VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wānanga o te Ūpoko o te Ika a Māui

School of Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

Transpiling Whiley to C++

Max McMurray

Supervisor: David Pearce

Submitted in partial fulfilment of the requirements for
Bachelor of Engineering with Honours.

Abstract

Whiley2Cpp is a plugin for the Whiley programming language that is being developed as part of this project. The plugin translates Whiley code into C++ code. Whiley is a programming language that uses an automated theorem prover to statically check programs for correctness. C++ is an efficient and widely used programming language that has limited tools to ensure the safety of the code. Transpiling to C++ is not as straightforward as a word for word translation. This is because of differences in the languages such as the discrepancies between C++ types and Whiley types.

Contents

Chapter 1

Introduction

C++ is a popular programming language that originated from the C programming language. C++ is still used today due to its high performance and a high number of projects that already use the language. As C++ originated from C, the language provides support for interacting with the underlying hardware. This makes C++ an efficient language [8] as well as making it more difficult to use than higher-level languages.

Unlike C++, Whiley is a programming language that uses an automated theorem prover to statically check programs. This eliminates many errors at compile time [3], [6]. This means programs can be checked on their correctness by the compiler allowing programs to be much more trustworthy and safe. It is a similar language to Dafny and Spec# as the language involves support for preconditions/postconditions and loop/type invariants which are enforced by static checking. Whiley currently has plugin support for translating the verified code into other languages such as JavaScript [4].

Although efforts are underway to add syntax for preconditions/postconditions to C++, verifying the full C++ language is still beyond the state-of-the-art. An approach other than directly verifying the C++ code is to write the code in a language with a verifying compiler and then transpile it into C++. Dafny currently does this with its C++ backend [10]. In this project, we want to prototype a C++ backend for Whiley to help identify any pain points before a full implementation is developed.

Having a C++ backend for Whiley presents a step towards making C++ safer to use and to have more confidence in programs written. This project attempts to provide a way for C++ to be more safe and reliable by creating a plugin for Whiley. The plugin will generate C++ program files and binaries that are generated from verified Whiley code. The generated C++ code will follow the specifications set in the Whiley code and be able to benefit from the verification from Whiley.

There are cases where translating from Whiley to C++ is very simple. For example, while loops are the same in Whiley as in C++. There are also many cases where there is not such a direct translation. For example, the default union type in C++ has differences from the way union types behave in Whiley.

There would be some uses of a full implementation of the prototype. One includes creating sections of a C++ codebase that are written in Whiley. This could allow for the gradual conversion of a C++ codebase into Whiley. These codebases could be existing codebases that are being expanded on or sections that need safety verification.

1.1 Contributions

Contributions made to this project include:

1. Developed and designed a prototype C++ backend for Whiley
2. Evaluated the backend using the existing Whiley test suite
3. Found failing tests and implemented changes to make the prototype more complete
4. Found problem points of a C++ backend for Whiley

Chapter 2

Background

2.1 Whiley

Whiley is a programming language in which the compiler verifies the program using preconditions, post-conditions, data-type invariants and loop invariants [1]. The language has a functional core with an imperative outer shell. It uses syntax that is indented, similar to languages such as Python.

When the Whiley compiler compiles code it generates a file that contains Whiley Intermediate Language (WhIL). WhIL is written in bytecode and is register-based similar to Java bytecode. The WhIL code is what is executed when a Whiley file is compiled and ran.

2.1.1 Specification Features

Method Contracts

Precondition and postconditions are used to add contracts to methods and functions in Whiley [2]. They are defined in the method or function declaration. A precondition defines requirements for the declared parameters to meet to run the function. A postcondition defines restraints of the returned value of the function. The Whiley compiler is able to determine if the specifications are met and will return an error if this is not the case.

Data-type Invariant

Data-type invariants are used to restrict declared types [2]. They are declared when the type is declared. If at any point an instance of the type does not meet the invariant then the

```
function decrement(int x) -> (int y)
// Precondition requiring x to be greater than zero
requires x > 0
// Postcondition requiring the return to be greater than or equal to 0
ensures y >= 0
// Postcondition requiring the return to be less than the input
ensures y > x:
    return x - 1
```

Figure 2.1: Whiley code demonstrating preconditions and postconditions

```
// Data-type invariant requiring all instances of nat to be greater than or equal to 0
type nat is (int n) where n >= 0
```

Figure 2.2: Whiley code demonstrating data-type invariants

```
...
int i = 0
// Loop invariant requiring i is 0 or more on every iteration of the loop
while i < x where i >= 0:
    i = i + 1
...
```

Figure 2.3: Whiley code demonstrating loop invariants

compiler returns an error if it can detect it.

Loop Invariant

Loop invariants are conditions that are maintained in loop iterations [2]. Loop invariants should hold: on entry of the loop, at the end of each loop, and directly after the loop. They are declared at the start of the loop, after the loop condition.

2.2 Related Work

There are no currently directly comparable works to the best of our knowledge, other solutions have been presented towards similar issues. This section discusses a few of them. The works are categorised into two types: translating to and from similar verification languages to Whiley and works that handle translation from C++ into other languages.

2.2.1 Verification language translation

Prusti

Prusti [7] is a plugin for rustc, the Rust compiler, that verifies Rust code by translating it into the Viper verification language. The errors detected by Viper are then sent back to rustc. Viper is an intermediate verification language that uses permission logic to verify. It has similar verification tools to Whiley as it has preconditions, postconditions and loop invariants [5]. The translation provides some points on how to manage features similar to those in Whiley. The issue is that some of the features in Whiley differ to those in Dafny so it cannot be used as a complete guide.

2.2.2 Translation from C++

CRUST

One paper [8] attempts to solve a similar issue of unsafe C++ code by transpiling C++ to Rust using a program called CRUST. Rust is a systems programming language with highly enforced compiler restrictions and is without garbage collection. These features make it safer than C++ while still providing access to the underlying hardware. Being able to transpile C++ into Rust creates efficient code that is much safer.

CRUST uses a Nano-Parser scheme to transpile C++ into Rust. The scheme involves modular nested parsers that can identify the current expression and calls the specific nano-parser that is associated with the current grammar. These parsers are able to call other parsers to create a chain. This method was chosen as it is a novel and efficient way of building compilers.

CRUST can convert new programs and small to medium-sized existing programs into Rust reliably. The ability to change a large C++ project into Rust is would be costly as the code still needs to be debugged once transpiled.

SSCCJ

Another paper [9] compares current options that exist for converting C++ into Java and presents a system that avoids some of the issues that the options have. They try to solve the issue of maintainability that C++ lacks as it is a low-level language. Java is a programming language that has a similar syntax to C++ with much fewer interactions with the underlying hardware. Java is a high-level language making it much easier to read and therefore easier to maintain.

The paper found that the main challenges when it came to translating with C++ are that Java does not implement all of the C++ features. They needed to do this by relying on substitutes of the features in Java that preserved structure and user-defined information.

Chapter 3

Implementation

3.1 Design

Currently, the plugin is able to translate many of the features of Whiley into C++. The Whiley compiler generates an intermediate object structure, called an AST. The AST is the structure of a program that has been converted into an object-based structure. The AST is then converted into a second intermediate object structure. The new structure is where the logic of the translation occurs. This C++ object structure is then passed into a designed file printer which converts the structure into a string. This string is the Whiley code translated as C++ code. The string is then sent to an output such as a file.

3.2 Motivating Example

Figures 3.2 and 3.3 are an example of an input program and the output transpilation that the plugin produces. There are a few features in Whiley that have a simple translation. The implemented ones include variable declaration, methods, functions, if-else blocks, switch statements and invoking. These simple translations are demonstrated in the motivating examples

There are standard library headers imported in every translated C++ file. These are *vector*, *assert.h*, *variant*, *tuple*, *string*, *cstdint*, *functional* and *memory*. The assert header is included to

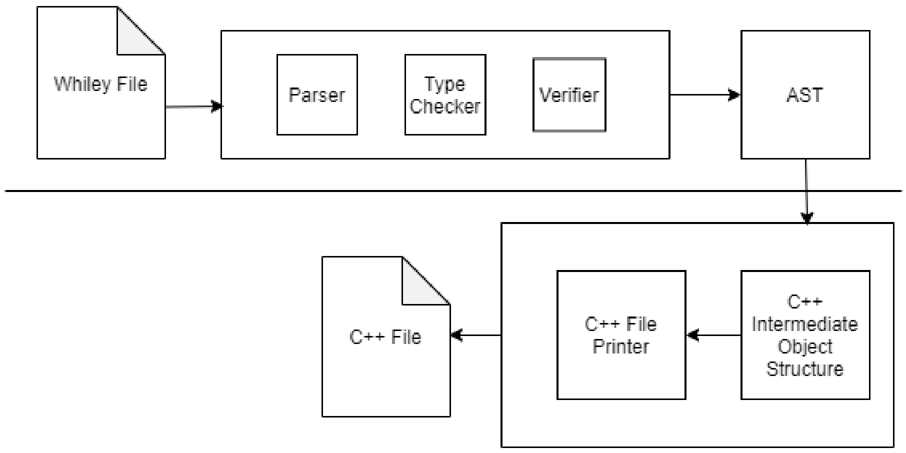

Figure 3.1: Transpiler Design

```
function abs(int x) -> (int r)
ensures r >= 0
ensures (r == x) || (r == -x):
    if x >= 0:
        return x
    else:
        return -x

function abs(int[] arr) -> (int r)
requires |arr| > 0
ensures r >= 0
ensures (r == arr[0]) || (r == -arr[0]):
    int a = arr[0]
    return abs(a)

public export method test() :
    assert(abs([-1, 1]) == 1)
```

Figure 3.2: Whiley Example

```
#include <cstdint>
... // Required Standard Libarary Headers
#include <any>
int example$abs$I$I(int x) {
   int r;
   if(x >= 0)  {
      return x;
   } else  {
      return -x;
   }
}
int example$abs$aI$I(std::vector<int> arr) {
   int r;
   int a = arr[0];
   return example$abs$I$I(a);
}
void example$test() {
   assert(example$abs$aI$I((std::vector<int>{-1, 1})) == 1);
}
int main() {
    example$test();
    return 0;
}
```

Figure 3.3: C++ Example

translate asserts properly in testing as explained in section 4. The other imports are used for translating various features of Whiley into C++. These are explained later.

Every header is included in every file even if they are not in use. This is due to the time limitations and the priorities of the prototype.

3.3 Integers

In Whiley, the integer type has an arbitrary size while C++ has many fixed-size integers. Currently, the translation directly translates Whiley integers into the default C++ integer which has a 4 byte signed width. This will need to be changed if a translation is to not incur any transitional bugs such as overflows and underflows. A potential solution is to use the biggest integer width available for C++, `long long int`. In C++11 `intmax_t` was introduced which indicates an integer of the maximum width which is $2^{63} - 1$ bits or higher. The actual size of the type depends on the library implementation which could lead to some issues.

3.4 Name Mangling

Whiley supports the overloading of methods that have the same name but different parameter types. While C++ also supports overloading, it is possible that two types in Whiley translate to the same type in C++. The solution to this problem is to use a name mangling system. The names of translated methods are mangled when they are translated into C++. This gives distinct names to each overloaded method. For example this C++ code:

```
type nat is (int x) where x >= 0

function f(nat x) -> (int r): ...
function f(int x) -> (int r): ...
```

Translates without mangling into:

```
int f(int x) { ... }
int f(int x) { ... }
```

This does not compile in C++ as there are two methods with the same name and parameters. Translating with mangling solves this issue:

```
int f$I$I(int x) { ... }
int f$nat$I(int x) { ... }
```

3.5 Arrays

Translating arrays from Whiley are straightforward for the most part. Arrays in Whiley are dynamic arrays with a dynamic length while C++ arrays have a fixed length. This means that arrays can not be directly translated. Instead, the vector class from the C++ standard library are used as they are a sequence container that encapsulates arrays with a dynamic length.

Whiley

```
int[] x = [1, 2, 3]
```

C++

```
std::vector<int> x = (std::vector<int>{1, 2, 3});
```

There is one currently known issue when it comes to arrays and that is to do with the empty array case. In Whiley there is support for empty arrays that are declared inline to have their type interpreted, e.g., "`[] == [1, 2]`". Ideally, the empty array would translate to '`(std::vector<int>{})`'. However, the array is interpreted in the Whiley AST as a void array that needs to have its type inferred. In C++, types need to be strictly declared as there is no way to define a void vector.

3.6 Records

In Whiley, records are a type that describes a compound made of one or more fields. These are very similar to C++ structures which were first used as a direct translation.

Whiley

```
type Point is { int x, int y }
```

C++ using structs

```
struct Point {int x; int y;};
```

However, translating equality of structs lead to some issues. In Whiley, the contents of the record are compared when using the equality operator and every record can be compared to any type of record. In C++, each individual structure needs to have its comparison method written for every type that it would be compared to. While this is technically possible it was considered overly cumbersome.

Another potential solution is to translate records into tuples. A C++ tuple is an object that holds different types of elements. Tuples do not have declared keys for the contents but rather work on an index system. The benefit of using tuples is that their equality operator directly compares the contents of the tuple. This would reduce the complexity of translating equality involving Records with identical content types.

C++ using std::tuple

```
using Point = std::tuple<int, int>;
```

The solution that was decided on was tuples as they have useful methods that have been implemented. Also, the implementation of structure equality lead to issues that caused a set of tests to fail that was solved by the tuple implementation.

3.7 Unions

In Whiley, there are union types that allow a variable to contain one of a selection of types. In C++ there are two potential types that unions could be translated to.

The first possible translation is the C++ `union`. The `union` in C++ is a carryover from C. A C++ `union` is similar to a union in Whiley except that it requires named types and a predefined structure.

The second option is the `variant` from C++ standard library. A `variant` is a union that is type-safe that doesn't require named types.

The main difference between a `variant` and a `union` is that `unions` work at a low-level while `variants` have useful tools for checking what type the variant holds and visitation methods.

Due to Whiley translations not requiring the low-level compatibility of unions, variants are used as a translation. Variants have useful methods included in the library that allow for retrieving the correct value and type checking the variable.

Whiley

```
method unions() :
    int|int[] x
    x = 1
    int y = x
    x = [1,2,3]
    int[] z = x
```

C++

```
void unions() {
    std::variant<int,std::vector<int>> x;
    x = 1;
    int y = std::get<int>(x);
    x = (std::vector<int>{1, 2, 3});
    std::vector<int> z = std::get<std::vector<int>>(x);
}
```

In the Whiley AST the types are stored in an array with an order as defined in the file.

3.8 Templates

In Whiley, templates are used to define a variable type for a method or class definition. Whiley templates are used in the same ways as templates in C++ which makes the translation slightly easier.

Whiley

```
type Decorator<T> is { T data }

function id<T>(T x) -> (T y):
    return x
```

C++

```
template <typename T>
using Decorator = std::tuple<T>;

template <typename T>
T id(T x) {
   T y;
   return x;
}
```

3.9 Lambdas

In Whiley, lambdas are references to functions and methods or are used as a variable. C++ has a similar lambda in the standard library which is used as a direct translation for the parameter defined lambdas. References to methods are translated as a reference to the method.

Whiley lambdas capture outer variables. C++ lambdas are non-capturing by default. This can be easily be changed by setting the lambda to capturing variables. This can cause issues if the lambda is being defined as a static variable as there are no variables to capture. This leads to a compiler error in C++. The current translation is for lambdas not to be capturing if it is part of a static variable. This is still an accurate translation as there should not be any variables to capture.

Whiley

```
type Lambda is function(int) -> int
Lambda staticLambda = &(int x -> x + 1)

method getLambda() -> function(int) -> int:
    return &(int x -> x + 1)
```

C++

```
using Lambda = std::function<int(int)>;
Lambda staticLambda = [](int x) {
   return x + 1;
};

std::function<int(int)> getLambda() {
   return [=](int x) {
      return x + 1;
   };
}
```

A lambda can also be used as a type for a parameter or return type. In C++ `function` from the standard library is used as a translation.

3.10 Null

In Whiley, the `null` keyword represents the absence of a value. This has two main uses in Whiley. The first is to assign as a value to variables that need to have no value. This leaves the variable in a similar state to being uninitialised. Using the null variable use can also be used for equality operations.

The second use is as a unit type that can only hold the null value. This is useful in union type definitions to indicate that null checking should be used and can be enforced by the verification. For example `type Nullable is null|int` defines a union type that can be an integer or a null value.

There is no direct translation in C++ that works exactly like `null` in Whiley. This means the translation needed to be split up into multiple chunks. The variable use had two options that were considered. The first being the `NULL` keyword which originated in C as a null pointer macro. When translated over to C++ it is defined as the value `0` in most compilers. The option chosen is the `nullptr` which was introduced in C++11 which represents a null pointer more accurately.

The second use of the `null` keyword is as a type, specifically as part of a union. In C++ there is no type that can only be a null value. Instead, a `void*` type is used as a translation. This is because `void*` holds a pointer with no associated type. This allows for variants to hold `nullptr` and can still be retrieved.

Whiley

```
int|null x = 1
```

C++

```
std::variant<int,void*> x = 1;
```

3.11 References

References in Whiley are simple pointers. In C++ one potential option is to translate the references as C++ pointers. Due to the low-level nature of C++, this can lead to memory management issues. In the standard library, there are smart pointers. Smart pointers are types that can be used in the same way as a pointer but have automatic memory management.

The smart pointer that most closely relates to Whiley pointers is the `shared_ptr`. The shared pointer allows for multiple variables to reference the same object. It uses reference counting to store the number of times the variable has been referenced. Once this count reaches zero then the memory is cleared. This translation is not fully functionally equivalent to references in Whiley as it won't reclaim memory if there are cycles in the object graph. Whiley allows reference cycles. However, this is close enough for the prototype.

Whiley

```
method swap(&int x_ptr, &int y_ptr):
    int tmp = *x_ptr
    *x_ptr = *y_ptr
    *y_ptr = tmp
```

C++

```
void swap(std::shared_ptr<int> x_ptr, std::shared_ptr<int> y_ptr) {
   int tmp = *x_ptr;
   *x_ptr = *y_ptr;
   *y_ptr = tmp;
}
```

Chapter 4

Evaluation

4.1 Testing

The method of checking the implementation is able to translate Whiley code correctly is using automatic testing. Testing is done by using an existing suite of around 700 Whiley tests from the main Whiley repository. The test suite covers many of the features and quirks of the language. Within each Whiley test file, assertions are used to test functionality. Similar assertions are available for C++ in the standard library which is used as a translation.

Methods in the tests are indicated to be run by being named *test*. Each test file is translated into a C++ file and is then compiled using g++. As C++ requires a main method, one is inserted into the generated C++ file so that the test runs correctly. The main method consists of a call to the test method. The C++ file is then compiled with g++. Tests are considered passed if the generated C++ file is able to compile and the compiled binaries run without an assertion being thrown.

4.2 Currently Failing Tests

Around 50% of the test suite is currently failing. This is mainly due to complex interactions of individual features as well as features that are yet to be fully implemented. Below are some features that cause tests to fail

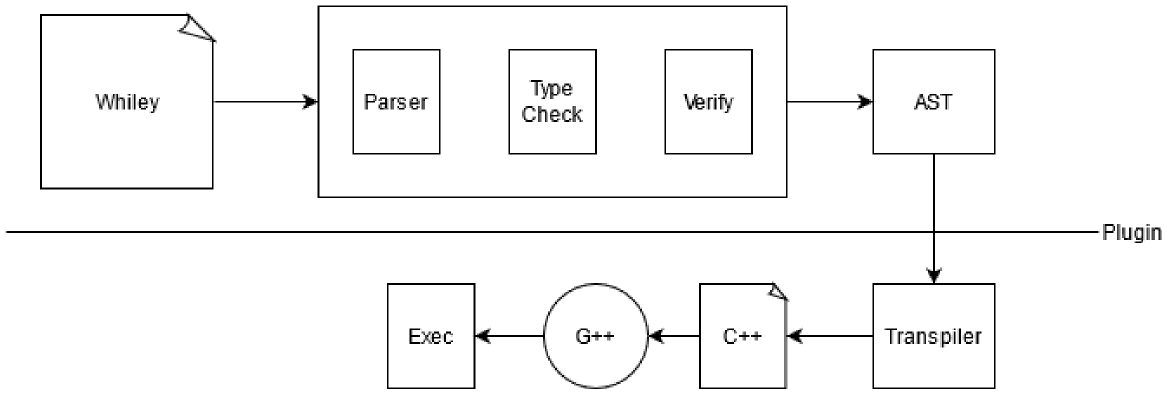

Figure 4.1: The translation process of running an individual test

4.2.1 Recursive Types

In Whiley recursive types are types that are able to hold a reference to the same type. They are useful for describing tree-like structures. For example, a record that holds a record of the same type:

```
type RecType is {int x, RecType rec}
```

This is not possible in the same way in C++ due to each type needing to know the size of the variable it will hold. Containing a type within itself does not compile in C++ when used as a generic parameter. This is the case for the record translation. One possible solution to this is to use a smart pointer as a reference. This would be suited to the `shared_ptr` quite well due to its tree structure.

Another way of implementing a recursive type in Whiley is with the following:

```
type LinkedList is null | Link
type Link is {int data, LinkedList next}
```

In the prototype this would translate to this:

```
using LinkedList = std::variant<void*, Link>;
using Link = std::tuple<int, LinkedList>;
```

This would not compile in C++ due to the order of declarations mattering. This is not the largest issue, however. When allocating how much memory a `LinkedList` would take, C++ variants use the size of the largest type. In this case it is the `Link`. `Link` has a size of `int` plus the size of a `LinkedList`. This would lead to an infinitely sized type which is a problem. Figuring out how to manage the memory would be a key part of a complete implementation.

Chapter 5

Conclusion

5.1 Conclusions

In this project, we design and prototype a C++ backend for the Whiley compiler that is capable of transpiling verified Whiley code into C++ code. The majority of the basic components of Whiley are supported and largely functionally equivalent.

There is defiantly potential to have a fully working C++ backend for Whiley. There are only a few cases that are failing the majority of the tests. Theoretically, the prototype can be expanded for a near-complete translation of the Whiley code.

As a recap from 1.1, the main contributions of this project include:

1. Developed and designed a prototype C++ backend for Whiley
2. Evaluated the backend using the existing Whiley test suite
3. Found failing tests and implemented changes to make the prototype more complete
4. Found problem points of a C++ backend for Whiley

5.2 Future Work

There are a few things that can be done with the findings of this project.

- The issues outlined in the evaluation (4) point out where a full implementation would have difficulties. A full implementation could be developed with these problem points in mind.
- Currently, there is no attempt to translate the Whiley pre-/post-conditions and other specification elements. All of the safety checking in Whiley is done statically meaning that they do not necessarily have to be translated into C++ for verification. They could potentially be translated in future using assert statements but they are ignored for now to avoid out-of-scope complexity.

References

[1] D. J. Pearce and L. Groves, “Whiley: A platform for research in software verification,” in *Software Language Engineering*, M. Erwig, R. F. Paige, and E. Van Wyk, Eds., Cham: Springer International Publishing, 2013, pp. 238–248, ISBN: 978-3-319-02654-1.

[2] D. J. Pearce. (2014). “Whiley language specification,” [Online]. Available: `http://whiley.org/download/WhileyLanguageSpec.pdf`. (accessed: 2021/10/01).

[3] D. J. Pearce and L. Groves, “Designing a verifying compiler: Lessons learned from developing whiley,” *Science of Computer Programming*, vol. 113, pp. 191–220, 2015, ISSN: 0167-6423. DOI: `https://doi.org/10.1016/j.scico.2015.09.006`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S016764231500266X`.

[4] C. Slater. (2015). “Developing a whiley-to-javascript translator,” [Online]. Available: `https://whileydave.com/publications/Slater15_ENGR489.pdf`. (accessed: 2021/10/02).

[5] P. Müller, M. Schwerhoff, and A. J. Summers, “Viper: A verification infrastructure for permission-based reasoning,” in *Verification, Model Checking, and Abstract Interpretation*, B. Jobstmann and K. R. M. Leino, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 41–62, ISBN: 978-3-662-49122-5.

[6] D. J. Pearce. (2017). “Whiley overview,” [Online]. Available: `http://whiley.org/about/overview/`. (accessed: 2021/05/18).

[7] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, “Leveraging rust types for modular specification and verification,” *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. DOI: `10.1145/3360573`. [Online]. Available: `https://doi.org/10.1145/3360573`.

[8] N. Shetty, N. Saldanha, and M. N. Thippeswamy, “CRUST: A C/C++ to Rust Transpiler Using a “Nano-parser Methodology” to Avoid C/C++ Safety Issues in Legacy Code,” *Emerging Research in Computing, Information, Communication and Applications*, vol. 882, pp. 241–250, 2019. DOI: `https://doi.org/10.1007/978-981-13-5953-8_21`.

[9] P. Bhatt, H. Taneja, and K. Taneja, “SSCCJ: System for Source to Source Conversion from C++ to Java for Efficient Computing in IoT Era,” in *Proceedings of International Conference on IoT Inclusive Life (ICIIL 2019), NITTTR Chandigarh, India*, M. Dutta, C. R. Krishna, R. Kumar, and M. Kalra, Eds., Singapore: Springer Singapore, 2020, pp. 393–400. DOI: `https://doi.org/10.1007/978-981-15-3020-3_35`.

[10] (). “Dafny compilation to c++,” [Online]. Available: `https://dafny-lang.github.io/dafny/Compilation/Cpp`. (accessed: 2021/10/02).