWhile	
	//// do : for-variableName--begin-end	Create a for bloc
	//// do : endFor	
	//// do : line	Write line in the aspect

B. ACRE

When the aspect definitions are embedded in the source code by the testers, we use ACRE to read them and to generate the corresponding aspects. ACRE is written in Java and is available on-line[2] under the GPL license. ACRE takes as input a folder containing C++ source code files (.hpp or .cpp), compilable or not. ACRE then goes through the folder and parses every .hpp and .cpp file. If ACRE finds the definition of a testing aspect, then, depending on the type of the aspect, it calls the corresponding sub-parser, analyses the definition, and generates the appropriate AspectC++ code. If a problem is found in an aspect definition, ACRE does not create this aspect but still parses the remaining files for other definitions. ACRE then creates a .ah file whose name comes from the aspect definition and then writes the relevant aspect code. It also automatically translates some C++ keywords into AspectC++ keywords when needed. For example, *this* does not exist in AspectC++ and must be replaced by $tjp \rightarrow that()$.

[2]http://web.soccerlab.polymtl.ca/~ducloset/ACRE

For the checking type, we saw in the previous subsection that a *store* action is used to give a value to a variable using a method. However, the name of the variable is not given in the DSL lines. ACRE checks whether or not the method obtained by the store action is a getter. If it is (i.e. if the method starts with *get_*), then the variable is given the name of the used attribute. For example, if the provided line is *//// store: int &&* $this \rightarrow get\_size()$ the generated line is *int size =* $tjp \rightarrow that() \rightarrow get\_size()$;. If the method is not a getter, then the variable is given the method name, e.g. *//// store: int && a.size()* generates *int size = a.size();*.

For the logging type, the generated aspect prints the complete signature of a method each time the program enters or leaves it. We use the AspectC++ keyword *JointPoint::signature()* to get this signature. We also create one advice for each *when* line in the description. Thus, for an aspect that writes before and after each call to each method of the program, the DSL contains two *when* lines and the generated aspect contains two advices. We also use an attribute inside the aspect to take care of indentation.

For a counter aspect, a change to the OO source code is required: we must initialise the attribute representing the counter. ACRE reminds the testers of this initialisation by providing them with the piece of code to be inserted in their source code. We could have automatically added these two lines, but we prefer to let the developers choose how to modify their code so that they know where these lines are if they want to remove them. Listing 7 gives an example of such an aspect.

V. EMPIRICAL STUDY

The *goal* of our study is first to see if AOP can be used to test C++ programs. If this is possible, we aim to provide a tool that automatically generates testing aspects from the source code. Different types of aspects can be created, which allow memory, invariant, and interference testing.

The *quality focus* of our study is to use our tool to automatically generate aspects and to detect at least one bug of each type in a C++ program, NOMAD, using these aspects. We use ACRE to test NOMAD because it is a C++ program used in both industry and research, which promotes the generalisation of our results.

The *perspective* of our study is that of developers and testers interested in testing C++ programs without having to modify their source code. Our approach can also be useful for developers who want to test programs without knowledge of testing techniques and of AOP.

We investigate three research questions:

- ***RQ1:*** *Can generated aspects be used to do memory testing for C++ programs ?*
- ***RQ2:*** *Can generated aspects be used to do invariant testing for C++ programs ?*
- ***RQ3:*** *Can generated aspects be used to find interference bugs in C++ programs ?*

A. Analyses

To answer RQ1 and convince ourselves that AOP can be used to test C++ programs, we first try to find a memory leak in a toy program, described in Section III-B. We then find a memory leak in a real program, NOMAD, with a generated aspect. To answer RQ2, we verify a crucial mathematical point of the algorithm implemented in NOMAD. We mutate the source code of NOMAD and use a generated aspect to detect a wrong algorithm implementation. Finally, to answer RQ3, we use generated aspects to get access times for both reads and writes to a shared variable in NOMAD. These times are then provided to another approach, [34], to help finding possible interference bugs.

For RQ1, our oracles are a memory leak introduced on purpose in the toy program and a user bug report concerning a memory leak in NOMAD. For RQ2, our oracle is the source code mutated independently by a developer of NOMAD. For RQ3, our oracle are the developers validating the interference bugs.

B. NOMAD

NOMAD [10] is a C++ program that can be used with any operating system. It is designed to solve difficult black-box optimization problems, using the mesh adaptive direct search (MADS) algorithm [35], which involves minimizing a constrained objective function. A black-box is a function that may be noisy, could be expensive to compute, and may fail to evaluate even at feasible points. Such functions are often encountered in engineering applications, for example the computation of the shape of a wing. NOMAD is used both by researchers and in industry. It is available on-line[3] under the LGPL license and has been downloaded more than 3,000 times since 2008. It is composed of 59 classes for around 48k LOCs, and is thus representative of industrial C++ programs.

VI. Results

We explain how we answer our research questions and show the generated aspects used for that.

[3]http://www.gerad.ca/nomad

A. RQ1

We first embed an aspect into the source code of the toy program described in Section III-B to detect a memory leak. This aspect behaves as follows: each time a constructor of the class specified in the *pointcut* (i.e. A or B) is called, a counter is incremented. Each time a destructor of these classes is called, the same counter is decremented. Then, after the execution of any main method (*% main(...)*), we print the counter. The only modification to the source code of our toy program is the initialisation of the counter in one of the .cpp files. This modification is not specific to our approach, it comes from AspectC++: we must initialise static attributes of aspects in a *.cpp* file. We then use ag++, the AspectC++ weaver, to compile our program with this aspect woven into its source code. We then run our program and, as expected, the line *Final count of A or B: 1* appears. The memory leak has been successfully detected using AOP.

We then investigate NOMAD. A user reported a bug concerning a memory leak when running NOMAD on Windows (report available upon request). We generated one aspect per class, i.e. one counter per class. We then add the counter initialisations to the source code, modify the makefile to use ag++ instead of g++, and compile NOMAD. We run NOMAD on the problem instance reported in the bug report and find that 74 objects of the class *Eval_Point* are instantiated but never freed. This result is given by the aspect shown in Listing 7, which is produced by the two DSL lines presented in Listing 3.

This result allows us to answer RQ1: yes, ACRE can be used to perform memory testing for C++ programs.

B. RQ2

The checking aspect showed in Listing 8 aims to verify that a crucial mathematical point of the MADS algorithm is always satisfied by invariant testing. This point concerns the step size of the algorithm, which is essential for the convergence of the solving method. We can investigate this invariant by generating an aspect without modifying the NOMAD's behavior. To generate the aspect we use the DSL statements in Listing 4. We do not find any bugs, so this part of the algorithm works as expected for all the 37 problems that the NOMAD team of developers and us tried. However, to verify that we can find a bug if any, the NOMAD developers team mutated the NOMAD code to have a wrong implementation of this algorithm, by changing an operator (an addition becomes a subtraction). After this mutation, the invariant that the aspect checks is false as expected.

This result allows us to answer RQ2: yes, ACRE can be used to perform invariant testing for C++ programs.

C. RQ3

For interference testing, as explained in Section III-D, aspects are used to get access times for both read and write

Listing 7. Generated aspect for memory testing

```
#include <stdio.h>
#include <stdlib.h>

aspect EVPCG{
  public: static int _Eval_PointCount;
  // Do not forget to initialize this
  // variable in the source code !!
  // Just copy these lines into the
  // appropriate .cpp file
  // #include "EVPCG.ah"
  // int EVPCG::_Eval_PointCount = 0;

  pointcut Eval_PointCounted() =
       "NOMAD::Eval_Point";
  advice Eval_PointCounted() : slice struct{
    class Eval_PointCount{
       public: Eval_PointCount(){
          EVPCG::_Eval_PointCount++;}
       public: ~Eval_PointCount(){
          EVPCG::_Eval_PointCount--;}
    } Eval_Point_counter;
  };

  advice execution("% main(...)") : after(){
    printf("Final count of Eval_Point :
              %d\n", _Eval_PointCount);
    if(_Eval_PointCount > 0)
       printf("Memory leak !!\n");
  }
};
```

Listing 8. Generated aspect for invariant testing

```
#include <stdio.h>
#include <stdlib.h>
#include "Mads.hpp"

aspect Deltas{
  advice execution("% ...::iteration(...)")
          : after(){
   NOMAD::Signature * s =
        tjp->that()->_p.get_signature();
    int n = s->get_n();
   NOMAD::Mesh mesh = s->get_mesh();
    int l = mesh.get_mesh_index();
   NOMAD::Point delta_m(n), delta_p(n);
    mesh.get_delta_p(delta_p, l);
    mesh.get_delta_m(delta_m, l);
    for(int i = 0 ; i < n ; ++i){
      if(delta_m[i] > delta_p[i]){
       printf("Error :
         delta_m[%i] > delta_p[%i]\n", i, i);
       exit(0);
      }
    }
  }
};
```

events. To automatically generate a testing aspect to collect timings, we use the DSL statements shown in Listing 5. The corresponding aspect, shown in Listing 9, gives us all the access times for the attribute *Mesh_index*, which is used in the MPI version of NOMAD. MPI[4], for Message Passing Interface, is a communication protocol used in parallel programming.

[4]http://www.mcs.anl.gov/research/projects/mpi/

Once these times are recorded, we provide them as an input to the approach proposed by Bhattacharya et al. [34], which gives us the optimal place to inject a delay (and its duration) in the source code so that we maximize the possibility of having an interference bug pattern. After injecting the given delay in the NOMAD source code, we found that indeed an interference bug may occurs for the attribute *Mesh_index*.

This result allows us to answer RQ3: yes, ACRE can be used to perform interference testing for C++ programs.

Listing 9. Generated aspect for interference testing

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
using namespace std;

aspect timingMeshIndex {
 public:
  struct timeval courant;
  long timeCourant;

  advice execution("% NOMAD::Mesh::
        set_mesh_index(...)"): after() {
    gettimeofday(&courant, NULL);
    timeCourant = courant.tv_usec;
    printf("Ecriture de mesh_index :
        %ld \n", timeCourant);
  }
  advice execution("% NOMAD::Mesh::
        get_mesh_index(...)") : after() {
    gettimeofday(&courant, NULL);
    timeCourant = courant.tv_usec;
    printf("Lecture de mesh_index :
        %ld \n", timeCourant);
  }
};
```

D. Discussion

As shown in previous Section, we can detect a memory leak in NOMAD using an aspect automatically generated by ACRE. However, we could argue that this bug was actually introduced by the aspect and that it is not the one mentioned in the bug report and is not present in the not-tested version of NOMAD. To verify if the found bug was not introduced by ACRE and the underlying AspectC++ code, we searched for the leak without using any aspects. We modified the source code of NOMAD, adding a static integer *counter* as an attribute in the class *Eval_Point*, initialising it to 0. We then increment this counter in the constructor and decrease its value in the destructor of *Eval_Point*. Finally, we printed this counter at the end of the main method (we thus manually accomplish the task previously performed by the aspect). We then compiled this code as any NOMAD developer would do and ran NOMAD on the memory-leak problem. We find the same leak as before: 74 objects of type *Eval_Point* are created but not deleted. This analysis shows that the leak found was not introduced by the aspect but is a real bug. We communicated the bug to the NOMAD team of developers and they were able to fix it.

The presented memory leak was also found by the NOMAD team using Valgrind. However, Valgrind indicated only that a leak was there, but not which class was concerned. On this point, our approach is more precise than Valgrind, in addition of being faster because an injected aspect does not change in a significant way the execution time of NOMAD, compared to the use of Valgrind.

We showed in Section IV that timing aspects concern only attributes that are available using getters and setters. This is a limitation of our approach, due to internal limitation of AspectC++. With the current version of AspectC++ execution model, we cannot declare pointcut on variables and we cannot declare advice on other code elements than classes and methods.

Counter aspects may be subjects to interference problems. We showed in Section IV that these aspects increment and decrement a static attribute each time a constructor or a destructor of a class is called. However, we could not find a case where two instances of the same class tried to access the attribute at the same time and so we do not know yet what could happen in such a case. A part of future work will be to test this point and see if our results are still valid in such cases.

With the current version of the DSL, counter aspects cannot be used on other types than classes. Allocation and deallocation of arrays could have been done, using the malloc and free functions, however there is no possibility to distinguish two different arrays. Counter aspect could then only say that there exist some arrays that are non-deallocated, but could not say exactly which ones.

It is easy to add a new type of aspect to the DSL: apart from the first two lines, the aspect definition is treated by the parser that corresponds to its type. Adding a new type of aspect means adding a new parser. We provide an API in the source code of ACRE to help developers who want to create a parser for a new type. Using this API involves adding a few lines to the source code of ACRE and implementing few well-defined methods.

VII. Validity Threats

An *internal validity* threat is that the bugs found may be caused by the aspects added to the source code. The user must ensure that the test to be performed does not introduce new bugs. We report that the memory leak detected by our AOP approach can also be found without using aspects.

An *external validity* threat is that ACRE has been used only under Linux. However, it is written in Java and can therefore be used on any operating program with the appropriate Java virtual machine (v1.6). Furthermore, AspectC++ is available for every operating systems, which suggests that our approach can be used on any platform.

Another *external validity* threat is that ACRE has only be applied on NOMAD, a large C++ program used in industry and research. Yet, future work includes applying ACRE on other programs, such as ffmpeg of firefox.

Regarding *reliability validity*, the toy program, the source code of ACRE (under the GPL license)[2], and the source code of NOMAD (under the LGPL license)[3] are publicly available on the Internet.

VIII. Conclusion and Future Work

We used AspectC++ to perform memory, invariant, and interference testing in non-embedded C++ programs. To the best of our knowledge, using AOP to generate tests has previously been tried only with Java programs and embedded C++ programs. Moreover, AOP was mainly used for unit and integration testing in conjunction with other non-AOP techniques.

We proposed ACRE, an Automated aspeCt cREator, that automatically generates testing aspects from the source code of tested programs. We build ACRE atop a domain-specific language (DSL) that we use to describe the testing aspects and whose statements are embedded into the C++ source code without impacting its compilation and behavior. ACRE implements a parser for this DSL and automatically generates testing aspects from the DSL statements embedded in the source code. Developers who use our approach do not need expertise in AOP development. We applied ACRE to an industrial program, NOMAD. We showed that ACRE can be used to find real bugs in NOMAD: we were able to find a memory leak, to verify a mathematical invariant, and to show that interference bug may occur under certain condition in NOMAD.

In future work, we will extend ACRE to generate a wider range of aspect types, for example to test pre- and post-conditions. We will also extend ACRE to generate aspects in AspectJ, to use our approach on Java programs. We also plan to add a graphical user interface to ACRE to ease the use of its DSL, and to make usability test of ACRE with developers. Finally, we would like to avoid placing the aspect definitions in the source code (for example, by adding them into external files), which could allow ACRE to be used without extra definitions in the source-code files.

Acknowledgment. This work has been partly funded by the NSERC Research Chairs in Software Patterns and Patterns of Software.

References

[1] B. P. Lienz and E. F. Swanson, *Software Maintenance Management.* Addison-Wesley Longman Publishing Co., Inc., 1980.

[2] H. Schildt, *The Art of C++.* McGraw-Hill Osborne Media, 2004.

[3] D. L. Heine and M. S. Lam, "A practical flow-sensitive and context-sensitive C and C++ memory leak detector," in *PLDI*, 2003.

[4] R.-G. Xu, P. Godefroid, and R. Majumdar, "Testing for buffer overflows with length abstraction," in *ISSTA*, 2008.

[5] F. Zeng, M. Chen, K. Yin, and X. Wang, "Research on buffer overflow test based on invariant," in *9th IEEE Int. Conf. on Computer and Information Technology*, 2009.

[6] M. D. Ernst, J. H. Perkins, P. J. Guo, S. Mccamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," in *Science of Computer Programming*, 2006.

[7] S. Park, S. Lu, and Y. Zhou, "Ctrigger: exposing atomicity violation bugs from their hiding places," *SIGPLAN Not.*, vol. 44, no. 3, pp. 25–36, 2009.

[8] M. Musuvathi, S. Qadeer, and T. Ball, "Chess: A systematic testing tool for concurrent software," 2007.

[9] T. Elrad, M. Aksit, G. Kiczales, K. Lieberherr, and H. Ossher, "Discussing aspects of AOP," *Commun. ACM*, vol. 44, no. 10, pp. 33–38, 2001.

[10] S. Le Digabel, "Algorithm 909: NOMAD: Nonlinear optimization with the MADS algorithm," *ACM Transactions on Mathematical Software*, vol. 37, no. 4, pp. 44:1–44:15, 2011.

[11] O. Spinczyk, D. Lohmann, and M. Urban, "AspectC++: Extension de la programmation orienté aspect pour C++," *Software Developer's Journal*, no. 6, 2005.

[12] X. Li and X. Xie, "Research of software testing based on AOP," in *3rd International Conference on Intelligent Information Technology Application*, 2009.

[13] J. Metsä, M. Katara, and T. Mikkonen, "Comparing aspects with conventional techniques for increasing testability," in *Int. Conf. on Software Testing, Verification, and Validation*, 2008.

[14] G. Xu, Z. Yang, H. Huang, Q. Chen, L. Chen, and F. Xu, "Jaout: Automated generation of aspect-oriented unit test," in *11th Asia-Pacific Software Engineering Conference*, 2004.

[15] P. Knauber and J. Schneider, "Tracing variability from implementation to test using AOP," in *International Workshop on Software Product Line Testing*, 2004.

[16] M. d'Amorim and K. Havelund, "Event-based runtime verification of java programs," in *3rd int. workshop on Dynamic analysis*, 2005.

[17] J. Pesonen, "Extending software integration testing using aspects in Symbian OS," in *Testing: Academic & Industrial Conference on Practice and Research Techniques*, 2006.

[18] J. Pesonen, M. Katara, and T. Mikkonen, "Production-testing of embedded systems with aspects," in *Haifa Verification Conference*, 2005.

[19] J. Metsä, M. Katara, and T. Mikkonen, "Testing non-functional requirements with aspects: An industrial case study," in *ICQS*, 2007.

[20] D. Mahrenholz, O. Spinczyk, and W. Schrder-Preikschat, "Program instrumentation for debugging and monitoring with aspectc++," in *Int. Symp. on Object-Oriented Real-Time Distributed Computing*, 2002.

[21] S. Farhat, G. Simco, and F. J. Mitropoulos, "Using aspects for testing nonfunctional requirements in object-oriented systems," in *IEEE SoutheastCon*, 2010.

[22] M. Ceccato, P. Tonella, and F. Ricca, "Is AOP code easier or harder to test than OOP code?" in *AOSD*, 2005.

[23] M. Kumar, A. Sharma, and S. Garg, "A study of aspect oriented testing techniques," in *ISIEA*, 2009.

[24] T. Xie and J. Zhao, "Perspectives on automated testing of aspect-oriented programs," in *3rd workshop on Testing aspect-oriented programs*, 2007.

[25] D. Sokenou and S. Herrmann, "Aspects for testing aspects?" in *1st workshop on Testing aspect-oriented programs*, 2005.

[26] H.-J. Boehm, "Advantages and disadvantages of conservative garbage collection," 2012, http://www.hpl.hp.com/personal/Hans_Boehm/gc/issues.html.

[27] M. Gati, "Analyzing run-time component memory consumption with aspect-oriented techniques," Stan Ackermans Institute, Tech. Rep., 2004.

[28] H. Mcheick, H. Dhiab, M. Dbouk, and R. Mcheik, "Detecting type errors and secure coding in C/C++ applications," in *ACS/IEEE Int. Conf. on Computer Systems and Applications*, 2010.

[29] A. Sioud, "Gestion de cycle de vie des objets par aspects pour C++," Master's thesis, UQaC, 2006.

[30] U. Software Quality Research Group, Faculty of Science, "Concurrency anti-pattern catalog for java," 2012, http://faculty.uoit.ca/bradbury/concurr-catalog/.

[31] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur, "Framework for testing multi-threaded java programs." *Concurrency and Computation: Practice and Experience*, pp. 485–499, 2003.

[32] Y. Ben-Asher, E. Farchi, and Y. Eytani, "Heuristics for finding concurrent bugs," in *17th Int. Symp. on Parallel and Distributed Processing*, 2003.

[33] P. Joshi, M. Naik, C.-s. Park, and K. Sen, "Calfuzzer: An extensible active testing framework for concurrent programs," in *21st Int. Conf. on Computer Aided Verification*, 2009.

[34] N. Bhattacharya, O. El-Mahi, E. Duclos, G. Beltrame, G. Antoniol, S. Le Digabel, and Y.-G. Guéhéneuc, "Optimizing threads schedule alignments to expose the interference bug pattern," in *4th SSBSE*, 2012.

[35] C. Audet and J. Dennis, Jr., "Mesh adaptive direct search algorithms for constrained optimization," *SIAM Journal on Optimization*, vol. 17, pp. 188–217, 2006.