C++11/14 Mutation Operators Based on Common Fault Patterns

Ali Parsai(✉), Serge Demeyer, and Seph De Busser

University of Antwerp, Middelheimlaan 1, 2020 Antwerp, Belgium
{ali.parsai,serge.demeyer}@uantwerpen.be, seph.debusser@gmail.com

Abstract. The C++11/14 standard offers a wealth of features aimed at helping programmers write better code. Unfortunately, some of these features may cause subtle programming faults, likely to go unnoticed during code reviews. In this paper we propose four new mutation operators for C++11/14 based on common fault patterns, which allow to verify whether a unit test suite is capable of testing against such faults. We validate the relevance of the proposed mutation operators by performing a case study on seven real-life software systems.

Keywords: Software testing · Mutation testing · C++11/14 Mutation operators

1 Introduction

Nowadays, the process of software development relies more and more on automated software tests due to the developers interest in testing their software components early and often. The level of confidence in this process depends on the quality of the test suite. Therefore, measuring and improving the quality of the test suite has been an important subject in literature. Among many of the studied techniques, mutation testing is known to perform well for improving the quality of the test suite [10].

The idea of mutation testing is to help identify software faults indirectly by improving the quality of the test suite through injecting an artificial fault (i.e. generating a *mutant*) and executing the unit test suite to see whether the fault is detected [19]. If any of the tests fail, the mutant is said to detected, thus *killed*. On the other hand, if all the tests pass, the test suite failed to detect the mutant, thus the mutant *survived*. However, some mutants result in code which does not pass the compiler and these are called *invalid mutants*. And in other situations, a mutant fails to change the output of a program for any given input hence can never be detected—these are called *equivalent mutants*.

A mutant is created by applying a transformation rule (i.e. *mutation operator*) to the code that results in a syntactic change of the program [9]. Given an effective set of mutation operators, mutation testing can help developers identify the weaknesses in the test suite [1]. Nevertheless, designing effective mutation

I. Medina-Bulo et al. (Eds.): ICTSS 2018, LNCS 11146, pp. 102–118, 2018.
https://doi.org/10.1007/978-3-319-99927-2_9

operators requires considerable knowledge about the coding idioms and the common programming faults often made in the language [9]. More importantly, good mutation operators should maximize the likelihood of *valid* and *non-equivalent* mutants [4].

The first set of mutation operators were reported in King et al. [12]. They were later implemented in the tool Mothra which was designed to mutate the programming language FORTRAN77. With the advent of the object-oriented programming paradigm, new mutation operators were proposed to cope with specific programming faults therein [11]. This is a common trend in mutation testing: languages evolve to get new language constructs; some of these constructs cause subtle programming faults; after which new mutation operators get designed to shield against these common faults. For example, with the evolution of Java related languages, mutation operators have been designed to account for concurrent code [2], aspect-oriented programming [7], graphical user interfaces [18], and Android applications [6].

The C++11/14 standard (created in 2011 and 2014 respectively) offers a wealth of features aimed at helping programmers write better code [20]. Most notably there is more type-safety and compile-time checking (e.g. static_assert, override). Unfortunately, the standard also provides a few features that may cause subtle faults (e.g. lambda expressions, list initialization, ...). Our goal is to identify these sources of common faults and introduce new mutation operators that address them. While it is possible that some subset of these faults are addressed by C++99 mutation operators, previous experience shows targeted mutation operators prove useful in improving the test suite quality further [3,5]. In this study, we seek to answer the following research questions:

- **RQ1.** Which categories of C++11/14 faults are most likely to be made by programmers, and what are the corresponding mutation operators?
- **RQ2.** To what extent do these mutation operators create valid, non-equivalent mutants?

The rest of this paper is structured as follows: In Sect. 2 we provide the necessary background information about this study, and briefly discuss the related work. In Sect. 3 we discuss our approach to answering our research questions, and show our results in Sect. 4. Finally, we present our conclusions in Sect. 5 and highlight the future research directions rooted in this work.

2 Background and Related Work

In this section we provide the necessary background information needed to comprehend the rest of the article and discuss the related work. First, we describe mutation testing and its related concepts. Then, we describe the new C++11/14 features, focusing on subtle faults that may be revealed via mutation testing.

2.1 Mutation Testing

Mutation testing is the process of inserting bugs into software(*Mutants*) using a set of rules(*Mutation Operators*) and then running the accompanying test suite for each inserted mutant. If all tests pass, the mutant survived. If at least one test fails, the mutant is killed. If the mutant causes an error during compilation of the production code, it is invalid. A valid mutant that does not change the semantics of the program, thus making it impossible to detect, is called equivalent.

An equivalent mutant is a mutant that does not change the semantics of the program, i.e. its output is the same as the original program for any possible input. Therefore, no test case can differentiate between an equivalent mutant and the original program, which makes it undesirable. The detection of equivalent mutants is undecidable due to the halting problem [16]. The only way to make sure there are no equivalent mutants in the mutant set is to manually inspect and remove all the equivalent mutants. However, this is impractical in practice. Therefore, the aim is to generate as few equivalent mutants as possible.

Mutation operators are the rules mutation testing tools use to inject syntactic changes into software. Most operators are defined as a transformation on a certain pattern found in the source code. The first set of mutation operators ever designed were reported in King et al. [12]. These mutation operators work on basic syntactic entities of the programming language such as arithmetic, logical, and relational operators. Offutt et al. came up with a selection of few mutation operators that are enough to produce high quality test suites with a four-fold reduction of the number of mutants [17]. Kim et al. extended the set of mutation operators for object-oriented programming constructs [11].

Because of the complexity of parsing C++, building a mutation testing tool for C++ is almost equivalent to building a complete compiler [8]. It is only with modern tooling, e.g. the Clang/LLVM compiler platform, that it became possible to write such tools without an internal parser.

Kusano et al. developed CCmutator, a mutation tool for multi-threaded C/C++ programs that mutates usages of POSIX threads and the C++11 concurrency constructs, but works on LLVM's intermediate representation instead of directly on C++ source code [13]. Delgado-Perez et al. have expanded on the work done for the C language by adding class mutation operators, and created a set of C++ mutation operators [5]. In addition, they show that the class mutation operators compliment the traditional ones and help testers in developing better test suites.

2.2 C++11/14

C++11 was introduced in 2011 with the goal of adapting C++ and its core libraries to modern use cases of the language (e.g. multi-threading, genetic algorithms, ...). This release was followed by C++14 in 2014 with similar goals. The introduction of C++11/14 has changed the language to the point that earlier iterations of the language are dubbed the classical C++, and modern C++[1]

[1] http://www.modernescpp.com/index.php/what-is-modern-c.

starts with C++11/14. The release of the standard was followed by real-time adoption in compilers such as Clang and G++.

Unfortunately, the C++11/14 standard also provides a few features that may cause subtle faults, thus where support in the form of new mutation operators would be desirable. In this subsection we briefly explain these features of C++11/14.

Range-Based for Loop. [http://en.cppreference.com/] is syntactic sugar made to simplify looping over a range of elements. For example, the following two loops are similar:

```
for(int i : v) {
std::cout << i << '\n'; }
```

```
for(int i=0; i<v.size(); i++) {
std::cout << v.at(i) << '\n'; }
```

Lambda Expressions. [http://en.cppreference.com/] allow for the definition of unnamed in-line functions. For example, in the following piece of code, lambda contains a function which *captures* `a` and `b` (they are available in the body of lambda as const expressions), takes an input parameter `x`, and returns a `bool`.

```
int a, b;
auto lambda = [a, b](int x) {return x > a + b;}
```

It is possible to have a default capture at the start of the capture list, e.g. '=' for by-value, or '&' for by-reference capture. This causes all variables referenced in the lambda body to be captured the specified way.

Move Semantics. [http://en.cppreference.com/] are introduced in C++11/14 to address the inefficiencies of copy construction when the copied value is deleted after the execution of the constructor. For example, the following code would be inefficient in C++03:

```
std::vector<int> v(ComputeLargeVector(1000));
```

In C++03, this code would create the vector in `ComputeLargeVector`, call the copy constructor for v, which copies all elements into a newly allocated buffer, and then destroys the original. With move semantics, `v` would simply copy the internal size, capacity, and pointer to the elements in the temporary vector and set the members of the temporary vector to 0.

To enable this, value categories[2] got redefined in C++11. Every expression is either an `lvalue`, an `xvalue`, or a `prvalue`. The difference between these value categories lies in two properties: whether or not they have identity (i.e. it is possible to determine whether two expressions are the same using an address), and whether they can be moved from (move semantics can bind to the expression).

[2] http://en.cppreference.com/w/cpp/language/value_category.

`lvalues` and `xvalues` have identity, while `xvalues` and `prvalues` can be moved from. All `rvalues` can bind to `rvalue` references, which are denoted by &&. For example, the signature of the move constructor of vector is:

```
vector<T> (vector<T>&&);
```

It is possible to convert an `lvalue` to an `xvalue` through `std::move`, which casts the object to an `rvalue` reference type.

Perfect Forwarding. [http://en.cppreference.com/] allow for forwarding of input arguments to other functions as-is. For example, the `emplace` family of functions in the standard containers accept any number of arguments and forward them to the constructor of the element type. The following template function constructs an object of type T with a given argument:

```
template<typename T, typename Arg>
T construct(Arg&& argument) {
  return T{std::forward<Arg>(argument)};
}
```

Because `Arg` is a template parameter, `Arg&&` is a forwarding reference [22]. This means that it will resolve to either an `lvalue` or an `rvalue` reference depending on `argument`. If `argument` is an `lvalue`, `std::forward` is a no-op, and if `argument` is an `rvalue` reference, it behaves the same way `std::move` does.

List Initialization. [http://en.cppreference.com/] is a new syntax introduced in C++11 that allows the initialization of an object from braced initial values. It expands the ability to construct structs and arrays using braced initializer to all types in C++. For example, the following is a valid syntax for creating and initializing an array of *int*:

```
int b {1,2,3,4,5};
```

Also, a type with a constructor that takes `std::initializer_list` as an argument can be initialized using this new syntax. For example, the following declaration of a `std::vector` creates a vector of integers with 5 elements:

```
std::vector<int> v{1,2,3,4,5};
```

3 Study Design

In this section, we discuss the design of our study. First, we explain our evaluation criteria, and then we describe the process by which we determine the fault categories and create mutation operators. Finally, we present the details of our data set.

3.1 Evaluation Criteria

RQ1. Which categories of C++11/14 faults are most likely to be made by programmers, and what are the corresponding mutation operators?
To evaluate the results of this question, the mutation operator needs to fulfill the following criteria:

- Can the mutation operator simulate a fault from the fault category we identified?
- Is it reasonable to assume that the software developer can create faulty code similar to the generated fault?

We look at guidelines provided by experts concerning the new standards and the common pitfalls mentioned therein. We search for such patterns and select those that can be reconstructed into a mutation operator.

RQ2. To what extent do these mutation operators create valid, non-equivalent mutants?

$$Mutation\ Operator\ Score = 1 - \frac{E - D}{T - I - D} \tag{1}$$

T = Total Number of Mutants, E = Number of Equivalent Mutants, D = Number of Easily-Detectable Equivalent Mutants, I = Number of Invalid Mutants

An effective mutation operator generates valid semantic faults. This means that mutation operators need to generate as few equivalent mutants as possible. We borrow this criterion from Delgado-Perez et al. who used it in their study [4]. It is also important for each mutant to be valid, i.e. the mutated program compiles without errors. To quantify the effectiveness of each mutation operator, we calculate the percentage of equivalent mutants among the valid mutants after filtering the easily-detectable equivalent mutants. The mutation operator score is then calculated by deducting the mentioned percentage from 100% (see Eq. 1). For each mutation operator, we provide methods to filter easily-detectable equivalent mutants.

To see how our operators work in real-life scenarios, we looked at seven open source projects that are using C++11/14 (see Table 1). Our analysis consists of applying our mutation operators to create all possible mutants. We do this by manually searching for the code patterns that match (using grep). Then, we manually categorize the resulting mutants into invalid, equivalent, and valid non-equivalent mutants. If a mutant did not change the semantics of the program, we classified it as an equivalent mutant. If the operator created a non-compilable program, we classified the mutant as invalid. Otherwise, we considered the mutant as valid non-equivalent.

3.2 Data Set

In this subsection, we present the details of our data set. Our data set is publicly available in the replication package available at https://www.parsai.net/files/research/ICTSSRepPak.zip.

In order to find the common fault patterns related to C++11/14, we looked at the authoritative sources of fault patterns such as those suggested by Scott Meyers in his book titled Effective Modern C++ [15], and C++ Core Guidelines by Bjarne Stroustrup [21]. We also took into account the standard proposal N3853 by Lavavej [14] which points out problems with range-based for loop syntax.

Table 1. Project statistics

Project	Commit	Size (Lines of Code)		Number of commits	Team size
		Production	Test		
i-score	c86cd3d	108K	3.5K	5358	14
C++React	1f6ddb7	11K	2K	417	1
EntityX	6389b1f	9K	1K	296	28
Antonie	59deb0d	9K	0.1K	306	2
Json	a09193e	8K	18K	1973	59
Corrade	ff3b351	6.5K	9.1K	1898	10
termdb	bd0fb4a	783	153	26	2

For the evaluation of the mutation operators, we looked at seven open source projects that use C++11/14 (Table 1). These projects range from a small, several hundred lines of code header-only library, to a full application with over 100,000 lines of code with years of active development:

- i-score is an interactive intermedia sequencer, built in Qt.
- *C++React* is a C++11 reactive programming library, based on signals and event streams.
- *EntityX* is an Entity Component System that uses C++11 features.
- *Antonie* is a processor of DNA reads, developed at the Bertus Beaumontlab of the Bionanoscience Department of Delft University of Technology.
- *Json* is a single-header library for working with Json with modern C++.
- *Corrade* is a C++11/14 utility library, including several container classes, a signal-slot connection library, a unit test framework, a plugin management library and a collection of other small utilities.
- *termdb* is a small C++11 library for parsing command-line arguments.

4 Results

In this section, we present the results of our research. For each mutation operator, first we give its definition, then we discuss the motivation behind it to answer RQ1, and finally we provide our analysis of the data set to answer RQ2.

4.1 For

The range-based "for" reference removal (FOR) operator finds instances of range-based for loops of the form `for (T& elem : range)` or `for (T&& elem : range)`, where T is either `auto` or a concrete type, and removes the reference qualifier from the range declaration. Table 2 shows the results for this mutation operator.

Code Excerpt 1.1. Original For

```
for(auto& elem : range) { ... }
```

Code Excerpt 1.2. Mutated For

```
for(auto elem : range) { ... }
```

Motivation (RQ1). FOR operator is based on the possibility of confusion over the default value semantics of the new range-based for loop, whereas previous methods of looping over containers resulted in reference semantics. This was noted previously by Stephan Lavavej [14]. In his standard proposal, he lists three problems with the most idiomatic-looking range-based for loop, `for (auto elem : range)`, namely:

- It might not compile - for example, `unique_ptr`[3] elements are not copyable. This is problematic both for users who won't understand the resulting compiler errors, and for users writing generic code that'll happily compile until someone instantiates it for movable-only elements.
- It might misbehave at runtime - for example, `elem = val;` will modify the copy, but not the original element in the range. Additionally, `&elem` will be invalidated after each iteration.
- It might be inefficient - for example, unnecessarily copying `std::string`.

From a mutation testing perspective, the second reason is the main motivation to create a mutation operator. In the case of a range-based for loop that modifies the elements of a container in-place, the correct and generic way to write it is `for (auto&& elem : range)`. For all cases except for proxy objects and move-only ranges, `for (auto& elem : range)` works as well.

This operator is only a minor syntactic change that is easily overlooked even in code review if such fault pattern is not actively looked for. Surviving mutants of this type can pinpoint the loops whose side effects on container elements are not tested.

Analysis (RQ2). *Invalid Mutants:* The invalid mutants are comprised of two groups. The majority of the invalid loops were over containers of move-only types. Of the invalid mutants in *i-score*, 33 were containers of pointers to virtual interface classes with custom dereferencing iterators, making the mutant try to

[3] http://en.cppreference.com/w/cpp/memory/unique_ptr.

Table 2. Results of FOR operator

Project	Total	Invalid	Equivalent	Easily detectable	Score
i-score	251	101	115	110	87.5%
Corrade	24	1	13	13	100%
Json	1	0	0	0	100%
EntityX	2	0	2	2	N/A
termdb	0	0	0	0	N/A
C++React	8	0	6	6	100%
Antonie	39	10	18	18	100%

instantiate a non-instantiable type. Both of these cases can be easily checked when generating the mutants.

Equivalent Mutants: In the majority of equivalent cases, the body of the loop did not mutate the referenced element in the container, thus making it equivalent to a loop with an added `const` qualifier. This is relatively easy to verify automatically, hence such mutants are listed as detectable. Only a handful of equivalent cases were loops that did mutate the elements of the container, but the container never gets used after the loop finishes. This would require more complicated static analysis.

4.2 LMB

The lambda reference capture (LMB) operator changes a default *value* capture to a default *reference* capture. Table 3 shows the results for this mutation operator.

Code Excerpt 1.3. Original Lambda

```
[=](int x) { return x + a; };
```

Code Excerpt 1.4. Mutated Lambda

```
[&](int x) { return x + a; };
```

Motivation (RQ1). This operator is based on the warnings on default capture modes in Core Guideline F53 and Meyers' 31st item [15,21]. This mutation operator results in code that leads to undefined behavior if the lambda is executed in a non-local context, because the references to local variables are not valid. This can happen when the lambda is pushed up the call stack or sent to a different thread for asynchronous execution.

Just like the FOR operator, this operator is only a minor syntactical change that can easily be overlooked, and results in faults that are not necessarily easy to detect; thus it is worth testing for its absence. Mutants created by this operator are not easy to detect either, because they invoke undefined behavior which is highly dependent on compiler optimization levels and runtime circumstances.

Table 3. Results of LMB operator

Project	Total	Invalid	Equivalent	Easily detectable	Score
i-score	189	0	113	101	86.3%
Corrade	0	0	0	0	N/A
Json	0	0	0	0	N/A
EntityX	0	0	0	0	N/A
termdb	0	0	0	0	N/A
C++React	1	0	0	0	100%
Antonie	0	0	0	0	N/A

Analysis (RQ2). *Invalid Mutants:* We did not witness any invalid mutants generated by this operator in our data set.

Equivalent Mutants: All undetectable equivalent mutations were ones where the lambda gets passed into a function that executes it within its own scope. While it is theoretically possible to detect them, we classify them as undetectable because it would require complicated non-local reasoning. The other equivalent mutants are detectable by taking into account what the capture list actually captures. For example, in Code Excerpt 1.5, the minimal capture list is empty, whereas in Code Excerpt 1.6 the minimal capture list is `[a]` and in Code Excerpt 1.7 the minimal capture list is `[this]`. In the first and third examples, replacing the default value-capture with reference-capture changes nothing about the capture list. In i-score, these made up the majority of equivalent cases, hence the high percentage of detectable equivalent mutants.

Code Excerpt 1.5. Empty Capture

```
[=](int x) {return x < 1;};
```

Code Excerpt 1.6. Local Capture

```
int a; [=](int x) {return x < a;};
```

Code Excerpt 1.7. 'this' Capture

```
struct Foo {
  int a;
  auto getFilter() {
    return [=](int x) {return x < a;};
  }
};
```

4.3 FWD

The forced `rvalue` forwarding (FWD) operator replaces `std::forward` instances with `std::move` to force moving from forwarded arguments. Table 4 shows the results for this mutation operator.

Code Excerpt 1.8. Original Forwarding

```
template<class T>
void wrapper(T&& arg)
{
    foo(std::forward<T>(arg));
}
```

Code Excerpt 1.9. Mutated Forwarding

```
template<class T>
void wrapper(T&& arg)
{
    foo(std::move(arg));
}
```

Motivation (RQ1). There are often two possible errors in relation to forwarding semantics (which Meyers warns about in his items 24 and 25 [15]): forgetting to use `std::forward` (and thus passing both `lvalues` and `rvalues` on as `lvalues`) or moving instead of forwarding (and thus passing `lvalues` on as `rvalues` to be moved from).

As an example, the following function constructs an object of type T using uniform initialization by forwarding the variadic list of arguments using perfect forwarding:

```
template<typename T, typename... Args>
T construct(Args&&... args) {
  return T{std::forward<Args>(args)...};
}
```

We then use the following type, chosen because `std::string` has a destructive move constructor and `std::unique_ptr` is a move-only type:

```
struct Widget
{
  std::string text;
  std::unique_ptr<int> value;
};
```

Then the following code constructs two Widgets with the same text and different values:

```
std::string text{64,'a'}; //Long enough to disable SSO
auto w1 = construct<Widget>(text,std::make_unique<int>(0));
auto w2 = construct<Widget>(text,std::make_unique<int>(1));
```

Both calls result in `Args` being `[std::string&,std::unique_ptr<int>&&]`, which makes `std::forward` correctly forward the first argument as `lvalue` and the second as `rvalue`. Forgetting to use `std::forward` results in both arguments being forwarded as `lvalues`, which fails to compile since `std::unique_ptr` is a move-only type. When forgetting to forward, code will always either compile and default to copying the types, or fail to compile because a move-only type is used. Since for all types, the only visible effect of doing a copy instead of a move is a performance degradation, this would not be a useful operator for testing purposes.

Table 4. Results of FWD operator

Project	Total	Invalid	Equivalent	Easily detectable	Score
i-score	71	13	18	9	81.6%
Corrade	5	0	0	0	100%
Json	14	0	14	6	0%
EntityX	7	0	1	1	100%
termdb	0	0	0	0	N/A
C++React	160	0	17	15	98.6%
Antonie	0	0	0	0	N/A

Replacing the `std::forward` with `std::move`, however, does has the potential to change program behavior. With `construct` mutated as in the code sample above, the string `text` will be moved from in the first call, and the second call results in unspecified behavior. In most standard library implementations, `w2` will end up with an empty text. Meyers argues that it is easy to confuse `rvalue` and forwarding references because of their identical syntax, making this a likely fault for developers to make.

A large part of these mutants can be targeted by using forwarding on a non-const `lvalue` argument, since it cannot bind to an `rvalue` reference. Another way of testing these is to use a type with a destructive move, and test the state of the original object after passing it into the function as an `lvalue`.

Analysis (RQ2). *Invalid Mutants:* The invalid mutants were comprised of two groups: fixed template argument and non-const `lvalue` reference callee arguments. The first group forwards to another template function while explicitly stating the template argument as seen in Code Excerpt 1.10. This causes the code to not compile when called with a non-const `lvalue`. If it is called with const `lvalues` or `rvalue` references it will have the same runtime behavior as the original.

Code Excerpt 1.10. Fixed Template Argument Forwarding

```
template<typename T>
void foo(T&&);

template<typename T>
void bar(T&& t) {
  foo<T>(std::forward<T>(t));
}
```

The second group forwards into a function with fixed arguments, at least one of which is a non-const `lvalue` reference, as seen in Code Excerpt 1.11 which defines a function that calls another with a prepended integer argument. Because

the second argument is a non-const `lvalue` reference, applying the operator here results in an invalid mutant because it cannot bind to an `rvalue` reference.

Code Excerpt 1.11. Forwarding into Non-Const Lvalue Reference

```
void foo(int,int&,int);

template<typename... Args>
void bar(Args&&... args) {
  foo(1,std::forward<Args>(args)...);
}
```

Equivalent Mutants: There are three categories of equivalence for this operator. The first is where std::forward gets used within a `decltype` or `noexcept` context, where the operator either changes nothing, or makes the code fail to compile. This is why we classify these as detectable equivalent mutants. The second case is where the forwarded argument never gets stored, which makes irrelevant the difference between `std::forward`, `std::move`, and passing by reference. The third and final category is where the callees are guaranteed to not take `rvalue` references or value parameters of movable types. Of these three categories, the first is easily detectable by filtering out mutants within a `decltype` or `noexcept` expression. The second would require sophisticated flow analysis which is why we listed them as not easily-detectable. The last category can be detected if it is feasible to find all possible callees and see whether they take any `rvalue` references or value parameters of movable types. This is only feasible for mutants calling functions that cannot be overloaded by external code, since it is otherwise theoretically possible to introduce a new overload of the called function that takes a parameter of a type with a destructive move, making the mutant non-equivalent. The mutants for which this analysis is possible are listed as detectable in our analysis.

4.4 INI

The initializer list constructor (INI) operator checks constructor calls of types with an initializer list constructor and changes to/from uniform initialization in order to provoke calling a different constructor. Table 5 shows the results for this mutation operator.

Code Excerpt 1.12. Original Initializer

```
std::vector<int> v(3,2);
```

Code Excerpt 1.13. Mutated Initializer

```
std::vector<int> v{3,2};
```

Motivation (RQ1). While initializer list constructors are helpful in defining container contents, they are possible sources of faults as well. For example, when using uniform initialization one needs to pay attention to the correct syntax,

Table 5. Results of INI operator

Project	Total	Invalid	Equivalent	Easily detectable	Score
i-score	1	0	0	0	100%
Corrade	0	0	0	0	N/A
Json	0	0	0	0	N/A
EntityX	0	0	0	0	N/A
termdb	1	0	0	0	100%
C++React	0	0	0	0	N/A
Antonie	18	0	0	0	100%

since using {} instead of () by mistake changes the semantics of the expression drastically. A prominent example of this problem is `std::vector` of integer types, which Meyers points out in his 7th item [15]. The non-mutated version in Code Excerpt 1.12 defines a vector of three elements with value 2, whereas the mutated vector in Code Excerpt 1.13 has only two elements: 3 and 2.

Analysis (RQ2). *Invalid Mutants:* This operator has no way of creating invalid mutants by design, because it checks whether or not a different constructor is called when it is applied. This includes checking for narrowing conversions; e.g. when trying to mutate `std::vector<char>(10,'a');`.

Code Excerpt 1.14. Equivalence Cases for INI

```
struct Default1 {
int foo = 1;
Default() = default;
Default(int f) : foo(f) {};
};

std::vector<Default1> v1(1); //v1{1}
std::vector<int> v2(2,2); //v2{2,2}
```

Equivalent Mutants: There are only a few corner cases for `std::vector` where this operator results in equivalence (e.g. Code Excerpt 1.14).

In both of these cases, the mutated initializer results in the same vector as the original. Given the number of times this pattern was observed in our data set (20 instances in all projects), it is unlikely that such equivalent mutants are found in any significant number.

4.5 Discussion

We have aggregated the number of all generated mutants per kind for each mutation operator in Fig. 1. The FOR operator generates the highest number

of mutants, most of which are either invalid or easily detectable equivalent. Hence, it is possible to filter most of these mutants easily. This is why this mutation operator is promising. The most promising mutation operator is INI, which generated no invalid or equivalent mutants in our data set. However, the low number of mutants generated by this mutation operator means that it might not be applicable in every case. FWD is the operator that generates the most valid, non-equivalent mutants along with a low number of equivalent and invalid mutants, while LMB generates no invalid mutants at all but has a slightly higher ratio of equivalent mutants that are hard to detect.

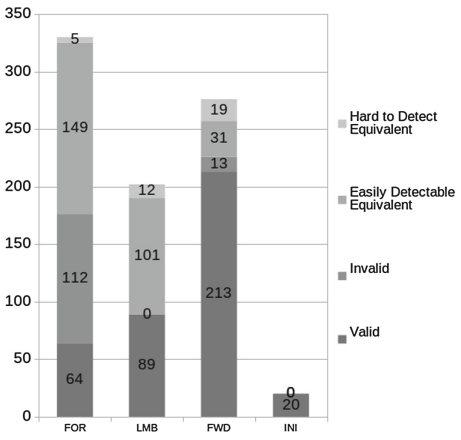

Fig. 1. Generated mutants

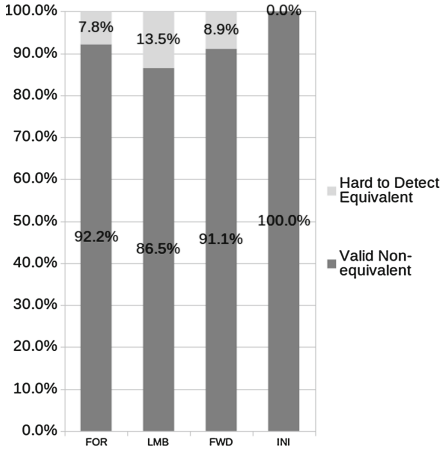

Fig. 2. Mutation operator scores

Figure 2 shows the mutation operator score for each mutation operator. It is clear that all mutation operators are within reasonable boundaries regarding the percentage of generated hard to detect equivalent mutants when compared to other C++ mutation operators (e.g. Delgado-Perez et al. [4]). Overall, we found that these mutation operators have a high mutation operator score, with all of them generating very few equivalent mutants (13.5% or less of the total number of mutants).

One of the noticeable trends among these mutation operators is their tendency to generate lots of mutants in a single project, and few in others. For example, INI generated 18 mutants in Antonie, and 2 in all other projects, while LMB generated 189 mutants in i-score and only 1 in others. Other than the size of the projects, we found that the adoption of the new syntax has not been uniform in all of the projects, i.e. some projects make use of mostly a single new syntactic feature and not all of them.

5 Conclusions and Future Work

In this study, we created a set of mutation operators that target the common faults introduced by C++11/14 syntactic features. We collected advice about the new C++11/14 syntax from authoritative sources, and created four new statement-level mutation operators (FOR, LMB, FWD, and INI). For each mutation operator, we discussed the motivation behind its creation and the type of faults they generate. We used Mutation Operator Score as a way to measure the effectiveness of each mutation operator. For this, we selected 7 real-life C++11/14 projects, and counted the number of valid, invalid, easily detectable and hard to detect equivalent mutants generated by each mutation operator for each project. Our results show that all of the introduced mutation operators generate at most 13.5% hard to detect equivalent mutants. The high operator scores indicate that these mutation operators are a useful addition to the mutation operators suggested previously in literature.

Several aspects of this study can be researched further. In particular, the use of our proposed mutation operators alongside traditional and class mutation operators may result in finding multiple redundancies among these mutation operators. In addition, a comparative study similar to Delgado-Perez et al. [5] between these mutation operator sets would provide more insight into the usefulness of each set of operators depending on the context.

Acknowledgments. This work is sponsored by (a) the ITEA3 ReVaMP^2 Project (number 15010), sponsored by VLAIO—Flanders Innovation Sponsoring Agency; (b) Flanders Make vzw, the strategic research centre for the manufacturing industry.

References

1. Baker, R., Habli, I.: An empirical evaluation of mutation testing for improving the test quality of safety-critical software. IEEE Trans. Softw. Eng. **39**(6), 787–805 (2013). https://doi.org/10.1109/TSE.2012.56
2. Bradbury, J.S., Cordy, J.R., Dingel, J.: Mutation operators for concurrent java (J2SE 5.0). In: Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006), p. 11, November 2006. https://doi.org/10.1109/MUTATION.2006.10
3. Chekam, T.T., Papadakis, M., Traon, Y.L., Harman, M.: An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp. 597–608, May 2017. https://doi.org/10.1109/ICSE.2017.61
4. Delgado-Pérez, P., Medina-Bulo, I., Domínguez-Jiménez, J.J., García-Domínguez, A., Palomo-Lozano, F.: Class mutation operators for C++ object-oriented systems. Ann. Telecommun. - annales des télécommunications **70**(3), 137–148 (2015). https://doi.org/10.1007/s12243-014-0445-4
5. Delgado-Pérez, P., Medina-Bulo, I., Palomo-Lozano, F., García-Domínguez, A., Domínguez-Jiménez, J.J.: Assessment of class mutation operators for C++ with the MuCPP mutation system. Inf. Softw. Technol. **81**, 169–184 (2017). https://doi.org/10.1016/j.infsof.2016.07.002

6. Deng, L., Offutt, J., Ammann, P., Mirzaei, N.: Mutation operators for testing android apps. Inf. Softw. Technol. **81**, 154–168 (2017). https://doi.org/10.1016/j.infsof.2016.04.012
7. Ferrari, F.C., Maldonado, J.C., Rashid, A.: Mutation testing for Aspect-Oriented programs. In: 2008 1st International Conference on Software Testing, Verification, and Validation, pp. 52–61, April 2008. https://doi.org/10.1109/ICST.2008.37
8. Irwin, W., Churcher, N.: A generated parser of C++. NZ J. Comput. **8**(3), 26–37 (2001)
9. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. **37**(5), 649–678 (2011). https://doi.org/10.1109/TSE.2010.62
10. Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R., Fraser, G.: Are mutants a valid substitute for real faults in software testing? In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, pp. 654–665. ACM, New York (2014). https://doi.org/10.1145/2635868.2635929
11. Kim, S., Clark, J.A., McDermid, J.A.: Class mutation: mutation testing for object-oriented programs. In: Proceedings of Net Object Days 2000, pp. 9–12 (2000). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.16.6116&rank=1
12. King, K.N., Offutt, A.J.: A Fortran language system for mutation-based software testing. Softw.: Practice Exp. **21**(7), 685–718 (1991). https://doi.org/10.1002/spe.4380210704
13. Kusano, M., Wang, C.: CCmutator: a mutation generator for concurrency constructs in multithreaded C/C++ applications. In: Proceedings of 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, pp. 722–725 (2013). https://doi.org/10.1109/ASE.2013.6693142
14. Lavavej, S.T.: ISO/IEC JTC1/SC22/WG21 N3853: Range-Based For-Loops: The Next Generation (2014). http://open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3853.htm
15. Meyers, S.: Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14, 1st edn. O'Reilly Media Inc., Sebastopol (2014)
16. Offutt, A.J., Pan, J.: Automatically detecting equivalent mutants and infeasible paths. Softw. Test. Verif. Reliab. **7**(3), 165–192 (1997). https://doi.org/10.1002/(sici)1099-1689(199709)7:3⟨165::aid-stvr143⟩3.0.co;2-u
17. Offutt, A.J., Voas, J.M.: Subsumption of condition coverage techniques by mutation testing. Technical report, George Mason University (1996)
18. Oliveira, R.A.P., Alégroth, E., Gao, Z., Memon, A.: Definition and evaluation of mutation operators for GUI-level mutation analysis. In: 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 1–10, April 2015. https://doi.org/10.1109/ICSTW.2015.7107457
19. Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Traon, Y.L., Harman, M.: Mutation testing advances: an analysis and survey. In: Advances in Computers (2018). https://doi.org/10.1016/bs.adcom.2018.03.015
20. Stroustrup, B.: Programming: Principles and Practice Using C++, 2nd edn. Addison-Wesley Professional, Boston (2014)
21. Stroustrup, B.: C++ Core Guidelines (2017). http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines
22. Sutter, H., Stroustrup, B., Reis, G.D.: ISO/IEC JTC1/SC22/WG21 N4262: Wording for Forwarding References (2014). http://open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4262.pdf