Roope Rekunen

Autogenerating Language Bindings for a Codebase

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communication Technology

Bachelor’s Thesis

5 May 2024

Abstract

Author:	Roope Rekunen
Title:	Autogenerating Language Bindings for a Codebase
Number of Pages:	41 pages
Date:	5 May 2024
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Professional Major:	Smart IoT Systems
Supervisors:	Keijo Länsikunnas, Senior lecturer Alex Battiston, Agile Manager

---

The usage of the API of a project is often limited to the programming language that the project is originally written in. To allow for the usage of the API in other programming languages, an interface similar to the API can be defined in a foreign language, also called language bindings. While this can be done manually, doing so comes with additional work, as any updates to an API have to be done for the language bindings as well.

The case company had a situation, where their manually written language bindings for the API of their product were not up to date. To help in this situation, the development of a specialized language binding autogeneration tool was started. The purpose of this project was to ensure that when a developer changes something that is also present in a foreign programming language, the changes would automatically carry over.

In addition to the proprietary tool made for the case company, the development of an open source project called Autoglue began. Autoglue is a library aiming to give developers an interface to create their own autogeneration tools for generating language bindings. Instead of supporting autogeneration from a single language to other languages like many other project do, Autoglue uses the concepts discussed in this thesis to provide abstractions, that can be used to generate language bindings from any language to any language.

In summary, this thesis studies language bindings, their autogeneration and how they work internally. The concepts are covered as generically as possible, while providing more specific examples to give an idea of how things can be implemented.

Keywords: C++, Python, Java, language bindings, ABI, API, Clang

Tiivistelmä

Tekijä:	Roope Rekunen
Otsikko:	Koodikannan autogenerointi muille kielille
Sivumäärä:	41 sivua
Aika:	5.5.2024
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto–ja viestintätekniikka)
Ammatillinen pääaine:	Smart IoT Systems
Ohjaajat:	Lehtori Keijo Länsikunnas Agile-Päällikkö Alex Battiston

---

Ohjelmointirajapinnan eli API:n käyttö yleensä rajoittuu siihen ohjelmointikieleen, millä rajapinta on alun perin kirjoitettu. Käytön muilla ohjelmointikielillä voi sallia tekemällä halutulla kielellä rajapinnan, joka muistuttaa käytettävää APIa. Tätä kutsutaan myös kielisidokseksi. Vaikka kielisidoksia pystyy kirjoittamaan käsin, se lisää työmäärää, sillä jokainen päivitys alkuperäiseen APIin pitää tehdä myös kielisidoksiin.

Yrityksellä, jolle projekti tehtiin, oli ongelma, jossa heidän tuotteensa käsintehdyt kielisidokset eivät aina pysyneet ajan tasalla. Ongelman ratkaisemista varten aloitettiin projekti, jossa kehitettiin työkalu. joka tuottaa yrityksen tuotteelle kielisidoksia. Projektin tarkoitus oli varmistaa, että kun tuotteen APIin tehdään muutoksia, muutokset siirtyvät kielisidoksiin automaattisesti.

Yritykselle tehdyn työkalun lisäksi uuden avoimen lähdekoodin projektin nimeltä Autoglue kehitys aloitettiin. Autoglue on ohjelmointikirjasto, jonka tarkoituksena on antaa kehittäjille rajapinta, jota voi käyttää oman kielisidoksia tuottavan työkalun kehittämiseen. Sen sijaan että Autoglue toimisi kuin monet muut projektit, jotka tuottavat kielisidoksia yhdestä kielestä monelle kielelle, tämän työn esittelemiä käsitteitä hyödynnetään abstraktioihin, joita voidaan käyttää kielisidosten tuottamisseen mistä tahansa kielestä mille tahansa kielelle.

Tämän työn tarkoitus on tutkia kielisidoksia, niiden sisäistä toimintaa ja miten niitä pystyy generoimaan automaattisesti. Käsitteet käydään läpi mahdollisimman yleisellä tasolla, mutta niiden seasta löytyy tarkempia esimerkkejä, jotka antavat käsityksen oikeasta toteutuksesta.

Avainsanat: C++, Python, Java, Kielisidos, ABI, API, Clang

Contents

List of Abbreviations

ABI: Application Binary Interface

API: Application Programming Interface

AST: Abstract Syntax Tree

ELF: Executable Linkable Format

JNI: Java Native Interface

JVM: Java Virtual Machine

:

POD: Plain Old Data

RAII: Resource Acquisition Is Initialization

USR: Unified Symbol Resolution

Glossary

foreign language: A programming language that is foreign to the language that a wrapped API is written in

language bindings: An interface written in a foreign language that mimics an API written in another language

LibClang: A simplified stable interface for Clang

Python API: A library used to interact with a Python virtual machine from another language

wrapped API: An Application Programming Interface (API) that is ported to another programming language

1 Introduction

This thesis explores techniques used for programming language bindings and their automatic generation.

1.1 The problem

The case company is developing a product where primarily only one programming language is used. This is often necessary as most programming languages are not compatible with each other, as datatypes and functions defined in one language cannot be directly used in another language due to differences in syntax and semantics. Limiting a codebase to a single language has the disadvantage of restricting users from using any other language than what the API of the codebase is already written in.

To solve this issue, the case company created manually written interfaces in a foreign language for the product API, also known as language bindings. Manually writing language bindings can be a good entry point to supporting other languages, but on a large scale it is not a realistic way to maintain them. As the product grows, the manually written language bindings would have to be updated to keep up with the work. This gets very impractical as every addition and modification to the API might require manual updates.

1.2 Initial analysis

In order to solve the problem of maintaining manually written language bindings, a tool that automatically generates language bindings had to be created. By letting a tool programmatically generate an entire API for a foreign language, the need for manual API maintenance is terminated and expansion of it is made easy.

While tools for this exact use case already existed, most did not suffice all that well due to them requiring excessive configuration. This is why the case company

ended up choosing to create a custom tool that can be easily adapted to fit the patterns and quirks of the product API with little to zero configuration. Additionally supporting new programming languages is made easier, as automatically generating language bindings gives the given language access to an entire API for free. The drawback of creating a custom tool is that it can take a lot of time and knowledge, but when done it will save even more time as manual API binding or maintenance is no longer as necessary.

While the autogeneration tool project focused on generating specialized language bindings from one language to multiple, the goal of this thesis was to give a generic overview of language bindings and their autogeneration that the reader can utilize to implement their own tool. In order to understand how language bindings can be created and automatically generated, it is important to understand some prerequisites.

2 Different types of programming languages

2.1 Native languages

Native languages are programming languages, which run natively on the processor. They are generally compiled ahead of time, which means that the source code is translated into an executable format containing machine code. [1, pp. 11–12.] Some examples of a native language are C, C++ and Rust.

In order to make source code written in a native language executable, it has to be compiled first. Compilation is the process of parsing and converting source code to a format that can be more easily understood by a computer, such as machine code, or an intermediate abstraction of it. The final form of any source file is an object file, which contains symbols referring to definitions such as functions [1, p. 26] and machine code representing the logic of the program. Object files themselves are not generally executable as they might contain unresolved references, such as calls to functions defined outside the given source file [1, p. 30]. To make a program executable, every source file has to be compiled

separately and then be passed onto the linker.

2.1.1 The linker

Linking is the process of taking multiple object files and combining them together into an executable format, such as Executable Linkable Format (ELF). Linking is done by the linker, which is a program that receives compiled object files and combines them into the given executable format. Running the linker results in an executable file, which can be loaded by the platform that the linking was done for. [1, p. 37]

In order to guarantee that the resulting executable file is valid, the linker might have a closer look at the given object files. An important detail that a linker might want to make sure of is that whenever a symbol is referenced, such a symbol also exists somewhere where the linker can find it. If left undone, running code from the resulting executable file might have unexpected side effects, such as a crash when a function without a sensible implementation is invoked. [1, p. 30]

2.1.2 The loader

Before any execution can be done, the executable format has to be loaded. The loader is a program that parses executable formats made by the linker and loads them into the memory for execution [1, pp. 45–46].

Much like the linker, the loader might make sure that any referenced symbols exist somewhere where the loader can find them. Because an executable format might be loaded outside the environment that it was linked in, such a check is important in order to avoid unexpected behavior.

2.1.3 Symbol linkage

Symbol linkage determines the visibility of the given symbol within an object file.

Compilers generally default to internal linkage, which means that symbols are generated with a name conforming to a specific scheme. This means that the resulting object files can contain symbols with such ambiguous names that realistically only the same compiler that initially generated them has an easy way to reference them. An example of such behavior is a compiler feature called name mangling that is implemented by C++ compilers.

Name mangling in C++ adds additional information to generated symbol names, such as parent classes and parameter type information. Having this verbosity in the symbol names allows for things like function overloading, where multiple functions of the same name exist, but with different parameters.

```
void foo(int a)
{
}

void foo(float a)
{
}
```

```
_Z3fooi:
        push    rbp
        mov     rbp, rsp
        ...
_Z3foof:
        push    rbp
        mov     rbp, rsp
        ...
```

Listing 1: Two overloads of function foo and their machine code counterparts generated by GCC 12.2 x86-64 that demonstrate name mangling.

As shown in listing 1, the different overloads of function foo have additional information in their generated names, which differentiates them from each other. When a compiler notices an invocation of a specific overload, it can generate machine code that invokes the corresponding function.

In a simple programming language such as C, where no advanced features such

as function overloading and nested function definition are supported, name mangling is irrelevant. This simplicity allows for a more straightforward output as names are not ambiguated, which is also called external linkage.

```
void foo(int a)
{
}
```

```
foo:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        nop
        pop     rbp
        ret
```

Listing 2: Simple C Function and its machine code counterpart generated by GCC 13.2 x86-64 that show how the function name remains the same.

2.1.4 ABI and calling conventions

An Application Binary Interface (ABI) defines how data should be accessed by the processor. This includes information such as how parameters should be passed upon a function invocation [2, pp. 15–23] and how data is represented [2, pp. 11–14].

Due to the ABI defining how data should be represented, native programming languages generally come with a static type system. In order for a compiler to generate machine code that treats data in the same way as specified in the source code, any data type that the programmer uses should be known at compile time. This makes it possible for the compiler to be aware of what kind of a data type the programmer intends to use and treat it accordingly as specified in the platform ABI [2, pp. 11–14].

2.2 Bytecode languages

Bytecode languages are programming languages which are executed inside a virtual machine instead of directly on the processor. Instead of compiling down to machine code, bytecode languages represent the source code as bytecode which is much like machine code, but is designed to work with the given virtual machine instead of a processor. While native languages generally can only run on the platform that they are compiled for, bytecode languages can technically run anywhere as long as the virtual machine is available.

Bytecode languages are generally interpreted, which means that the virtual machine reads the bytecode and does decisions based on it. Such decisions could include the creation of a variable, the evaluation of an arithmetic operation or the invocation of a function. Because the virtual machine is responsible for everything that is happening within the program, it also has to store all data that is being managed. Depending on the language, this can allow for reflecting on type details of the given object or even the original program hierarchy. The type system also is completely up to the language as even a dynamic typing where the type of an object can change at any time is possible.

3 Externally callable functions

As discussed previously, programming languages are able to call functions that have been defined within the same language, but what if a function defined by one language should become callable from another? Externally callable functions can be used to provide such functionality.

3.1 Exposing functions from native languages

In native languages, externally callable functions can be defined as functions that make use of external linkage, which can be used to make a symbol more easily accessible.

If a compiler ambiguates the names of the generated symbols such as function names, the functions cannot be easily accessed from other languages who do not know how the name was ambiguated. To expose such functions to other languages, all ambiguation should be removed to guarantee that the given function can be resolved just by its name. This makes it easier for a foreign language to call the said function.

In a language such as C++ where name mangling is present, the name ambiguation can be removed by the usage of C-style linkage, which means that the compiler will generate a C-compatible function like shown in listing 2[3].

```
void foo()
{
}

extern "C"
void foo(int value)
{
}

void foo(int value1, int value2)
{
}
```

```
_Z3foov:
        push    rbp
        mov     rbp, rsp
        ...
foo:
        push    rbp
        mov     rbp, rsp
        ...
_Z3fooii:
        push    rbp
        mov     rbp, rsp
        ...
```

Listing 3: Multiple overloads of function foo with the corresponding machine code generated by GCC 12.2 to demonstrate the effect of applying C-style linkage.

As shown in listing 3, the function overload that has been given C-style linkage has a much simpler name compared to the others, much like the function shown in listing 2.

3.2 Bridge functions

While it is easy to unambiguate the linkage of the given function, problems could occur when it is done for multiple functions of the same name. For example, the linker might get confused when there are multiple symbols of the same name and refuse to finish the linking process. A possible solution to this problem is to define functions that have unambiguous names, which still have some verbosity that can be used to differentiate them.

Bridge functions are externally callable functions that simply call their original counterparts. Such functions should have a verbose name that provides some context, such as the containing classes or parameter type information. While this sounds a lot like name ambiguation, the naming of a bridge function is a lot more predictable, since the pattern can always stay the same unlike a compiler specific name ambiguation scheme.

```
class Foo
{
public:
  void bar(int value, float otherValue);
};

extern "C"
void Foo_bar(void* objectHandle, int value, float otherValue)
{
  Foo* fooInstance = static_cast <Foo*> (objectHandle);
  fooInstance->bar(value, otherValue);
}
```

Listing 4: An example of a how a bridge function can be made for a C++ class member function.

Defining bridge functions has the advantage of not having to adjust the original functions in the wrapped API to make function invocation easier for a foreign language. As shown in listing 4, by defining an externally callable function that simply calls another function based on the given data, no modifications need to be done to the original function. To support things like function overloading, additions such as the parameter type information or an index can be made to the bridge function name.

3.2.1 Type abstraction

To make sure that any given programming language can correctly call bridge functions, language specific data types such as class types should become Plain Old Data (POD). POD types such as integers, floating point numbers and pointers are ideal for passing data between different languages, as they are the most trivial known types as standardized by the platform ABI [2, pp. 11–14]. Another important aspect is that when a bridge function is called, the calling language has to conform to the platform ABI in order for it to correctly pass parameters of the correct format.

3.2.2 Object handle

As seen in listing 4, the bridge function Foo_Bar has one additional argument as the first one as shown in 4. A common pattern that languages supporting member functions use is forcing the code to access the member functions through an object of the parent class. Taking the object handle as the first parameter allows for hiding the details of how a member function should be accessed through the given object, as the bridge function can do it on behalf of the caller.

3.2.3 Constructors

In order to instantiate an object of any given data type defined in another language, a bridge function can be made to do it. When a bridge function for instantiating the given data type exists, a foreign language does not have to know the details of how an object is actually instantiated as the bridge functions does it

on behalf of the caller.

Since constructor bridge functions are supposed to create objects, the created objects should be returned to the caller in an abstract format such as a typeless pointer that the original language can treat appropriately. This is what makes the invocation of member functions defined in other languages work, as a foreign language can request a handle to an object, which can then be passed to another bridge function as the object handle.

```
extern "C"
void* Foo_construct()
{
  // Return a heap allocated object of type Foo.
  return new Foo;
}
```

Listing 5: An example C++ bridge function that constructs an object of type Foo shown in listing 4.

3.3 Exposing functions from bytecode languages

Defining externally callable functions in bytecode languages depends on the virtual machine, as that is what runs the code. Bytecode languages such as Java and Python provide an API, that can be used to interact with the language in native code [4; 5]. This interaction could mean the invocation of functions defined within the bytecode language.

In order to expose a function defined within a bytecode language to a foreign language, the foreign language would have to have support for the API provided by a virtual machine that can be used to perform function invocations. This can narrow down the amount of language candidates as it is not guaranteed that the API provided by the given bytecode language has support for the foreign language. Instead of performing the function invocation directly in a foreign language, bridge functions can be utilized to hide the details of how a function should be invoked.

Like shown in listing 4, a similar bridge function can be implemented that invokes the given function from a bytecode language. It is important that the runtime required by a bytecode language, such as the virtual machine is setup before attempting to call the given exposed function as it is responsible of executing the code.

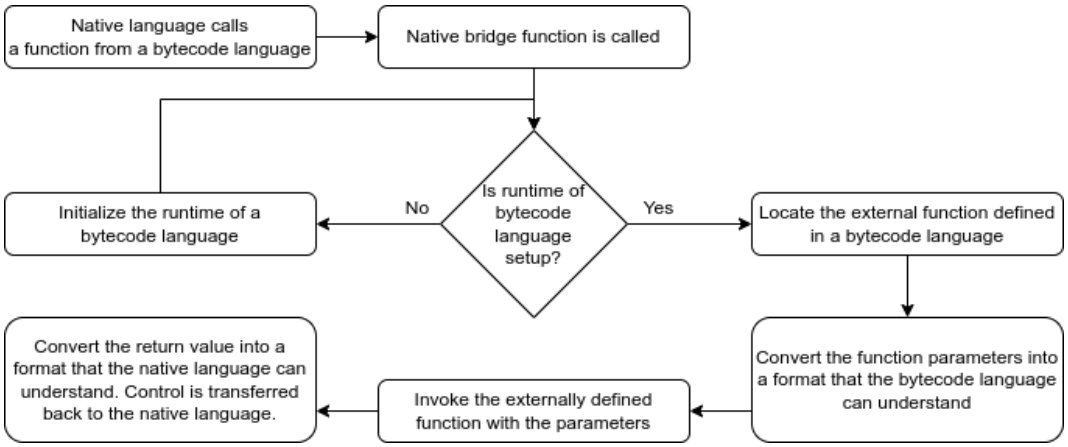

Figure 1: An example of how a native language might call an externally callable function defined in a bytecode language.

For example, Java enforces the usage of the Java Native Interface (JNI), which is a library made for interacting with objects living inside the Java Virtual Machine (JVM) [4]. Since the JNI allows for the invocation of Java methods, an externally callable callable bridge function that invokes the given Java method through the JNI can be defined.

```
extern "C"
void Foo_bar(void* obj, int value, float otherValue)
{
  // Find the class type of the object.
  auto object = reinterpret_cast <jobject> (obj);
  auto classType = env->GetObjectClass(object);

  // Invoke the bar method that has the appropriate parameters.
  auto method = env->GetMethodID(classType, "bar", "(IF)V");
  env->CallVoidMethod(object, method, value, otherValue);
}
```

Listing 6: An example bridge function for the function bar in class Foo if it were written in Java.

3.4 Bridge function invocation

As discussed, externally callable functions can be defined in order to make them accessible from other languages. When bridge functions are implemented as native functions with a verbose name, the programming language that wants to call the given bridge function has to be able to resolve it first.

3.4.1 Invocation from native languages

If a native language can expose functions using external linkage, it probably can resolve externally callable functions defined elsewhere. On a lower level such as assembly, it can be easy to resolve an externally callable function, as the given function just has to be referred to which makes the linker resolve it. Higher level languages such as C and C++ are more strict, as they want to know what the function signature looks like. The function signature generally contains types for the function parameters and the value that the function returns

Native languages such as C and C++ have a feature, where a function can be invoked even if it is defined in another language [3].

```
extern "C"
{
  // Resolve an externally defined function that has C-style
    linkage.
  extern void Foo_bar(int, float);
}

int main()
{
  // Call the externally defined function.
  Foo_bar(1, 2);
}
```

```
main:
        push    rbp
        mov     rbp, rsp
        mov     eax, DWORD PTR .LC0[rip]
        movd    xmm0, eax
```

```
        mov     esi, 1
        mov     edi, 0
        call    Foo_bar     # The invocation of Foo_bar happens here.
        mov     eax, 0
        pop     rbp
        ret
.LC0:
        .long   1073741824
```

Listing 7: An example of resolving and invoking an external C-compatible function in C++ with the corresponding machine code generated by GCC 12.2 x86-64.

As discussed previously, when the externally defined function is referred to, it will be resolved by the linker. If such a function definition is unavailable to the linker, an error about the unresolved reference can be reported.

3.4.2 Invocation from bytecode languages

Because bytecode languages run on a virtual machine, there is no direct way to call a function from another language, unless an explicit interface exists for it. While it would be possible for a virtual machine to define an interface where function calls can be made from any given language, it would be too specific to implement an interface for every possible programming language in every possible virtual machine. To allow for a bytecode language to invoke externally defined functions, virtual machines such as the JVM and the python virtual machine support invocation of native functions [4; 5]. When bridge functions are utilized with this functionality, function calls from any given programming language become possible as long as there are bridge functions to support them.

```
public native void Foo_bar(long thisPtr, int value, float
    otherValue);
```

Listing 8: Java method declared as native to let the JVM know that it is a native function.

In order for a bytecode language to call a native function, the virtual machine that runs the code has to able to access the function. Some virtual machines such as

the JVM are able to dynamically load a library from which they can locate symbols. For an example, when a function in Java is marked native, upon a call to it the JVM will try to locate a function with a special name.

```
extern "C"
{

// Resolve the bridge function for bar.
extern void Foo_bar(void*, int, float);

// Define a JNI function that calls the resolved bridge function.
JNIEXPORT void JNICALL Java_com_roopereku_Foo_bar(JNIEnv *, jobject,
    jlong thisPtr, jint value, jfloat otherValue)
{
  Foo_bar(reinterpret_cast <void*> (thisPtr), value, otherValue);
}

}
```

Listing 9: JNI glue code used to call bridge functions defined for ExampleClass as shown in listing 10

3.4.3 Calling convention compatibility

In order to ensure that a bridge function can consistently be called from different languages, it is important to note calling convention compatibility. As discussed previously, forcing the usage of C-style linkage makes a function defined in C++ callable from C. In addition, calling a function from C++ that has C-style linkage makes the compiler generate code that uses calling conventions compatible with C [3].

While C-style linkage can be used to expose a C++ bridge function to other languages, they might not treat the bridge function as a C compatible function unless specified otherwise. Because of such a possibility, every language that desires to use the given bridge function should treat the function in the same way, such as a C-compatible function.

3.5 Communication between different bytecode languages

For any given combination of bytecode languages where one language can define externally callable functions and another can call externally callable functions, communication between bytecode languages can be implemented by using the techniques discussed previously.

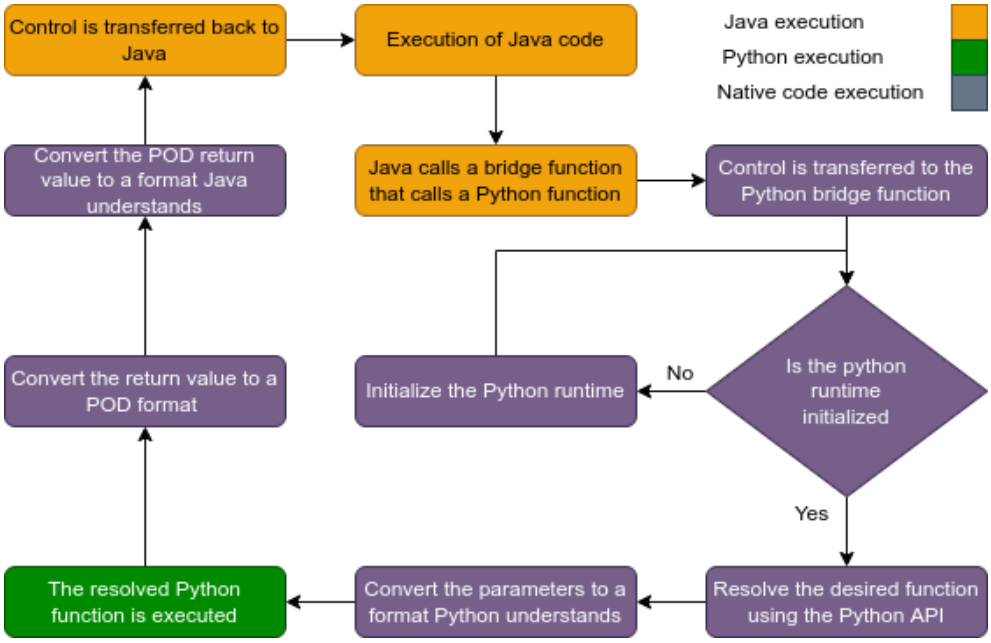

Figure 2: A demonstration of what might happen, when an externally callable function defined for Python is invoked during Java execution.

4 Language bindings

Programming language bindings are a collection of interfaces, which the user of the given language can use to access an API defined in another language.

As discussed previously, bridge functions can make it easy for other languages to invoke functions defined in the given language. Because an API generally consists of functions, language bindings can utilize bridge functions to get easy access to the original functions of the wrapped API. Instead of having users directly use the bridge functions, the language bindings can provide another set of functions that more closely resemble the original API.

```
class ExampleClass
{
public:
  void useValue(int value)
  {
    mValue = value;

    // Do something else...
  }

private:
  int mValue;
};
```

```
public class ExampleClass
{
  // Functions that call ExampleClass bridge functions in the JNI.
  private native long ExampleClass_construct();
  private native void ExampleClass_useValue(long objHandle, int
    value);

  // Pointer pointing to an object of ExampleClass.
  private long objectHandle;

  public ExampleClass()
  {
    objectHandle = ExampleClass_construct();
  }

  public void useValue(int value)
  {
    ExampleClass_useValue(objectHandle, value);
  }
}
```

Listing 10: An example class written in C++ and simple language bindings for it made in Java.

As shown in listing 10, many previously discussed techniques, such as bridge functions and object handles are combined into a Java class that mimics the C++ version of ExampleClass. With this interface, the users of Java are able to interact with ExampleClass like they can do in C++. It is important to note that in order to

create functional language bindings for Java, the interface is not enough as additional JNI code like shown in listing 9 is required.

4.1 Inheritance

Inheritance is a feature in programming languages that can be used to extend previously defined classes with new functionality. When a class derives another, on top of its own contents it will also contain everything that a base class has specified to be inheritable [6, pp. 2–3]. For example, if a member function is inherited from a base class, it will be available in the derived class.

4.1.1 Class hierarchy

Multiple classes being related to each other through inheritance is referred to as class hierarchy. In language bindings, to mimic an API as accurately as possible, it can be a good idea to copy as much of the class hierarchy defined in the wrapped API as possible. Copying a class hierarchy simply means that for any class that should be exported to the language bindings, base classes are also exported and inheritance is set appropriately to make the classes related. When the class hierarchy is present, an object of any given class type from the language bindings can be treated as its base type, assuming that the given foreign language supports it. This allows for the user of the language bindings to invoke of any function defined in a base class through an object of the derived class.

4.1.2 Object handle details

When there are multiple layers in the class hierarchy, it is a good idea to only store the object handle once. Because a constructor bridge function for the given type only returns a single unique object handle, there is no reason to have multiple copies of it. Inheritance can be used to grant derived classes access to an object handle, that is stored in the root of the class hierarchy which has no further inheritance.

In order for a derived class to initialize the object handle, it first should call a constructor bridge function for the derived type, which returns a unique object handle. All classes within the class hierarchy can define a constructor, which accepts an object handle and depending on if the class is the root of the class hierarchy, it either can delegate the object handle to a base class recursively, or initialize the object handle.

4.1.3 Abstract classes

In the case where a class is not supposed to be instantiable, but inherited, an abstract class can be utilized. While in normal classes the member functions have a single definition, an abstract class can have member functions that have to be defined by derived classes [6, p. 19].

Generally the instantiation of an abstract class is forbidden due to incomplete member function definitions. While an interface for a foreign language can easily be created, the bridge function for instantiating an abstract class will likely not compile. This however can be worked around by creating a hidden class in the instantiating bridge function, which simply implements the required member functions. Instead of instantiating an object of the abstract class, an object of the hidden extension class can be returned to a foreign language.

Once an abstract class becomes instantiable through a constructor bridge function, it is possible to call its member functions from a foreign language. If the given foreign language supports class abstraction features, the language bindings can mark the appropriate member functions as abstract, which forces the user to implement those functions for their own classes. Once a user gives their own definitions for what the given abstract function should do, the wrapped API can call upon the custom functionality; however, since the inner workings of this can get quite complicated, further exploration of the topic was left out of this thesis.

4.2 Object lifetime

Like shown in listing 10, a class in a foreign language that mimics any given class from a wrapped API should have a handle to an object allocated by the original language and an invocation to a constructor bridge function to ensure that the object handle refers to something. Depending on the language of the wrapped API, more management might have to be done for an object that is allocated outside the language bindings. Because different languages can define object lifetime however they want, such details should be hidden from the language bindings. Much like how bridge functions are created to hide details of how a function should be invoked, a possible way of abstracting object lifetime is to let constructor bridge functions allocate an object in a way that allows it to stay alive until specified otherwise.

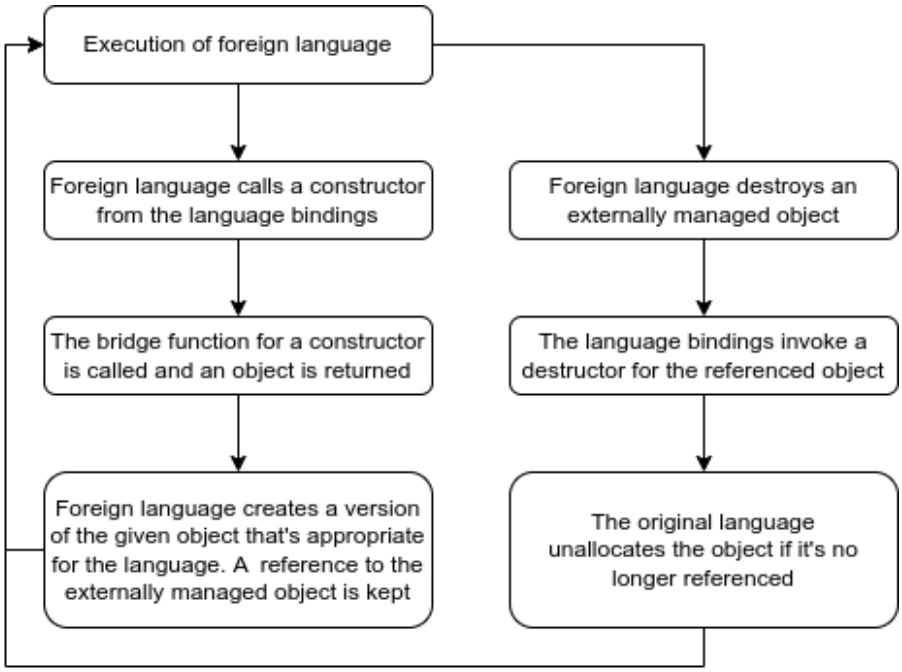

Figure 3: An example of how an externally managed object could be handled.

4.2.1 Unmanaged languages

Unmanaged languages such as C and C++ require the programmer to manage all of the memory that they have explicitly allocated. Explicit memory allocations are stored on the heap, which is a non-contiquous data structure used to allocate

objects of dynamic size. Objects allocated on the heap are not destroyed automatically and have to be cleaned up by the programmer which allows for a customizable object lifetime.

In addition to explicit memory allocations there are implicit memory allocations. Temporary objects such as variables within a function are allocated implicitly without any extra indication that an explicit memory allocation might have. Instead of allocating temporary objects on the heap, they are allocated on the stack which is a contiguous data structure where all of the data is piled on top of each other. [7, pp. 2–3]

In a language like C++ where classes and member functions exist, code can be executed upon the construction and destruction of an object. Specifically in C++ this is called Resource Acquisition Is Initialization (RAII), which means that whenever resources are acquired, initialization should also be done. While the name only talks about initialization, it also applies to object destruction ensuring that members are also uninitialized. [8, p. 14] An example usage of RAII is a class managing a heap allocation, where the constructor allocates memory from the heap and the destructor unallocates the memory. When a temporary object of such a class is created, the heap allocation happens and when the lifetime of object the object ends, the heap deallocation happens.

4.2.2 Managing objects from an unmanaged language

Management of an object that originates from an unmanaged languages is important in order to not leak memory. Because heap allocated objects will not disappear on their own, the language bindings have to make sure that when a foreign language object representing a type from an unmanaged language is destroyed, the associated heap allocation is deallocated.

When creating language bindings for an unmanaged language, there is also the possibility that a function will return a temporary value. Since such a value will be destroyed if not copied elsewhere upon the function exit, the language bindings

should allocate that value onto the heap. Doing so extends the lifetime of the said object as it now has to be explicitly deallocated, which is done by the language bindings.

4.2.3 Managed languages

Managed languages such as Java and Python manage the memory on the behalf of the programmer. One of the most common ways that managed languages manage the memory is through a garbage collector, which is an entity that cleans up unreferenced memory allocations. Depending on the language, there might be multiple different implementations of a garbage collector, which are more optimal for different scenarios. An example of such a case is Java [9].

4.2.4 Managing objects from a managed language

Management of an object that originates from a managed language is important, since when a foreign language receives an object through a function defined in a managed language, the object will likely be destroyed if left unmanaged. The language bindings can retain an object by telling the managed language that it has a new reference. This makes the managed language not destroy the object until the reference is released.

Because the details of constructing an object are language and virtual machine specific, it is a good idea to hide them behind a bridge function. For example, if language bindings are made for a Python library, a new object of a Python class can created using functions from the Python API [5].

```
extern "C"
void* ExampleClass_construct()
{
  // Resolve ExampleClass and create a new object of it.
  PyObject* classObject = PyObject_GetAttrString(moduleObject,
    "ExampleClass");
  PyObject* newObject = PyObject_CallObject(classObject, nullptr);

  // Clean up the temporary class object reference.
  Py_DECREF(classObject);

  // Return the newly created object to the user.
  return newObject;
}
```

Listing 11: Python bridge function that would create an object of class ExampleClass shown in listing 10, if it was written in Python.

4.2.5 Object ownership

When a foreign language creates a new object through the language bindings, the foreign language owns the object as long as it is allocated by a bridge function. This is called owning an object, which means that the lifetime of the object is determined by the foreign language, as the construction and destruction are done by the language bindings. An object can also be referenced, which means that the object is managed by the wrapped API. In such a case the referenced object is not destroyed when it's no longer accessible by a foreign language.

Depending on the language that a wrapped API is written in, some management may be needed for a referenced object. This mostly applies to managed languages that determine the lifetime of an object by counting references. When a reference object is destroyed in a foreign language, a destructor bridge function that decrements the reference count of an object could be invoked by the language bindings. This way a foreign language can ensure that a referenced object stays alive as long as there's at least one active reference.

```
extern "C"
void* ExampleClass_GetSomeObject(void* self)
{
  // Call a Python member function that returns some object.
  auto result = PyObject_CallMethod(
    static_cast <PyObject*> (self),
    "get_some_object", NULL
  );

  // Return a new reference for the resulting object.
  return Py_NewRef(result);
}
```

Listing 12: Python bridge function that saves a reference to an object returned from a Python function.

5 Autogeneration of language bindings

As discussed previously, language bindings can be implemented by creating functions that can be called across languages. While manually writing language bindings can be a great entry point for smaller projects, maintenance and support gets incrementally more impractical as more of the API is covered. This problem can be solved by automatically generating the language bindings.

Automatic generation is the process of converting some existing data into new data automatically. In the context of language bindings, any given API can be treated as input data which can be converted into language bindings for the given programming language.

5.1 Abstract syntax tree

When a binding autogeneration tool is fed source code, it first has to make sense of it in order to output something appropriate. A common format that can be used to describe source code is called an Abstract Syntax Tree (AST), which is a tree-like data structure that contains elements of the source code in a way that

can be easily understood by computers.

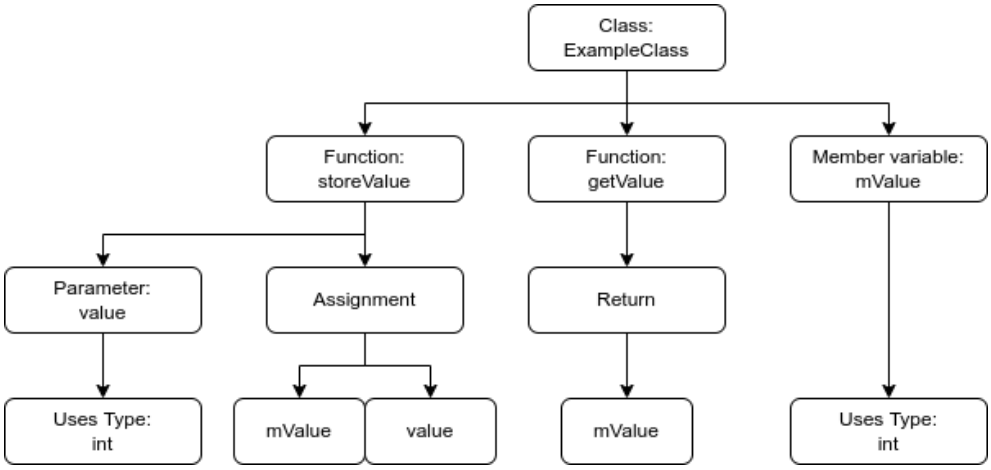

Figure 4: An example of how an AST might look like.

Though the contents of an AST are implementation defined, common concepts such as variable definitions, function calls and arithmetic operations are likely to be present.

5.2 Simplified hierarchy

Autogenerating language bindings directly from the AST might not be the most desirable option, since the structure and contents of different AST definitions might differ significantly. If only an AST was used to generate language bindings, every generator would have to support every type of AST format. The better option is to convert an AST into another intermediate format, which describes the hierarchy of the given API in a simple way.

The simplified hierarchy should only contain things that are common to most languages such as classes and functions. Anything that isn't accessible through the original API such as hidden class members or function implementations should be left out as they serve no purpose.

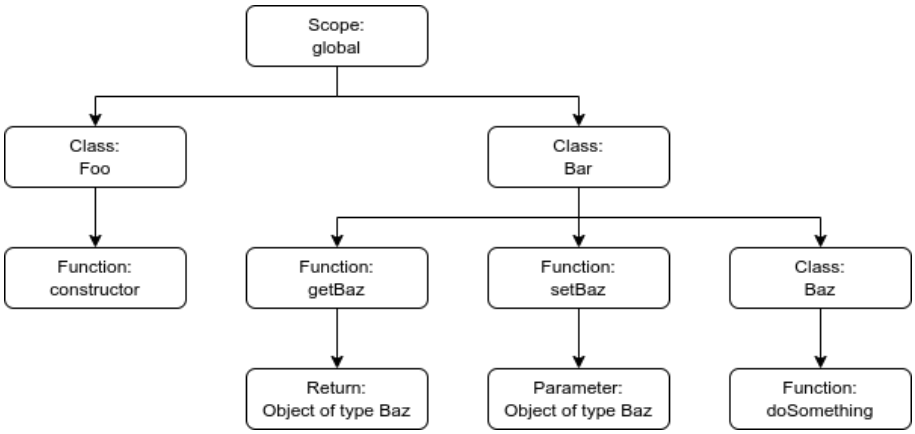

Figure 5: A demonstration of how a simplified hierarchy might look like.

Before an entity is added to the simplified hierarchy, it is a good idea to make sure that its parent entity already exists and add the new entity under the parent. This should be done to ensure that the structure of a wrapped API stays intact. If left undone, some entities such as classes or functions may appear in the wrong place.

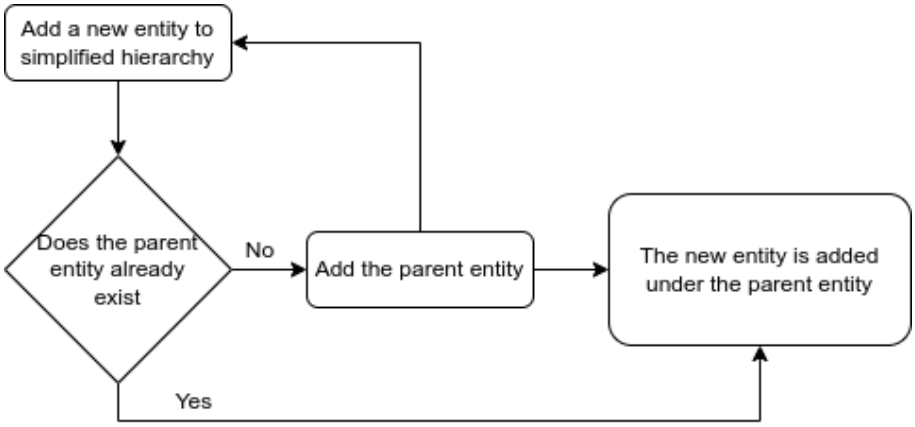

Figure 6: An example flowchart describing how the parent entities can be added recursively.

5.3 Generating bindings from a simplified hierarchy

Once a simplified hierarchy has been constructed, language bindings for different languages can be generated using it. As long as the data within simplified hierarchy is as generic as possible, the language that the original API has been written in shouldn't matter as the hierarchy contains all of the necessary information in a format which allows for any foreign language to utilize it.

The generation of language bindings should be as simple as traversing through the simplified hierarchy and writing entities in a way that's appropriate for the given foreign language.

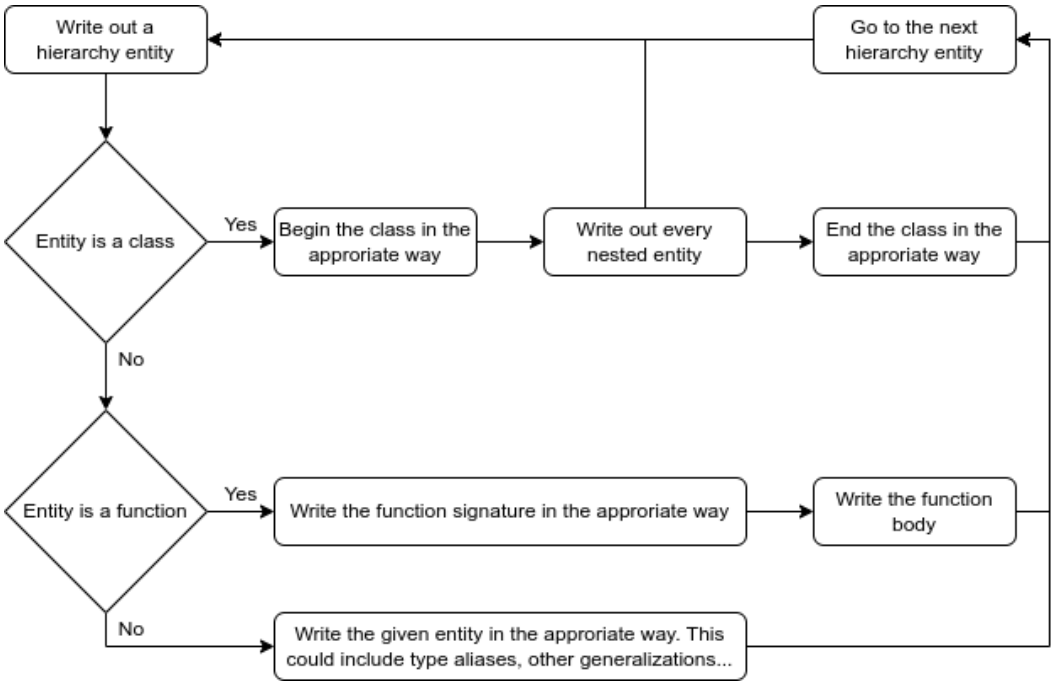

Figure 7: An example of what language bindings generation could look like.

5.3.1 Class generation

The most important thing to generate inside a class is the object handle representing the actual class object that originates from the wrapped API. Without it, calling bridge functions representing class member functions might become impossible. As discussed previously, an appropriate location for an object handle is any class that does not inherit anything.

To utilize inheritance, definitions for the base classes of any given class can be generated recursively, which ensures that anything that is inherited also exists. The names of these base classes can then be generated in an approriate way, which allows for specifying class inheritance.

Finally, nested declarations such as member functions and other definitions such as type aliases, variables, or even other classes should be generated. Because

other languages might not know details of how visible variables should be accessed, they could require some additional handling such as getter and setter functions. Bridge functions can be generated for every visible variable, which are then exposed by the class in the foreign language.

5.3.2 Function generation

Automatic generation of functions in language bindings require knowing the function name, return type and parameter types. All of this information should be present in the simplified hierarchy as each of these properties are common in programming languages.

In general, the generation of functions is as simple as generating the return type name, the function name and the parameters. Finally the corresponding bridge function is called inside the generated function. The simplified hierarchy should contain the bridge function name, as it is a common property for any language bindings.

If the function resides inside a class in the simplified hierarchy, the object handle should also be passed to the bridge function. If the language bindings don’t have classes to put functions in, the object handle should explicitly be a parameter.

5.3.3 Bridge function generation

To make sure that language bindings are able to call functions from the original API through bridge functions, they have to be generated as well. The same idea of traversing the simplified hierarchy can be utilized, but in order to only generate bridge functions, the generator should only be interested in functions. Generation of things such as classes is irrelevant, as instances of them are passed as POD types, such as typeless pointers.

Because the invocation of a function depends on the programming language, the bridge function generator should be aware of what language it is generating bridge

functions for. In addition, parameter passing depends on the programming language, which is why the bridge function generator should be aware of how to convert POD types, such as a parameter, or the return type to the appropriate format.

In order to give the bridge function generator more context for things like parameter types, the simplified hierarchy can contain the context in any node where it is needed.

5.4 Autogenerating language bindings from C++

In order to generate language bindings for a C++ codebase, a simplified hierarchy has to be constructed from an AST that's received from Clang. Clang is a frontend for multiple programming languages in the C-language family, such as C, C++ and Objective C [10]. While using Clang allows for generating language bindings for codebases written in many languages such as C and Objective C, this section will only focus on C++.

The Clang project has multiple interfaces that can be used to gain access to an AST, such as LibTooling and LibClang [11]. For the sake of simplicity, this section will discuss LibClang.

5.4.1 LibClang AST

In C++ an API is generally defined in header files, which are files that contain things like function declarations and class definitions. To gain access to an AST representing an API, LibClang can be told to parse source files that implement the given API.

In order to prevent compilation errors such as a missing definition, or a missing header inclusion, some additional compilation options might need to be passed to LibClang. Such information can be extracted from a compilation database, which is a file containing instructions on how a source file should be compiled [12].

```python
import clang.cindex

# Create an index for Clang and load a compilation database.
index = clang.cindex.Index.create()
db = clang.cindex.CompilationDatabase.fromDirectory("buildDir")

# Iterate all files within the compilation database.
for cmd in db.getAllCompileCommands():

    # Parse the given source file with the given arguments.
    # Redundant compilation arguments can be filtered out.
    unit = index.parse(cmd.filename,
        args=filter_arguments(cmd.arguments))

    # Start custom AST traversal for the AST from this source file.
    traverse_ast(unit.cursor)
```

Listing 13: Example of how an AST can be created from a C++ codebase using the LibClang Python bindings.

LibClang populates an AST with cursors, which is just another name for an AST node. A cursor can hold any type of an entity that can be found in C++ such as a function declaration, a variable declaration or a class declaration. The type of a cursor can be differentiated by checking the value of the cursor kind member.

```python
def traverse_ast(root):
    # Do a depth first traversal of the AST.
    for cursor in unit.cursor.walk_preorder():

        # Check if the cursor represents a class declaration.
        if cursor.kind == clang.cindex.CursorKind.CLASS_DECL:
            print(f'Found class {cursor.spelling}')

        # Check if the cursor represents a member function
            declaration.
        if cursor.kind == clang.cindex.CursorKind.CXX_METHOD:
            print(f'Found member function {cursor.spelling}')
```

Listing 14: An example implementation of traverse_ast as used in listing 13.

5.4.2 Constructing a simplified hierarchy

Once the AST is being traversed, it's rather easy to construct a simplified hierarchy from the given cursor since Clang provides easy access to the program hierarchy. To find out the location of any given cursor, Clang provides an identifier called the Unified Symbol Resolution (USR), which is a very verbose string containing all sorts information that's related to the cursor. For example, a class called ”Foo” inside a namespace ”Bar” would have a USR looking something like ”c:@N@Bar@S@Foo”.

While parsing the location of a cursor from the USR can seem convenient, the usage of more complex C++ features such as templates can make the USR difficult to interpret. To remove such complexity from the hierarchy generation, it might make more sense to stay within the domain of cursors instead of USR parsing. LibClang provides an easy access to the cursor that contains another cursor through an element called the semantic parent. [13] This allows for a method, where the AST root is located through the parent cursors, and one by one it is made sure that the simplified hierarchy contains every entity leading up to the given cursor, like shown in listing 6.

5.4.3 C++ bridge function generation

As discussed previously, bridge functions can be generated to provide language bindings an interface, through which they can invoke functions of the wrapped API without having to know any language specific details.

The generation of C++ bridge functions has two steps. The first step is generating inclusion directives for any types or functions that are being exposed through the language bindings. In C++, including a source file injects the contents of the said file to where the inclusion takes place [14], which in this case is a source file that contains bridge functions. Including files where the required types are defined is necessary, as C++ forbids the usage of incomplete types [15]. Especially in the case of a class based API, the class type definitions are important since member

function invocations depend on them.

The second step is generating the actual bridge functions. As discussed previously, a bridge function should be made in a way that enables other languages to easily call it. Assuming that an autogeneration tool generates language bindings that expect bridge functions to be C-compatible functions, the bridge functions that connect the wrapped API written in C++ to the language bindings can be marked with C-style linkage like shown in listing 4.

As long as every relevant inclusion exists, functions can simply be invoked with the given parameters. Because every bridge function parameter should be POD, some typecasts may need to be done for parameters such as class objects. When a bridge function is generated for a class member function, this also applies to the object handle parameter. The object handle passed to a C++ bridge function can be typecasted to the type which is the parent of the given function within the simplified hierarchy.

```
extern "C"
void Bar_Baz_doSomething(void* objectHandle)
{
  static_cast <Bar::Baz*> (objectHandle)->doSomething();
}
```

Listing 15: An example of how the doSomething member function from the Baz class shown in figure 5 could be accessed if it were written in C++. This is trivial to autogenerate.

5.5 Autogenerating language bindings from Python

In order to generate language bindings for a Python codebase, a simplified hierarchy has to be constructed from an AST that describes Python code. The AST module in Python can be used to compile Python source code and receive an AST representation of it.

5.5.1 Python AST

Much like what LibClang provides, the Python AST contains nodes which can hold any type of an entity that can be found in Python, but unlike LibClang, in the Python AST each node has a different class type. Node types in the Python AST can be differentiated by checking the actual class type of the node. [16]

```
import ast

def traverse(node):
    # Is this node a class definition.
    if isinstance(node, ast.ClassDef):
        print(f'Found class definition {node.name}')

    # Is this node a function definition.
    elif isinstance(node, ast.FunctionDef):
        print(f'Found function definition {node.name}')

    # Recurse into the child nodes.
    for child in ast.iter_child_nodes(node):
        traverse(child)

# Read a file containing an API and parse it to an AST.
with open('api.py', 'r') as file:
    tree = ast.parse(file.read())

traverse(tree)
```

Listing 16: Example of AST traversal using the python AST module. This example expects to find a Python file called api.py.

5.5.2 Constructing a simplified hierarchy

Python is a dynamically typed language, which means that the data types of function parameters and the return value can be anything at any time. While it is possible to explicitly specify what data types are accepted for the given Python function, it is completely optional [17]. This is fine in the case of manually written bindings, since the developer might have some context on how Python functions should be exposed to foreign languages. When language bindings are being

automatically generated from Python, it is quite important for the target API to contain explicit type information, as that's the only context that an autogeneration tool will implicitly have.

```
class Foo:
    def __init__(self, value: int):
        self.value = value

    def get_value(self) -> int:
        return self.value
```

Listing 17: A simple API written in python with explicit typing.

While it is possible to simulate dynamic typing in strongly typed languages without specifying what kind of parameters a python function might accept, there is a high risk that something unexpected will happen if the function doesn't guarantee that a given type will work. The same problem can also occur within Python when an API doesn't provide any context of how it should be used.

An alternative way of guessing the type information is through analyzing the Python AST. By checking what the given function does, there is a possibility that parameter type information and the return type can be guessed without explicitly specifying them; however, this might always not be possible if there are not clear giveaways.

5.5.3 Python bridge function generation

In order for other languages to invoke a python function, they need to have access to the Python API through which the Python virtual machine can be told to invoke the given function [5]. To expose a python function to as many languages as possible, bridge functions can be generated that hide all of the details, such as how a function can be resolved and how it can be invoked.

The Python API provides simple functions to resolve things such as classes and functions. For example, when a constructor bridge function for Python is being

generated, a function can be generated that simply resolves a class of the given name, of which an instance is created.

```
extern "C"
void* Foo_construct(int value)
{
  PyObject* classObject = PyObject_GetAttrString(moduleObject,
    "ExampleClass");
  PyObject* args = PyTuple_Pack(1, value);
  PyObject* newObject = PyObject_CallObject(classObject, args);

  Py_DECREF(args);
  Py_DECREF(classObject);
  return newObject;

}
```

Listing 18: An example constructor bridge function in C++ that could be autogenerated for any given Python data type.

Much like generating bridge functions for creating objects, the generation of bridge functions that invoke member functions can be trivial. If the member function has any parameters, they can be packed into a Python object, which can then be passed to the member function [18]. For the actual member function invocation, a special function can be used that accepts the object handle, the function name and its parameters [19]. The generator should be aware of temporary python objects, as their reference count should be decremented to avoid memory leaks.

```
extern "C"
void Foo_useValue(void* self, int value)
{
  PyObject* params = Py_BuildValue("(i)", value);
  PyObject* result = PyObject_CallMethod(
    static_cast <PyObject*> (self),
    "use_value", params
  );

  Py_DECREF(result);
  Py_DECREF(params);
}
```

Listing 19: Example bridge function for calling a member function. This can be trivial to autogenerate.

As long as everything relevant, like the type information and class hierarchy is stored in the simplified hierarchy, the bridge functions shown in listing 18 and listing 19 can be automatically generated.

5.6 Complete autogeneration tool

Different topics that are useful for autogeneration have been discussed in chapter 5; however, there has not been much discussion on how to actually put everything together. In order to build an autogeneration tool, one of the most important things is to decide whether the tool is generic or specialized for one purpose. The autogeneration tool that was built for the case company was specific to their product, which made the generation easier, as a lot of intentions were known in advance. In addition to the proprietary tool, an open source library called Autoglue used to create autogeneration tools is being developed on Github [20].

```
#include <autoglue/clang/Backend.hh>
#include <autoglue/clang/GlueGenerator.hh>
#include <autoglue/java/BindingGenerator.hh>

int main(int argc, char** argv)
{
  // Generate a simplified hierarchy using a clang backend.
  // The first command line argument points to the compilation
    database.
  ag::clang::Backend clangBackend(argv[1]);
  clangBackend.generateHierarchy();

  // Export everything from the autoglue namespace.
  clangBackend.getRoot().resolve("ag")->useAll();

  // Export bridge functions for C++.
  ag::clang::GlueGenerator glueGen(clangBackend);
  glueGen.generateBindings();

  // Export bindings for Java.
  ag::java::BindingGenerator javaGen(clangBackend);
```

```
  javaGen.generateBindings();
}
```

Listing 20: Simple autogeneration tool made with Autoglue that exports Java language bindings for Autoglue itself.

Unlike the proprietary tool, the example tool shown in listing 20 is made using Autoglue. This simplifies the creation process, as developers can use predefined building blocks to create an autogeneration tool that fits their needs without having to do everything from the ground up. It is important to note that the example tool isn't something that can be used for anything, as it is specifically made to export Java language bindings for Autoglue itself.

5.6.1 Autogeneration workflow

Since an autogeneration tool can be implemented in any way that the developer wants, there's no single way to run the autogeneration process. Some possibilities are running the autogeneration process as a compilation step of a project, or doing it when a developer clicks a button.

Much like the autogeneration process invocation, the configuration process is defined by the developer of the autogeneration tool. To make sure that any generated code in language bindings would compile, the tool built for the case company was equipped with a list containing the names of functions to export to the language bindings. This might be a good idea when there is a possibility that an autogeneration tool doesn't know how to generate something really specific. Alternatively, if there is a guarantee that an autogeneration tool will always generate valid code, it could generate everything instead of having the user explicitly specify what to generate.

With the assumption that an autogeneration tool doesn't customize things such as types that are used in a wrapped API, some things in the resulting language bindings might seem out of place. An example is the usage of a C++ standard library type in Java, where the users of the Java language bindings would have to

deal with something like a shared pointer. In the case where an autogeneration tool just brings over an API to a foreign language without any customization, the tool has no context of the developer's intentions by default. To combat such an issue, additional configuration options could be introduced to aid the developer in customizing the output. As different configuration options could applied to the processing of a wrapped API and some to the different variations of the resulting language bindings, there can be a large amount of different options customizing the output, which is why the way of exposing such options is completely up to the autogeneration tool.

5.6.2 Autogeneration tool internals

Internally an autogeneration tool should construct a simplified hierarchy from an AST, from which language bindings can be generated. As the implementation of receiving and processing an AST differ depending on the language of the wrapped API, an autogeneration tool should be aware of the language that language bindings are being made from. This information could be explicitly passed by the user of the tool, or it could be automatically detected from the source files provided by the user.

To generate language bindings, the autogeneration tool should be aware of what languages the user wants to generate language bindings for. While this information could be provided by the user, a predetermined set of languages can be implicitly used in the case where the autogeneration tool is made for a special purpose. To implement the autogeneration of language bindings for any given language, a generator can be made for it. The generator can be fed entities from the simplified hierarchy by traversing the hierarchy like shown in listing 7.

6 Conclusions

The goal of the project was to aid the developers of the case company maintain and expand language bindings made for their product. Upon anyone modifying a part of the wrapped API that also exists within the language bindings, the

modifications should be automatically transferred over to the language bindings at the press of a button. This can be a productivity booster for the developers as they can focus more on the wrapped API instead of having to worry about the language bindings. In addition, this can benefit the users of the product as more of the wrapped API can be included in the language bindings.

To implement the project, an autogeneration tool was made to allow for easy generation of an API for any supported foreign language. The tool was integrated into the development workflow as a step that had to be manually executed before changes were pushed to a remote repository. Having developers perform this manually gives them a chance of reviewing automatically generated changes, which could happen as a result of the developer changing a part of the wrapped API that is also present in a foreign language. The project was successfully carried out, as it can autogenerate the selected parts of the wrapped API to every supported foreign language. In order to guarantee that the tool would never output code that would not compile, filtering was added to the tool so that code is only generated when it is known to be valid. While the initial purpose of the tool was to generate language bindings for a single language, the tool was designed in a way that the generation of language bindings can easily be added for new languages.

Some future possibilities include further automation of the tool execution. Instead of having the developers interact with code related to a foreign language, an automated task that generates the language bindings and runs related tests could be created. Such a task can be made to report any errors on failure, where fixes to any tests that use outdated API features can be fixed. This helps the developers focus on the actual API, as they do not have to be concerned about the language bindings or whether the tests for a foreign language pass. Doing so comes with the side effect of delayed updates to the language bindings, which means that any developer that uses the language bindings will have to wait for them to update. However, this is not a big problem, as the task could be scheduled to run whenever a new update is made to the product.

While the created tool is heavily specialized to accommodate for the needs of the

product API, this thesis has attempted to discuss language bindings and their autogeneration in a more general way. This was done to help the reader understand the concepts in a way which could help them implement their own autogeneration tool that can potentially perform generation from any language to any language. Only the generation of one-directional language bindings have been discussed, which means that a foreign language can call functions from the wrapped API. Another topic that an autogeneration tool could include is the generation of two-directional language bindings, where the wrapped API is able to call functionality that a user can define in a foreign language through the generated language bindings. Some usages of this include handling exceptions through the language bindings and overriding virtual functions in derived classes that extend classes originating from the language bindings. However, this can get quite complicated, which is why it was excluded from this study.

References

1 Stevanovic, Milan. 2014. C and C++ compiling. An engineering guide to compiling, linking, and libraries using C and C++. Apress. <https://elhacker.info/manuales/Lenguajes%20de%20Programacion/C++/Advanced%20C%20and%20C++%20Compiling%20(Apress,%202014).pdf>. Accessed on Apr. 12, 2024.

2 Matz, Michael; Hubĭcka, Jan; Jaeger, Andreas & Mitchell, Mark. 2012. System V Application Binary Interface. Version 0.99.6. <https://refspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf>. Accessed on Jan. 15, 2024.

3 Language Linkage. Cppreference 2023. Online. <https://en.cppreference.com/w/cpp/language/language_linkage>. Accessed on Dec. 6, 2023.

4 Java Native Interface Specification - Introduction 2024. <https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/intro.html>. Accessed on Feb. 21, 2024.

5 Extending Python with C or C++. Python documentation 2024. Version 3. <https://docs.python.org/3/extending/extending.html>. Accessed on Feb. 21, 2024.

6 Taivalsaari, Antero. Sept. 1996. “On the notion of inheritance”. In: ACM Comput. Surv. 28.3, pp. 438–479. <https://doi.org/10.1145/243439.243441>. Accessed on Mar. 28, 2024.

7 Ferres, Leo. 2010. Memory management in C: The heap and the stack. Online. <https://cs.gmu.edu/~zduric/cs262/Slides/teoX.pdf>. Accessed on Feb. 20, 2024.

8 Bjerreskov, Rasmus. 2023. “Developing an Intra-program Messaging Utility in C++”. Bachelor’s Thesis. <https://www.theseus.fi/handle/10024/813474>. Accessed on Feb. 18, 2024.

9 Aghayev, Rauf. 2023. Exploring the Different Kinds of Garbage Collectors in Java. Online. <https://medium.com/@agayevrauf/exploring-the-different-kinds-of-garbage-collectors-in-java-7e1f10017c6a>. Accessed on Jan. 15, 2024.

10 Clang Front Page 2023. Online. <https://clang.llvm.org>. Accessed on Nov. 10, 2023.

11 Choosing the Right Interface for Your Application - Clang 2024. Online. <https://clang.llvm.org/docs/Tooling.html>. Accessed on Mar. 5, 2024.

12 JSON Compilation Database Format Specification - Clang 2024. Online. <https://clang.llvm.org/docs/JSONCompilationDatabase.html>. Accessed on Mar. 5, 2024.

13 Libclang documentation: clang_getCursorSemanticParent() 2024. <https://clang.llvm.org/doxygen/group__CINDEX__CURSOR__MANIP.html#gabc327b200d46781cf30cb84d4af3c877>. Accessed on Jan. 18, 2024.

14 Source File Inclusion. Cppreference 2024. Online. <https://en.cppreference.com/w/cpp/preprocessor/include>. Accessed on Mar. 7, 2024.

15 Type. Cppreference 2024. Online. <https://en.cppreference.com/w/cpp/language/type>. Accessed on Mar. 7, 2024.

16 AST - Abstract Syntax Tree. Python documentation. 2024. Version 3. <https://docs.python.org/3/library/ast.html>. Accessed on Jan. 18, 2024.

17 PEP 484 - Type Hints 2024. <https://peps.python.org/pep-0484/#type-definition-syntax>. Accessed on Mar. 8, 2024.

18 Py_BuildValue - Call Protocol. Python documentation. 2024. <https://docs.python.org/3/c-api/arg.html#c.Py_BuildValue>. Accessed on Mar. 19, 2024.

19 Py_CallMethod - Call Protocol. Python documentation. 2024. <https://docs.python.org/3/c-api/call.html#c.PyObject_CallMethod>. Accessed on Mar. 19, 2024.

20 Autoglue Github Page 2024. Online. <https://github.com/roopereku/autoglue>. Accessed on Apr. 11, 2024.