Testing a PL/I Compiler Using Precomputation-based Program Generation

Jesse Postema
Universiteit Van Amsterdam
Amsterdam, The Netherlands
mail@jessepostema.nl

Johan Fabry
Raincode Labs
Brussels, Belgium
johan@raincode.com

Yannick Barthol
Raincode Labs
Brussels, Belgium
yannick@raincode.com

Ana Oprescu
Universiteit Van Amsterdam
Amsterdam, The Netherlands
A.M.Oprescu@uva.nl

***Abstract*—In automated compiler testing, the focus typically lies in uncovering bugs caused by optimisations performed by the compiler. However, there is a class of compilers where little to no optimisations are performed: those for migration of legacy software. Therefore, it is not clear to what extent such legacy compilers would benefit from automated compiler testing. We investigated this in the context of the Raincode legacy compiler for PL/I, an industrial compiler targeting the .NET platform. We designed and implemented a framework for automated PL/I compiler testing through precomputation-based program generation and ran it on two versions of the Raincode PL/I compiler: an older with known bugs and the latest release. On the older version, our framework generated around 127.000 programs and found five bugs, two of which were previously unknown to us. For the latest compiler release, after 180 hours of tests and more than 718.000 generated programs, the framework did not reveal any bugs.**

1. Introduction

Compilers generally have a few features that make them interesting subjects for automated testing approaches: they are large, complex and yet expected to be correct and depended on by users. The former two mean they are difficult to test well, and the latter mean that it is of paramount importance that they are tested well. Prior research into the domain of automated compiler testing [1] focuses itself primarily on testing modern, optimized compilers, with much of this work targeting optimization phases of various C compilers [2], [3], [4], [5].

We focus instead on adapting and applying some of these techniques on a so called *legacy compiler* [6]. Legacy compilers differ from modern compilers in two key aspects. First is the language they cover, and second is the degree of optimisations. Their goal is not in producing the most efficient result, but rather making the language available on a more modern platform than it was originally designed for. This means covering the entire language syntax, with behaviour identical to that on the old platform remains the most important objective. Consequently, no significant effort is placed in developing optimizations.

While application of automated compiler testing techniques on legacy compilers is a relatively unexplored research area, these compilers can benefit greatly from automated testing, arguably to an extent even more so than modern compilers. This comes from the few opportunities they generally have for manual testing and production testing, due to the small number of domain experts and publicly available use cases, respectively.

2. Industry problem

Raincode©[1] is an independent compiler company, offering among others PL/I and COBOL legacy compilers for Microsoft's .NET framework. Automated compiler testing techniques present an interesting opportunity to expand their overall testing approach, which currently relies on a manual testing framework as well as production experience.

Automatically testing compilers is not a new idea. One effort to test a PL/I compiler [7] is at the time of writing just over half a century old. However, if the goal is to test not only if something compiles, but also if it compiles correctly (i.e. produces the right output), some form of oracle must be used [8]. Since true oracles that can tell for any given program what its output should be are not available in practice, they can be substituted in a couple of ways.

One approach is called Random Differential Testing (RDT), which comes down to compiling the program using multiple compilers, and assuming that the majority of the compilers produce the right result. CSmith [9] is perhaps the most well known example of this, having uncovered many bugs in the GCC and LLVM C compilers. However, RDT requires multiple compilers to be available, and becomes more reliable the more compilers are included. This makes RDT less feasible in an environment where not many different compilers are available. Another approach, Equivalence Modulo Input (EMI), is based on equivalent program mutation [2], [3], [10]. In EMI, an existing random program (sourced using a generator such as CSmith) is mutated so that its output stays the same under a predetermined input. Like with RDT, all mutants are compiled and run. If one of the mutants produces a different output, a bug is uncovered.

In this work we applied the *precomputation*-based program generation [4] and Equivalence Modulo Input [2] techniques on the *Raincode*© PL/I compiler. The main

1. http://www.raincode.com

challenge of our work is twofold: First, adapting automated compiler testing techniques to the PL/I language. Most of these techniques have been designed for testing C compilers, with earlier endeavours outside of the C language [11], [12], [13], [14] having varying levels of success. PL/I is a language with a very large spectrum of constructs available. It contains many features, some of which are unique to the language, and therefore lack any easily translatable generation or mutation strategies. This makes PL/I a good candidate to test the general applicability of automated compiler testing techniques in a legacy context. Second, targeting different kinds of bugs. Automated compiler testing approaches, and in particular mutative approaches are focused on uncovering bugs in the optimization phase(s) of a compiler [1]. Legacy compilers, on the other hand, categorically lack these optimization phases. It is interesting then, to observe how well these techniques will hold up in this context, for which they were not originally designed to be employed.

The goal of our research is exploring the adaptation of automated compiler testing to legacy programming languages. On a practical level, we design and implement a framework consisting of a program generator and an equivalence mutator for automated testing of a PL/I compiler. We define the following research questions:

RQ1 How can we adapt optimization-focused automated compiler testing techniques to legacy compilers?

RQ2 How can we design a program generator for equivalence mutation testing of a PL/I compiler?

RQ3 How can we design an equivalence mutator for a PL/I compiler?

We validate and evaluate our framework by running experiments on two separate versions of the *Raincode*© PL/I compiler: an older version of the compiler, with a set of known bugs, as well as the most current version.

3. Background

3.1. The PL/I Language and Subset/G PL/I

PL/I is a procedural programming language originally designed for IBM's S/360 system and accompanying operating system, OS/360. Its main design goal was to create a programming language that could satisfy the requirements of all three distinct categories of user groups that were present at the time: scientific, commercial and special-purpose. PL/I was designed by committee, a work that started in October 1963. The first document describing the language was presented in March 1964 [16].

As Dijkstra mentions, PL/I is of a "frightening size and complexity" [17], which arguably makes it both hard to implement as well as to teach [18]. To this end, the PL/I General Purpose Subset (Subset/G PL/I) [15] was designed during the 1970s, and released as a standard in 1981. Subset/G PL/I aims to preserve the most useful properties of PL/I while getting rid of rarely used or inefficient features, as well as things that were by the 1970s known as arguably bad language design. Figure 1 shows the statement types available in Subset/G PL/I.

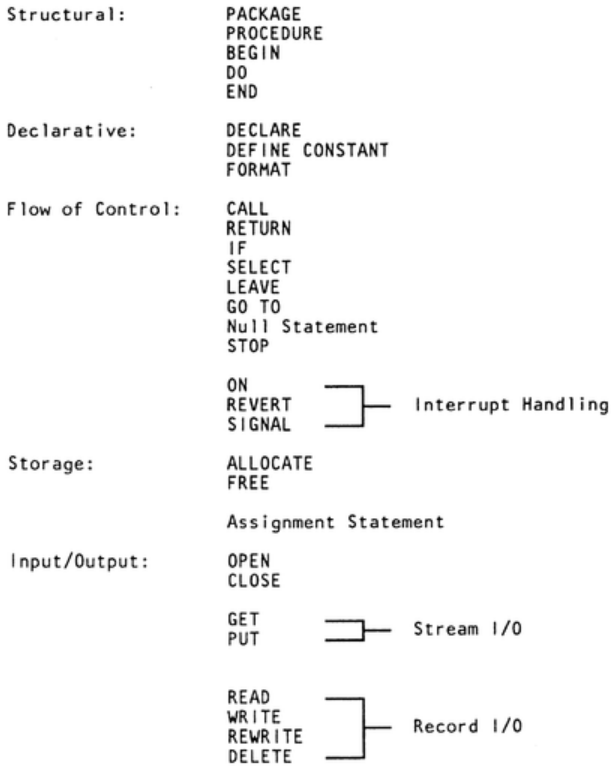

Figure 1. All statement types available in Subset/G PL/I, from [15]

A complete overview of PL/I or Subset/G PL/I is out of the scope of this text. Instead we will highlight two of the language features that are relevant to our discussion.

3.1.1. DECLARE. Variable declarations are done through the DECLARE keyword (abbreviation: DCL). This is followed by a name for the variable and any set of attributes. Multiple variables can be declared using a single declare statement, by putting them in a comma-separated list, as shown in Listing 1.

```
1 /* vars A and CH with values: 4 and default */
2 DECLARE A BINARY INIT(4), CH CHAR;
3 /* var B with value: [5, 0, 0, 0] */
4 DECLARE B DIMENSION(4) BINARY INIT(5);
5 /* var C with value: [5, 5, 0, 0] */
6 DECLARE C DIMENSION(4) BINARY INIT((2)5);
7 /* var D with value: [5, 6, 5, 6] */
8 DECLARE D DIMENSION(4) BINARY INIT((3)(5, 6));
```

Listing 1. Declare statement examples

Three notable attributes of DECLARE are:

type: Arguably the most important attribute is the one that declares the type of the variable. Apart from the usual types of binary, character and floats, PL/I also supports fixed-precision floats and complex numbers.

DIMENSION: An *array* declaration is performed through the *dimension* attribute. This attribute is denoted by the (optional!) DIMENSION keyword, followed by one or more

bounds enclosed in brackets and separated by a comma. Bounds consists of at least an integer value upper bound, optionally preceded by a lower bound followed by a colon. The default value of lower bound is 1, and it must always be less than or equal to the upper bound.
`INITIAL` value (abbreviation: `INIT`): This attribute allows the variable to be initialized with a value other than the default value of its type, *e.g.*, 0 for binary variables. The value is passed between brackets after the keyword. In case the variable is an array, multiple values can be passed to the initial attribute. Singular values or lists of values (enclosed in brackets, comma-separated) can also be repeated by prefacing them with a *repetition*, a single integer value (higher than zero) enclosed in brackets. Excess values, if any, are dropped.

3.1.2. `DO`. In its most basic form, a *do*-group groups statements into a block, executing this block exactly one time. However, its optional elements make it `PL/I`'s de facto looping mechanism. These include:
reference variable: Similar to modern looping mechanisms, there is an option to initialize a loop variable. This is called the *reference* variable.
`WHILE`: This is a loop termination expression evaluated *before* every iteration of the loop.
`UNTIL`: This is a loop termination expression evaluated *after* every iteration of the loop
`BY`: The expression accompanying this keyword is evaluated *once* before the first iteration of the loop, yielding the increment value for the reference variable, applied after each iteration. The default increment value is 1.
`REPEAT`: The expression accompanying this keyword is evaluated *after every* iteration, and the result is added to the reference variable.
`TO`: The expression accompanying this keyword is evaluated once before the first iteration of the loop and when the reference variable gets updated. The reference variable is then checked to be in the allowed range. If not, the loop is terminated.
`FOREVER`(/`LOOP`): If this keyword is present, the loop will iterate forever, unless a `LEAVE` or `GO TO` statement is encountered, or the program terminates.

A do-group can have a combination of while, until and/or to attributes. If any of their termination conditions are met, the loop is terminated.

3.2. The Raincode `PL/I` Legacy Compiler

The compiler that is the subject of our work is the Raincode `PL/I` legacy compiler [19], from here on simply referred to as 'the compiler'. The compiler fully supports `PL/I` syntax, and compiles it to the `.NET` platform [20].

Unlike the typical new code compilation use case, the purpose of Raincode legacy compilers is primarily to compile *existing* and possibly (very) old code, written for the `IBM` mainframe platform, so that it runs on the modern `.NET` platform [6]. As such, the goal of the compiler lies *not* in achieving more efficiency, but rather in making the `PL/I` language available outside of the old mainframe platform. This goal comes with two specific requirements:

Syntax supported by the `IBM` compiler must be supported by the compiler. This is because even small deviations from the language definition could require many changes in software portfolios, prohibiting porting them to the new platform.

Behaviour of compiled programs should remain the same after porting. This means that even bugs in the original `IBM` compiler need to be reproduced.

The compiler is part of a larger tool set at Raincode that allows for source code analysis and manipulation of various legacy languages, *e.g.*, also including `COBOL`. Part of this toolset is the Raincode engine for `PL/I`: it preprocesses a `PL/I` source file and parses it, yielding a parse tree annotated with semantic information. User-written scripts in the YAFL [21] programming language can then be used to operate on the parse tree for analysis and modification.

3.3. Precomputation-Based Program Generation

Random compiler testing is a technique that aims to test a compiler through a continually automatically generated set of random input programs. For as long as specified, random programs are generated. These programs are then compiled and executed. If during either compilation or execution an error arises, the program is saved for later analysis.

Nagai et al. [4] proposed *precomputation*-based program generation: a form of random program generation that generates programs in such a way that its result or output is known beforehand (*i.e.*, precomputed). This allows the compiler testing flow to detect miscompilations causing program crashes as well as those causing incorrect output. Program generation starts from a trivial seed program that does not print any output, but merely exits successfully. On this seed program a number of transformation rules are consecutively applied to generate the final result.

Introduced by Le et al. [2], Equivalence Modulo Input (EMI) is motivated by the difference between *statically* compiling a program P to work on all inputs, and executing it *dynamically* on a subset of those inputs. Given P and an input set $I \in dom(P)$ there exists some set of programs V such that $\forall Q \in V$ it holds that $\forall i \in I \ P(i) = Q(i)$. The set V is referred to as P's *EMI variants* or *mutants*.

Assuming there exists an oracle that given a program P and some input I can produce (a subset of) P's mutants V, this enables to test a compiler in the following way: let the compiler compile P and all $Q \in V$. Then, execute all compiled programs using I. If any of the programs failed to compile successfully, or failed to produce the same result as P, we have found a compiler bug.

To bring the above theory to life, we need actual implementations of the oracle that produces mutants, which typically happens by mutating an existing program. Lascu *et al.* [22] specified three strategies for developing mutations:
Studying specifications: Investigating language specification documents can lead to very detailed insights in how the language functions and how certain features interact.

Consulting prior work and domain experts: Prior work might be applicable to the current domain with slight modifications. Domain experts' knowledge can provide experience of 'real world' applications of the language and/or compiler, which can serve as inspiration for mutations.
Deriving metrics from the compiler: Obtaining code coverage while testing the compiler with the current set of mutations can show which areas of the compiler remain unexercised. This can inspire new mutations that specifically attempt to target those areas.

4. The Program Generation Framework

We designed and implemented a precomputation-based program generation and Equivalence Modulo Input framework for `PL/I` and we present it in this section.

4.1. What to Generate

As established in Section 3.1, `PL/I` is a large and complicated language. Because of this, it is infeasible to include every part of the language in our program generation and/or mutation processes. Inspired by the strategies presented by Lascu *et al.* (see Section 3.3), we base our set of included language constructs on two pillars: Consulting specification documents and domain experts.

4.1.1. Studying specifications. We studied the official language specifications (in the case of `PL/I`, this is the `IBM` language manual [23]) to decide which constructs to include and which to leave out. However, this document does not include any details on how often a construct is used, or how essential it is to the language. It fulfills the purpose of explaining what *is*, rather than what is *useful*.

This is where `Subset/G PL/I` [15] comes in. `Subset/G PL/I` is designed to "preserve the most useful properties of `PL/I`" [18]. Because of this, we decided to only include language constructs included in `Subset/G PL/I`. However, not every language construct lends itself well for random generation or mutation (for instance, because they require very specific circumstances to be valid). This lead us to leave out the record I/O and interruption handling constructs.

4.1.2. Consulting domain experts. We also consulted the main developer for the `PL/I` compiler from Raincode, who has worked extensively on developing their `PL/I` compiler, for their advice regarding which language constructs to focus on. Their suggestions included the following:
Initialization: This refers to the `INIT` keyword (see Section 3.1.1) used during variable declarations.
Loops: Loops, in the form of do-statements are extensively covered by our program generator (see Section 3.1.2).
Dynamically sized types: These include any variables, particularly arrays and procedure parameters, whose size is determined at runtime.

Mixing types: `PL/I` allows for the definition of compound data types, *e.g.*, structures. This stresses the memory-management part of the compiler, which is quite complex as `PL/I` allows for both bit and byte-level alignment.

We incorporated all the suggestions of the domain expert, except for the last, as it would lead to a significantly larger development effort.

4.2. Framework Design

The first design decision that has to be made with regards to the framework is that of the sourcing of input programs. Simply put, without an existing program, there is nothing to mutate. For some languages, tools like `CSmith` [9] exist that generate random programs, which can be used, potentially after some analysis, as input programs. Unfortunately, such tools are not readily available for `PL/I`. This leaves us with the option to use manually written test cases as our input, or build our own program generator. We chose the latter of these two options as it allows for a greater diversity in input programs.

An advantage to building our own generator compared to using a pre-existing one, is that we have the ability to control both which language primitives are added and which technique we rely on for its generation process. For the latter, we opt for *precomputation*-based program generation. The main reason for this is because next to a generated program, this method produces a *state*, bookkeeping the value of every variable at every point in the program. This information is essential to our approach, as our mutation technique requires knowledge of which code is dead or alive in order to execute mutations on it. Alternatively, we could have opted for a generation technique that did not provide us with this state bookkeeping. For example, Le *et al.* [2] take this approach for their implementation, using an existing profiling tool to extract this information from the generated programs. Similar to the program generator, such a tool is not readily available for `PL/I` and we would have to build one ourselves.

During testing, it might be beneficial to limit or tweak the behaviour of the framework. For example, a specific combination of language constructs could lead to compilation error. After a single program containing this error has been analysed, further programs being generated with the same error would be of little use. So, the tester may choose to configure the framework in such a way that this combination of constructs can no longer appear in the same program, in order to allow the framework to find as many different bugs as possible. We have chosen to have modifications be done directly in the code by the tester, as it arguably allows the most fine grained control of the framework. Of course, a notable drawback to this is that the tester needs to have intimate knowledge of the implementation of the framework.

4.3. The Program Generator

The program generator starts from a trivial seed program, shown in Listing 2, and a randomly selected set of

```
template: PROC (command_line) options(main);
    declare command_line char(100) varying;
    declare table(8) binary;
    declare k binary init(1);
    declare c char;

    DO i = 1 to length(command_line);
        c = substr(command_line, i, 1);
        if c = ' ' then k = k + 1;
        else table(k) = BINARY(table(k) || c);
    END;
END template;
```

Listing 2. Seed program with the declaration of the external variable `table` array highlighted

external variables and produces a random program. This program includes language constructs such as variable declarations, if statements and variable assignments. It is produced together with a state bookkeeping that indicates for every line in the program what the value of every variable in the program is at that point and whether the line will be executed (alive) or not (dead) during program execution.

Our generator is split up into the following components:

External variable generation: A random set of variables is generated that the resulting program will read.

Seed program: This program is shown in Listing 2. It forms the basis for generation and includes the required code to read out the external variables.

Variable declarations: A set of variable declarations are the first things added to our seed program. They are used as input for the following steps.

Variable assignments: The first use of our declared variables. Some of them are assigned a (new) value.

Control statements: These statements 'wrap' themselves around a variable assignment, thereby complicating the program's control flow.

Print statements: Finally, print statements are added that output the valuations of every variable.

We limit our discussion to three key elements of the generator, due to a lack of space.

Variable Declarations: Each declaration exists of a single binary variable or (multi-dimensional) array, possibly also containing an initialization via the `INIT` language construct (See Section 3.1.1). For every declaration, the variable name, its dimensions and values are recorded, adding to the *state* of the program we keep track of. Every line following the declarations is paired with this set of information, denoting the value of every variable at that point in the program. Next to this, we add data about whether this line of code is alive or dead, and which variable, if any, is altered on this line. Note that this strategy is only feasible because we have fine grained control over how our program is generated, ensuring that, for example, only a single variable is ever altered on one line and that we can predetermine ahead of time whether a line of code will be alive or dead.

Variable Assignments: In these, the left-hand side consists of a single variable (including index, if we are dealing with an array) randomly chosen from our list of declared variables. The right-hand side consists of a random integer value. When adding this statement, we create a new *state*, based on the state of the previous line, simply altering the value for the specific variable we are updating. Where our approach differs from earlier work, is that these simple assignments are later not expanded into either multiple lines or into more complicated arithmetic operations. This is a deliberate choice, as our focus is not to target the arithmetic optimization aspect of the compiler.

Control Statements: These take the form of `if` and `do` statements (see Section 3.1.2). Here, we focus on generations of `do` statements.

First, we generate a variable initialization, in the form $i \leftarrow$ some integer value or any of the declared variables.

Second, either a `repeat` or `to/by` keyword(s) is added, chosen randomly, since they are mutually exclusive options. For example, we add a `to` or `by` keyword, with some integer or any of the declared variables as their value. Since a combination of `to` and `by` can cause the body of the statement to no longer be executed (see Section 3.1.2), the framework determines this, and if the code is currently alive, the state is updated.

Third, optionally a `while` keyword is added. A condition is produced with i as its left hand side, some integer value or any of the declared variables as its right hand side, and an arbitrary comparator. Again, since this can cause the body of the statement to no longer be executed, the framework determines this and updates the state if needed.

Fourth, an `until` keyword is optionally added. However, since the `until` clause only gets evaluated *after* the first iteration, it does not influence liveness, and the framework does not need to check for it.

Fifth and last, the framework generates either a `leave` or `goto` statement with a corresponding label. This statement is appended to the body of the `do` statement, ensuring that the loop is only executed once at most, preventing infinite loops.

4.4. The Program Mutator

The program mutator generates from a program and its state bookkeeping a set of equivalent mutant programs, complete with their own state bookkeeping. Our general approach to creating mutants works as follows: as input we take any program (either the program generated by our

generator, or a previous mutant) and its state bookkeeping. We then randomly choose one of the mutation categories, and one mutation within that category to apply. If this mutation is successful (a mutation might fail because the required entity, such as a dead assignment, might not be available), we output the resulting program and its state bookkeeping. This output is then fed back into the mutator. This loop of feeding the result back into the mutator is performed T times, creating mutants that differ more and more from the original program. Compared to only applying a single mutation step on our source program, we take this approach to create more diverse mutants, as our program generator and mutator utilize different language constructs.

We have three categories of mutations: dead code removal, dead code insertion and live code insertion:

Dead code removal mutations remove a single dead statement from the source program. Programs generated by our generator contain two types of language structures that can be dead: Variable assignments and `DO`-groups, potentially made conditional with an `IF` statement. For each of these cases, we construct a basic mutation technique. For variable assignments, we remove the entire assignment and for `DO`-groups, we remove the entire group, including its contents and surrounding `IF` statement, if present.

Dead code insertion mutations insert a single dead statement into the source program. Dead code insertion can perhaps be considered the most 'free' type of mutation, as it does not rely on certain language structures existing in the code like dead code removal, nor is it constrained by its runtime evaluation like live code insertion.

For our mutation strategies, we focused on the following language constructs: Variable assignments, `IF` statements, `BEGIN`-groups, `SELECT` statements, Procedures and `STOP` statements. A complete discussion of these strategies is outside of the scope of this text. Most importantly, dead variable assignments are added at the beginning of dead procedures, `DO`- or `BEGIN`-groups, and `IF` statements and `BEGIN`-groups surround a dead variable assignment.

Live code insertion mutations insert a single live statement into the source program. As the code that we insert will actually be executed, it affects our program's semantics, requiring extra care. In this category, we implement mutations based on three language constructs: `IF` statements, Procedures and controlled variables. We discuss the first two, since space does not permit us to expand on controlled variables.

Given that we know for every `IF` clause whether it will evaluate to true or false, we can also generate these clauses in such a way that they will be true (or false, if that is the desired result). Hence, for a given variable assignment, we make the following two mutations: either put the assignment behind an `IF` statement with a condition that evaluates to true, or create an `IF` statement with a condition that evaluates to false (with an arbitrary variable assignment) and add the original assignment on the next line, possibly preceded by an `ELSE` statement.

We implemented three different mutations based on procedures, one that takes a procedure, `DO`- or `BEGIN`-group or a variable assignment and two that only take a variable assignment. The first works similarly to the dead procedure inserting mutation: it wraps the body in `PROCEDURE` and `END` keywords accompanied by the procedure name. However, in order to make sure that the code inside still gets executed, we append a line containing a `CALL` statement (again, accompanied by the procedure name). The other two mutations utilize procedures to calculate the right hand side of an assignment, instead of the originally present value.

4.5. The Test Runner

The test runner ties the various parts of the framework together. It manages compilation and execution of all generated programs and reports any errors that may have come up in the process. The runner is comprised of four elements: the parser, the compiler, the executor, and finally the comparator.

The input of the parser is the state bookkeeping of the original, non-mutated program. To turn this bookkeeping into the expected results of our programs, it takes the final state of bookkeeping, which corresponds to the last line in the program. It then parses this state, saving only the variables and their corresponding values.

The other part of the input is the set of programs, consisting of both the original, non-mutated program as well as all its mutants. Every program in this set is compiled using the Raincode compiler and the resulting `.DLL` files are saved. If during compilation an error occurs, the program and the output are saved for later analysis.

All `.DLL` files produced by the compiler are fed into the executor one by one, together with the external variables produced during program generation as a single string, space-separated. All outputs of these executions are saved. If any runtime errors occur, the program and its output are saved for later analysis.

Finally, all results produced by the programs are read by the comparator and saved in identical format to the expected output that was produced by the parser in the first step. Then, they are all checked for equality with the expected result. If any actual result is not equal to the expected result, the program and its output are saved for later analysis.

Note that if during the compilation, execution or comparison phase an error is detected in one of the programs, further mutants will not be processed. This is done to prevent the same error from being flagged multiple times, increasing manual analysis time.

After passing all the above-mentioned steps, a wrapper produces a new set of external variables, and the framework is executed again.

5. Validation

To validate our work, we first ran the framework on a version of the compiler with known bugs, to confirm that we can indeed find these bugs. We then ran the framework on the latest release of the compiler, in an attempt to uncover as-yet unknown bugs.

5.1. Validation Runs Setup

To conduct our experiments, the following versions, settings and tools are used:

A computer running a `Linux`-based operating system with a six core (twelve thread) CPU running at a max of 4.2GHz and 16GB RAM.

Raincode `PL/I` Compiler version 4.0.260.0 (26/05/2020).

The current timestamp (retrieved though `Python`'s `time` library) is used as the seed for random generation. Every ten minutes, the timestamp and seed are refreshed.

We chose this version of the compiler because of a bug reported on this version concerning incorrect behaviour of the do-group in some circumstances. Since our framework tests the do-group behaviour extensively through the program generator (see Section 4.3), it should be able to detect this bug, thereby validating our approach.

We configured the framework in such a way that it will continue to run until an error (either a compilation error, a runtime error or a miscompilation) is detected in ten different programs. For each of these types of errors, the first error program of its category is saved along with:

Run A number indicating the run in which this category was first encountered.

Seed used to generate this program, from the list of seeds stored with the run.

Generator and mutator call This is the exact command with which the generator and mutator are called, taking as parameters the seed and external variables.

Runner call The command calling the program runner, including the external variables as a parameter.

Error type when compiling and running this program: compilation fault, runtime error, or miscompilation.

Mutant version This indicates which mutant caused the error, or which is the first mutant to have this error (0 is the original program).

After each run, the flagged programs are manually analysed to establish the underlying origin of the error, for brevity we call this the *kind of error*. The framework is then tuned in order to prevent it from generating the same kinds of errors and it is run until it has flagged ten programs as containing an error, or produced 100.000 programs. This process was repeated four times.

5.2. Results of the Validation Runs

Table 1 shows the results of the four runs we conducted.

Run	Iterations	# of programs	# flagged	Kinds of errors
1	10	94	10	2
2	121	862	10	2
3	3.648	26.201	10	1
4	14.572	100.003	0	0

TABLE 1. VALIDATION EXPERIMENT RUNS

Run #1. The first run of our framework was done with the stock configuration of the framework and was very short, producing ten flagged programs in as many iterations (containing 94 programs in total). This means that **100%** of the iterations produced a flagged program. Analysis of these flagged programs resulted in two bug kinds: one related to init statements (see Section 5.2.1) and one related to loops with a zero-valued by expression (see Section 5.2.2).

Run #2. For the second run, we tuned the framework such that init statements were only allowed to have a nesting level of one and no zero-valued by values. As a result, producing ten flagged programs was only reached after 121 iterations (containing 862 programs in total), or an $\sim$8.3% rate of producing an erroneous program per iteration. Two bug kinds were found among the flagged programs: another related to init statements (see Section 5.2.3) and one related to loops (see Section 5.2.4).

Run #3. One modification was done before execution of the third run: the init statement nesting level was limited to zero. This lead to flagging ten programs after 3648 iterations (with a total of 26.201 produced programs), or a $\sim$0.3% rate of producing an erroneous program per iteration. One additional bug kind was uncovered: a runtime error related to the while condition (see Section 5.2.5).

Run #4. To prevent the loop-related bugs (Sections 5.2.4 and 5.2.5) from reappearing, while conditions were disabled from the framework. After 100.003 produced programs in 14.572 iterations, not a single program was flagged and we terminated execution of the run.

Hence, our validation runs encountered five different bug kinds. We discuss these next.

5.2.1. Init statement compilation error. The first kind of bug we found is a compilation error. After manual analysis of the program, we discovered it was triggered by a controlled variable allocation. More specifically, in the accompanying init attribute (see Section 3.1.1), the compiler was unable to handle three-level deep nesting of repetitions of values. For the sake of brevity, we do not include the exact code that caused the bug, as the init statement by itself is already 25 lines long. This bug was unexpected to us, as we were only aware of issues with do-group behavior in this version of the compiler.

5.2.2. Zero-valued by clause miscompilation. The second type of bug we found is a miscompilation, which after manual analysis we found to be related to the do-group behaviour (see Section 3.1.2). This is in line with what we expected to find, given that we selected this specific compiler version on the basis that this behaviour would be incorrect in some way. Listing 3 shows which part of the program caused the miscompilation result. The state at this point in the program is such that the `dwqxmdspjg(-64,4)` variable, used for the by attribute, has the value 0. Our hypothesis is that the compiler incorrectly assumes that any by value must be ≥ 1 when the to value is greater than or equal to the initial value of the reference variable. However,

```
...
plnqglfpnp(5,171) = 49;
IF tgzacmcvdc(120,44) <= 52 THEN DO i = tgzacmcvdc(127,82) TO table(2) By dwqxmdspjg(-64,4) UNTIL (
    i >= 17 );
    plnqglfpnp(19,160) = 69;
    GOTO awvsfeokai;
END;
awvsfeokai:;
...
```

Listing 3. Bug-triggering program #2

```
...
declare apbbudftlf binary init((5)((1)43));
...
DECLARE oarcdmufyo(-87 * -1) controlled binary init((badgprfmnu(2) + 5)(dcswogwwjd(-15),apbbudftlf)
    ,-94,(40 + -30)(badgprfmnu(-5),11,(15 + -10)58,84,93,(-80 + 86)table(7),65),-51,-79,vcytmodcos
    (44),-44,13,(apbbudftlf + -40)pszejzjgki,dcswogwwjd(2));
allocate oarcdmufyo(apbbudftlf);
...
```

Listing 4. Bug-triggering program #3

```
...
IF bzmqilnqgs(76,-30) <= ohvnbvybvk(110) THEN DO i = -63 TO 60 BY bzmqilnqgs(77,-18) + 1;
    IF cxjzieajsp(-87,28) <= 89 THEN DO i = -4 TO msddlbdijy(65) BY bzmqilnqgs(66,-18) + 5 WHILE ( i
        = bzmqilnqgs(58,-13) ) UNTIL ( i <= glqfrozsxl );
        msddlbdijy(60) = 69;
        GOTO scvzmkofql;
    END;
scvzmkofql:;
    GOTO clnpfajoyn;
END;
...
```

Listing 5. Bug-triggering program #4

```
...
vsuwdtcwyp:;
END;
IF 11 >= -9 THEN DO i = -64 REPEAT i * table(3) WHILE ( i >= aalvdrboal );
    urpoxlfvwo(0,146) = 47;
    LEAVE;
END;
...
```

Listing 6. Bug-triggering program #5

the IBM language manual [23] states that any value ≥ 0 is valid in this case. This causes the discrepancy in behaviour.

5.2.3. Init statement miscompilation. After the first run, we limited their maximum level of nesting repetitions for init statements to one, compared to the default value of three. This lead to finding another bug related to init statements, a miscompilation, shown in Listing 4. Our hypothesis is that the repetition of one, nested inside the repetition of five is incorrectly parsed as a value, rather than a repetition, leading to the difference in value for the `apbbudftlf` variable and thereby `oarcdmufyo`'s length during allocation. This bug was unexpected to us, as we were only aware of issues with do-group behavior in this version of the compiler.

5.2.4. While clause miscompilation. The fourth type of bug we uncovered is shown in Listing 5. Like bug #2 (Section 5.2.2), this is a miscompilation related to the do statement behaviour, where we expected to find bugs. Manual analysis revealed that the miscompilation is in the area of the while clause, which fails to prevent the loop from executing and causes `msddlbdijy(60)` to get assigned the value of 69.

5.2.5. While clause runtime error. The fifth and final type of bug also has to do with the while clause of a do-group and is shown in Listing 6. Rather than a miscompilation however, this error causes the program to produce a runtime error. Interestingly, if our initial manual analysis of the

previous bug (Section 5.2.4) had been more successful, we would have disabled generation of while clauses *before* the third run, rather than only disabling them *after*. However, the analysis of *this* program gained us the insight that the while clause is a problem in both programs. As such, we disabled generation of it for our fourth run.

5.3. Confirmation of Validation Results

After analyzing and categorizing all bug-triggering programs, we discussed the results with the main developer for the `PL/I` compiler from Raincode. They were able to confirm to us that the flagged programs indeed revealed bugs in that version of the compiler. They also stated that these bugs were fixed in two separate changes:

do-group reconstruction: The entire section of the compiler that deals with do-groups was rebuilt on 22/06/2020, which is why we selected this version of the compiler. This fixed the bugs in Sections 5.2.2, 5.2.4 and 5.2.5.

init statement grammar fix: Unaware to us, in 09/11/2020 the grammar used for parsing init statements was changed, fixing the bugs in Sections 5.2.1 and 5.2.3. According to the main developer, Raincode was never aware of the bug in Section 5.2.1 and the grammar was changed to fix the bug of Section 5.2.3, incidentally also fixing the bug in Section 5.2.1.

5.4. Testing the Current Production Compiler

As a last experiment, we used the Raincode `PL/I` Compiler version 4.1.184.0, released on 12/05/2021, which was the latest release at that time. We ran our framework on this version continuously for a week. This run did not produce a single bug-triggering program (fortunately or unfortunately, depending on the point of view). The results of this run are detailed in Table 2.

Run	Iterations	# of programs	# flagged	Kinds of errors
1	100.000	718.033	0	0

TABLE 2. PRODUCTION TESTING RUN

This run was performed with the stock configuration of the framework. In total, it took ~ 180 hours to complete, meaning on the above detailed hardware, ~ 4000 programs are generated per hour. Furthermore, each program contains on average ~ 81 lines of code. Compared to the seed program, which contains just 12 lines, this means our generation and mutation strategies insert on average ~ 69 lines of code.

6. Related Work

There is a significant body of work on compiler testing, with Chen *et al.* [1] providing an elaborate overview of the current state of the art in compiler testing.

To the best of our knowledge, the idea of automatically generating random test programs started by Hanford [7] introducing his "syntax machine" over 50 years ago. It contains an abstract grammar for `little PL/I` [24], a subset of the `PL/I` language. The generation technique includes the use of "syntax generators", which are used, for instance, to ensure that variables are declared before use in the generated programs. It was used primarily to test a compiler by simply checking for every generated program whether the compiler could process it successfully (by not crashing). Compared to their efforts, our approach not only covers a larger set of `PL/I` language constructs, but perhaps more importantly, is able to target the later stages of the compiler, compared to only the parser. This is because we are not only able to construct semantically valid programs, but also predict their intended outcome.

`CSmith` [9] is a random test-case generator for `C` compilers. Similarly to the "syntax machine", it works by continually randomly selecting a feasible item from its abstract grammar. It uses dynamic safety mechanisms (for instance integer and type safety checks) to keep it from producing any undefined behaviour or unspecified behaviour. They use a method called *differential testing* [25] to determine the correct output, meaning they compile and run their programs on multiple compilers, and assume that the majority of compilers give the correct result. This, of course, becomes more reliable the more compilers are included. According to Nagai *et al.* [26], `CSmith` is one of the most successful compiler testing systems. Compared to `CSmith`, our approach does not need to rely on differential testing, which is an advantage given the few `PL/I` compilers available for modern systems. Furthermore, due to the difference in programming languages and fine grained control our approach has over the program state, we do not require the dynamic safety checks they implement.

During the evaluation of their `CSmith` tool, Yang *et al.* [9] found that $\sim 28\%$ of the bugs they found in GCC were related to arithmetic optimization. Nagai *et al.* [4] state that arithmetic expressions are machine dependent, and therefore prone to bugs due to the need for retargeting, however Yang *et al.* [9] found that most ($\sim 62\%$) of the bugs they uncovered were in the machine independent middle end of the compiler. Nevertheless, Nagai *et al.* [4], [26] found numerous bugs in GCC and LLVM by focusing on arithmetic operations using their tool, `Orange3`.

A different approach to avoid undefined behaviour is introduced by Nagai *et al.* [4]. Their tool, `Orange3`, focuses on arithmetic optimizations and is able to pre-compute the expected result of running the program, discarding any program that would introduce unexpected behaviour. A side effect of this is that no differential testing is needed to check for miscompilations, as the correct answer is known beforehand. The trade-off is that they included a very small subset of the entire `C` language in their generated programs loops [27] in order to realise this. Later work [26] improved on this approach by preventing undefined behaviour by adding constant values and 'flipping' operators. This work inspired the design of `Orange4` [5], on which we base the design of our program generator.

Livinskii *et al.* [28] introduced `YARPGen`, a generation technique avoids undefined behaviour by using static analysis, similar to `Orange3` [4], but includes a much broader

subset of the C (and C++) language. They opted for static analysis, rather than dynamic (as with CSmith [9]) because they found that dynamic analysis sacrifices expressiveness. Similar to CSmith [9], they use differential testing [25] to determine correct program behaviour. Our approach differs from YARPGen in that we do not have to rely on differential testing, and our use of program mutation.

Mettoc [29] is the first tool to leverage mutative testing to test a compiler, using the equivalence relation to construct multiple equivalent mutants of a given source program. All mutants are then compiled and executed. If one mutant gives a different result (either a crash or miscompilation), then a bug in the compiler is uncovered. This technique avoids the need for a reference compiler such as with differential testing [25]. Le *et al.* [2] alter this idea by no longer requiring two programs to act the same under *all* possible input values, but only under a predetermined input, and executing the programs under that input. This technique is called Equivalence Modulo Input (EMI). In the original implementation, equivalent mutants are formed by removing dead code from a given source program, but later iterations added the ability to *add* dead code [10] and live code [3] to broaden the set of possible mutants even further. They use programs generated by CSmith [9] as the basis for their mutations, which they profile using available tooling. Our approach differs mainly in not relying on (unavailable) external tooling for program generation and profiling, as well as translating EMI mutation strategies to the PL/I programming language.

Instead of starting with a non-trivial source program, Nakamura and Ishiura [5] present Orange4, which starts from a source containing only `return 0;`. Their definition of equivalence does not consider program *state*, but rather program *output*. Their mutations are constructed in such a way that successful program execution equates to not printing any output and exiting. 'Verification' statements (in the form of conditionally printing output) are added to the program to check whether computation succeeds. While the original implementation only allows for declaration, assignment, if and for statements, its range was later extended [30] to include other language primitives such as arrays and function calls. Our approach uses the theory presented in this work as a basis for the program generator. However, we modify it by omitting expression expansion, as we are not focusing on the arithmetic optimizations of the compiler, and removing the verification statements, instead appending print statements that output the entire program state. These modifications allow our approach to then utilize EMI-based program mutation.

7. Conclusion and Future Work

Although compiler correctness is usually assumed, actually confirming their correctness is difficult, in part due to their complexity. Automated testing techniques aim to alleviate this issue by expanding the test set beyond the reach of manual testing. Existing efforts of this kind largely focus their attempts on modern compilers. However, the feasibility of automated compiler testing for *legacy compilers* is a relatively unexplored concept. We investigated this feasibility, specifically the applicability of optimization-focused testing techniques on a PL/I compiler.

Our work contains two major parts: a program generator and a program mutator, which adapt the *precomputation*-based program generation and Equivalence Modulo Input (EMI) testing techniques for application to the legacy context and PL/I programming language, respectively. Unique to our approach is that while we utilize techniques which traditionally focus on uncovering bugs in the optimization phase(s) of a compiler, we purposefully disregard arithmetic expressions and instead focus on covering the most important language syntax. However, we have kept the overall structure of these techniques in place, starting our program generation from a trivial seed program and applying all mutation categories that we are aware of.

We validated the design of our framework by testing its ability to uncover known bugs on an older version of the *Raincode*© PL/I compiler. During these experiments we generated around 127.000 programs and our framework uncovered five bugs. These bugs were triggered both by programs produced by our program generator (two bugs) and by their mutants produced by our program mutator (three bugs). All of the bugs were acknowledged by a developer as being bugs and all had been fixed in more recent releases. We also tested the most recent release of the compiler, generating around 718.000 programs in 180 hours of testing, but have not uncovered any bugs there.

The design of both our program generator and program mutator are suitable for testing a PL/I compiler (**RQ2** and **RQ3**). Our approach, based on optimization-focused automated compiler testing techniques but adapted to focus on including important language syntax rather than arithmetic optimizations, is suitable for testing a legacy PL/I compiler. Because of our 'black box' treatment of the specific compiler we tested, we anticipate that our approach generalizes well to other PL/I compilers. We also demonstrated that while generation and mutation strategies are specific to a programming language, their design can be adapted to other languages and contexts. Combined, this leads us to conclude that our overall approach is suitable for automated testing of various legacy compilers (**RQ1**). This is further supported by the fact that our approach has minimal prerequisites compared to other automated compiler approaches.

We identify five significant future work avenues. First, including more PL/I constructs, such as multiple data types and interrupt handling. Second, speeding up the bug-finding process by prioritizing programs more likely to have bugs. Third is to automatically reduce programs flagged by our framework as bug-inducing to a minimal case. Fourth is to use multiple PL/I compilers in a Differential testing approach. Fifth and last is to research the applicability of other automated compiler testing techniques in a legacy context.

References

[1] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing," *ACM Comput. Surv.*, vol. 53, no. 1, Feb. 2020. [Online]. Available: https://doi.org/10.1145/3363562

[2] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," *SIGPLAN Not.*, vol. 49, no. 6, p. 216–226, Jun. 2014. [Online]. Available: https://doi.org/10.1145/2666356.2594334

[3] C. Sun, V. Le, and Z. Su, "Finding compiler bugs via live code mutation," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 849–863. [Online]. Available: https://doi.org/10.1145/2983990.2984038

[4] E. Nagai, H. Awazu, N. Ishiura, and N. Takeda, "Random testing of c compilers targeting arithmetic optimization," in *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2012)*, 2012, pp. 48–53.

[5] K. Nakamura and N. Ishiura, "Random testing of c compilers based on test program generation by equivalence transformation," in *2016 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, 2016, pp. 676–679.

[6] D. Blasband, "Compilation of legacy languages in the 21st century," in *International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer, 2011, pp. 1–54.

[7] K. V. Hanford, "Automatic generation of test cases," *IBM Systems Journal*, vol. 9, no. 4, pp. 242–257, 1970.

[8] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015.

[9] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 283–294. [Online]. Available: https://doi.org/10.1145/1993498.1993532

[10] V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," *SIGPLAN Not.*, vol. 50, no. 10, p. 386–399, Oct. 2015. [Online]. Available: https://doi.org/10.1145/2858965.2814319

[11] E. Bouman, "Testing cobol compilers using metamorphic relations," Master's thesis, Universiteit van Amsterdam, 2017.

[12] A. Gül, "Testing a hlasm compiler with a mutative fuzzing approach," Master's thesis, Universiteit van Amsterdam, 2019.

[13] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," *SIGPLAN Not.*, vol. 50, no. 6, p. 65–76, Jun. 2015. [Online]. Available: https://doi.org/10.1145/2813885.2737986

[14] A. F. Donaldson and A. Lascu, "Metamorphic testing for (graphics) compilers," in *Proceedings of the 1st International Workshop on Metamorphic Testing*, ser. MET '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 44–47. [Online]. Available: https://doi.org/10.1145/2896971.2896978

[15] "Information technology — Programming languages — PL/I general purpose subset," International Organization for Standardization, Geneva, CH, Standard, Nov. 1992.

[16] G. Radin, *The Early History and Characteristics of PL/I*. New York, NY, USA: Association for Computing Machinery, 1978, p. 551–575. [Online]. Available: https://doi.org/10.1145/800025.1198410

[17] E. W. Dijkstra, "The humble programmer," *Commun. ACM*, vol. 15, no. 10, p. 859–866, Oct. 1972. [Online]. Available: https://doi.org/10.1145/355604.361591

[18] P. W. Abrahams, "Subset/G PL/I and the PL/I Standard," in *Proceedings of the 1983 Annual Conference on Computers: Extending the Human Resource*, ser. ACM '83. New York, NY, USA: Association for Computing Machinery, 1983, p. 130–132. [Online]. Available: https://doi.org/10.1145/800173.809714

[19] "The raincode `PL/I` compiler." [Online]. Available: https://www.raincode.com/technical-landscape/pli/

[20] "`Microsoft .NET` platform." [Online]. Available: https://dotnet.microsoft.com/

[21] "The `YAFL` programming language." [Online]. Available: https://www.phidani.be/yafl/article/index.html

[22] A. Lascu, M. Windsor, A. F. Donaldson, T. Grosser, and J. Wickerson, "Dreaming up metamorphic relations: Experiences from three fuzzer tools," in *Proceedings of the 1st International Workshop on Metamorphic Testing, MET@ ICSE*, 2021.

[23] *Enterprise PL/I for z/OS*, 5th ed., IBM, 555 Bailey Ave. San Jose, CA, 95141-1099 United States of America, 9 2019.

[24] J. J. Donovan and H. F. Ledgard, "A formal system for the specification of the syntax and translation of computer languages," in *Proceedings of the November 14-16, 1967, Fall Joint Computer Conference*, ser. AFIPS '67 (Fall). New York, NY, USA: Association for Computing Machinery, 1967, p. 553–569. [Online]. Available: https://doi.org/10.1145/1465611.1465685

[25] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[26] E. Nagai, A. Hashimoto, and N. Ishiura, "Reinforcing random testing of arithmetic optimization of c compilers by scaling up size and number of expressions," *IPSJ Transactions on System LSI Design Methodology*, vol. 7, pp. 91–100, 2014. [Online]. Available: https://doi.org/10.2197/ipsjtsldm.7.91

[27] K. Nakamura and N. Ishiura, "Introducing loop statements in random testing of c compilers based on expected value calculation," in *Proc. the Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2015)*, 2015, pp. 226–227.

[28] V. Livinskii, D. Babokin, and J. Regehr, "Random testing for c and c++ compilers with yarpgen," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020. [Online]. Available: https://doi.org/10.1145/3428264

[29] Q. Tao, W. Wu, C. Zhao, and W. Shen, "An automatic testing approach for compiler based on metamorphic testing technique," in *2010 Asia Pacific Software Engineering Conference*, 2010, pp. 270–279.

[30] S. Takakura, M. Iwatsuji, and N. Ishiura, "Extending equivalence transformation based program generator for random testing of c compilers," ser. A-TEST 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 9–15. [Online]. Available: https://doi.org/10.1145/3278186.3278188