C++ Support in Envision

Bachelor Thesis

Lukas Vogel

Supervisors: Dimitar Asenov, Prof. Dr. Peter Müller
Chair of Programming Methodology
ETH Zürich

September 3, 2013

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

Envision is a programming environment, which uses textual and visual elements to represent source code. It aims to support various object-oriented languages. Thanks to its plug-in based structure Envision can easily be extended.

In this thesis we extend Envision with a plug-in to import C++ code. For this we use Clang and translate from Clang's AST to Envision's model. This includes mapping the text based modularity system of C++ and the preprocessor constructs found in C++ to Envision's tree based model. The plug-in we present is able to import a variety of C++ code constructs with limited support for preprocessor constructs. It is a first step towards developing Envision using Envision itself.

We evaluate the plug-in in terms of features as well as in terms of performance. In addition to that we provide guidelines on how to improve this plug-in in respect to its features and its performance for future extensions.

Contents

1 Introduction

Envision is a visual programming environment that represents source code as a combination of visual and textual components. It aims to be a programming environment for various object-oriented languages. Envision is written in C++ by means of the Qt framework. The development of Envision evolves mainly in the context of Dimitar Asenov's PHD studies [5] and was started during his master thesis [3].

In this project we develop an Envision plug-in to import C++ code. The plug-in translates from Clang's C++ AST to Envision's model. The report provides details about the implementation process as well as an evaluation and guidelines for future enhancements.

In this section we present the motivation behind this project, give a short introduction to Clang, and finally list some challenges we faced. Section 2 will discuss the representation of code in Clang and in Envision. The design of the plug-in and how to use it, is explained in section 3. We evaluate the plug-in in section 4. In section 5 we show how the plug-in can be further improved.

1.1 Motivation

Envision currently has only some support to import Java code. However Envision aims to support many object-oriented languages. The choice to support C++ in particular has various reasons. For one C++ is a very commonly used language in many different fields. As Envision is written in C++ we have a high interest on importing Envision's code. This plug-in will provide the first step to develop Envision inside itself. That will help to improve the quality of Envision in general because it will be used day to day as opposed to now where it is used only for testing of newly implemented features.

This thesis therefore concentrates on adding support for C++ in Envision. The goal is to import C++ code which is semantically comparable to what is currently supported with Java code. This ranges from simple things such as classes and methods to generic code with templates.

1.2 Clang

To import C++ code the project will rely on Clang [7]. Clang is an open source front-end to the LLVM compiler. It is very mature and is used in industry. In the latest version (3.3) Clang supports the full ISO C++11 standard.

Clang has a great community that is very helpful. On Clang's mailing list as well as on the LLVM irc-channel the people were very nice and helpful for questions we had about Clang's AST and its interface. The community also built several tools, either based on Clang or to help using Clang. For example we use the Bear tool [18] which can create compilation database files, which are needed for Clang based tools.

1.3 Challenges

Importing C++ code to Envision will raise several challenges we have to tackle.

We have to find ways to handle the C++ modularity system which is based on header files and preprocessor directives. Header files should not always be translated the same way. For system header files we are only interested that we need them (i.e. that there was an `#include` statement in the source code) but do not want to see their content, as you normally do not need to edit them. On the other hand we should translate header files which belong to the project, that the plug-in is importing, to Envision's model.

This modularity system introduces many inherently textual constructs such as forward declarations. Not all of those constructs are needed in Envision's model and we have to find solutions on how to deal with such constructs.

Clang works in translation units, that means each source file is separately translated. In Envision we only have one model for a whole project. This means we have to merge header files which were included in multiple translation units.

Clang's AST contains a lot of nodes which are only needed for compilation. Such nodes have no counterpart in the written source. In Envision we only want to edit the code as it is written in the source file. Thus we have to consider only those nodes which represent some written code.

As Envision's current model is mainly designed with Java in mind we have to adapt the model to support C++ features alongside Java constructs.

2 Envision's model and Clang's AST

The abstract syntax tree of Clang and the model of Envision are one core part to understand to design a plug-in which can translate from Clang's AST to Envision's model. This section familiarizes the reader with those 2 representations and presents an overview of differences, which are important for our project.

2.1 Overview

In general both representations have four kind of nodes:

Declarations are nodes which define a unique identifier such as classes, methods and similar top-level constructs. A body of a declaration may contain several sub-declarations, statements and expressions.

Statements are used to model code which may alter the application execution flow. Examples include if-then-else statements, all kinds of loops but also return statements.

Expressions contain all the calculations of a program. One popular expression node is the binary operation.

Types model the type of other nodes.

While Envision makes a clear distinction between those four kinds Clang developers took another design decision. In Clang every expression is also a statement. This is not very usual for an AST and even the designer seems to be unhappy with the fact that an expression is a statement as one of the developer states [16]. In the context of this thesis this difference is rather insignificant.

Envision's model only has one single node for definition and declaration of declarations, whereas in Clang's AST a definition might be in a different node than the corresponding declaration.

In addition to those four kinds both representations also have some nodes, which do not fit into those categories. An example of such a node is the node for member initializers.

In Clang's AST most nodes provide a reference to the source location from where this node was created. Such a location can be resolved to a file name and a line number, which is helpful for us to report unsupported features in our plug-in.

2.2 Differences between Clang's AST and Envision's model

In this section we present differences between the two representations. First we discuss the completely different nature of Envision's model and the textual nature of C++. Then we present some interesting differences in certain nodes.

2.2.1 Textual versus tree based nature

C++ has an inherently textual character and has some features which are difficult to represent in a structured model. The modularity system of C++ is based on header files and partitioned declarations and definitions.

On the other hand we have Envision's model which is a highly structured tree model. In Envision's model a whole project is represented in one tree. Declarations and definitions are merged together. In this thesis we refer to Envision's model as a united concise model.

Features like conditional compilation and macros are easy to represent in a textual approach but it is not clear how to translate this in a tree based model. For compilation such constructs can be expanded but in the translated Envision model we still want to have the complete information about macros and conditional compilation. This would be needed to be able to develop C++ code in Envision.

2.2.2 Switch statements

In Clang switch statements are modeled very similarly to while loops. This means a switch statement has a condition variable or expression and a body. The body is a list of statements. To model cases both representations provide a case statement. In Clang's AST the case statement contains just one child statement. Whereas in Envision's model a case statement contains every statement that belongs to this case, this means every statement before the next case statement.

This structured approach of Envision provides an easy to overview AST, because a switch statement contains, except from the statements before the first case, just a list of case statements. While in Clang there is just a list of statements, which can be confusing.

This difference affects the translation, and is considered in our implementation of the translation.

2.2.3 Lambda expressions

Lambda expressions are interesting because the different approaches for different use cases can be seen. While Envision's model only need to provide a model to interact and visually represent, Clang's AST has to be suited to be compiled. Therefore lambda expressions are implemented quite differently. For Envision's purposes it is enough to have a single node representing a lambda expression. In Clang however the lambda expression has a corresponding class and method node. The arguments and the body of the lambda are stored in the method node. Those additional nodes for the class and the method are later used for compilation. To get information about the parameters of the lambda expression we use the method node.

2.2.4 Types

Envision has two kinds of types: First there are type expressions, which are used for interaction and visualization. Second there are types, that are used for actual type resolution. All expressions including type expressions provide a `type()` method, which provides the actual type. In both Clang's AST and Envision's model there are wrapping classes for qualified types. Clang however only has types and no type expressions because there is no need to interact with them like in Envision.

3 Plug-in design and implementation

In this chapter we explain the design and the implementation of the C++ import plug-in.

For our design we first had to decide on what interface method we use to interface with Clang. Section 3.1 provides an overview of the different interface methods of Clang and explains our selection. Our choice to use the LibTooling interface causes some constraints to our design. The setup of the classes which are specifically to interface with Clang is explained in section 3.2. In section 3.3 we explain how the plug-in is structured and provide information about the classes. Later we present the managing classes and the visitors. We discuss how the plug-in deals with unsupported code in section 3.6 and how to use it in section 3.7.

3.1 Clang interface

Clang offers various possibilities to interface with it. Following is a list of possible interface methods, and explanations why a certain method is used or why not.

LibClang is a stable C interface to Clang. It aims to stay backwards compatible and provides high-level abstractions from Clang's AST. The downside of this method is, that it does not provide full control over the AST. Furthermore it is more natural to use a C++ interface in a C++ environment.

Clang Plugins are a way to run additional action's on the AST and are easy to integrate with a build environment. However in this project the goal is to create a plug-in in Envision and we only need to read the Clang AST, due to this we do not use this method.

LibTooling [10] is a C++ interface which is to be used to write standalone tools. As Envision is already written in C++ it makes sense to also write the C++ import plug-in in this language. The only downside this method has for our purpose is that the code may need to be adapted after a change to Clang's AST. As this mostly happens due to a language change, it is important to look into this anyway as one may need to adapt Envision's model to support such changes.

Considering these points our choice is to use the LibTooling interface.

3.2 Classes specific to Clang

This section presents the classes which our plug-in needs to interface with Clang. A tool built with LibTooling runs a `FrontendAction` over some code [10]. The `FrontendAction` is the base interface for various front-end actions, those are actions which are performed in the front-end of the compiler. The front-end of a compiler deals with parsing, syntax analysis, and generation of the internal representation, the back-end is to optimize and generate code. For actions which only use the preprocessor there is the `PreprocessorFrontendAction`, this action can for example be used to rewrite certain preprocessor statements. There is also an interface for merging ASTs the `ASTMergeAction`. We use the `ASTFrontendAction` which is an abstract interface for actions which only work on the AST. Other actions can be found in Clang's documentation [8].

In figure 3.1 you can see how the plug-in uses the Clang specific classes. The `CppImportManager` creates a Clang tool. To create the tool the class will first search the provided directory for source files and for a compilation database because they need to be provided for the tool. The tool features a run method which takes a `FrontendActionFactory` as a parameter. The tool uses this factory, in our case the `ClangFrontendActionFactory` to create a `FrontendAction` for each translation unit.

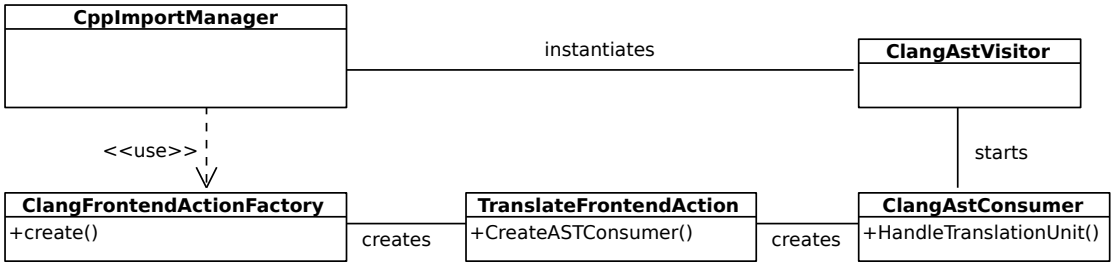

Figure 3.1: Clang specific classes

The `ClangFrontendActionFactory` class implements the `FrontendActionFactory` interface. It has to provide a `create()` method which is called for every translation unit the tool is run on. This method will then create the `TranslateFrontendAction`. As we want the visitor and the logging class to persist over all translation units the factory will provide those to the created front-end actions.

The `TranslateFrontendAction` is our implementation of the `ASTFrontendAction` interface. Therefore the class has to provide a method `CreateASTConsumer()` to create an AST consumer. This method will also get called for each translation unit. In our implementation the method creates an AST consumer and adapts the compiler instance in the persisting classes. A compiler instance provides access to the source manager of Clang which is needed for logging purposes in our tool.

The `ClangAstConsumer`, which implements the `ASTConsumer` interface, is in its current state just a wrapper to start the AST visitor. However it will be useful for extending the plug-in with more functionality such as comment handling which we discuss section 5. Therefore the class has a pointer to the `CppImportLogger` class which is used for logging.

3.3 Plug-in design

This section discusses the design of the plug-in apart from the classes which are specific to Clang.

The main interface of this plug-in for users and other plug-ins is the `CppImportManager`. It provides methods for testing and setting the source path. After the source path is set the model can be created with the `createModel()` method. The detailed usage of the plug-in is discussed in section 3.7.

For the translation we use several visitors which interact with each other. All of the visitors use the utilities and logger classes. The visitors are explained in detail in section 3.5.

The manager classes are used to find duplicates of nodes in Clang's AST and to merge them. They are explained in section 3.4.

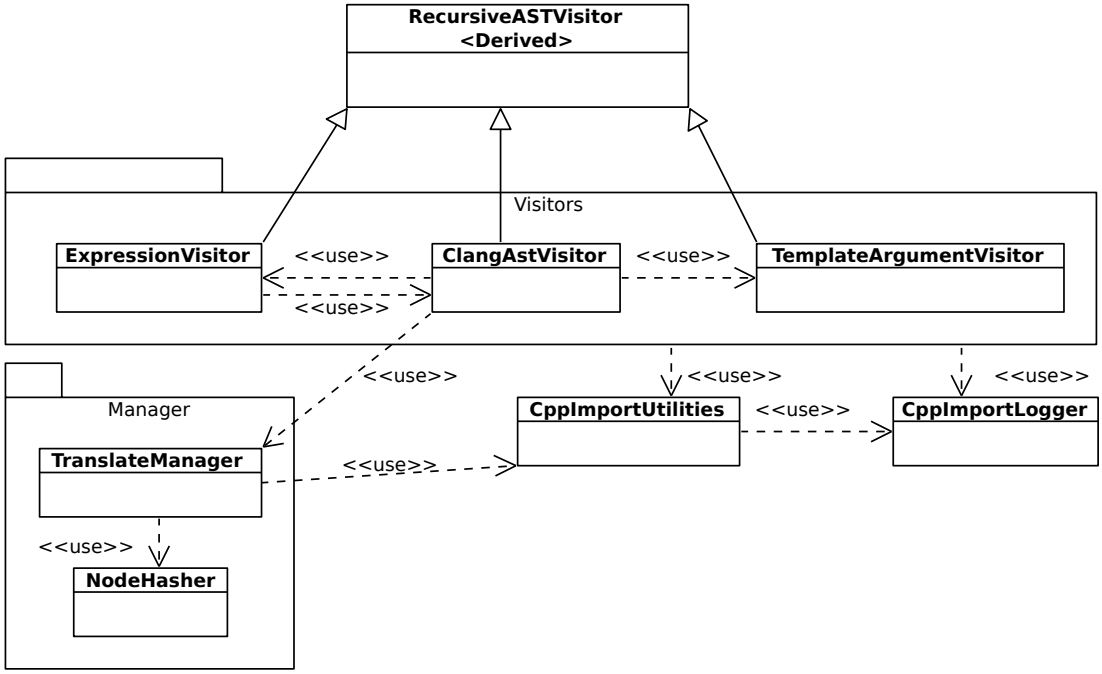

Figure 3.2: Plug-in design

The `CppImportUtilities` class is used for translation of simple things like operators. Therefore it contains a lot of boiler-plate code with switch case constructs.

The `CppImportLogger` class is used to log unsupported features and helps with error analysis. Its functionality is described in section 3.6.

3.4 Manager classes

As mentioned earlier Clang works in translation units, that means each file we want to translate gets processed separately. Therefore the plug-in will translate some header files multiple times and we have to manage duplicates of declarations. We also have to merge declarations and

definitions because, as we have explained earlier (section 2.1), Envision has a united concise model. For this we use the `TranslateManager` class, which keeps track on whether a certain declaration has already been visited. To store the data we use several maps which use a unique string hash of the declaration as a key and store a pointer to the corresponding node in the resulting model.

To create this unique string representation the `NodeHasher` class is used. This class has the sole purpose of generating unique strings for Clang nodes. To achieve this it uses a combination of the node name, the node's parent, the node's type, and the node's template arguments. Thanks to this we can merge declarations and definitions together and get everything in Envision's representation.

3.5 AST visitors

The core of the plug-in consists of 3 visitors. We have different visitors for different purposes, they all visit a certain kind of nodes and translate those nodes. To do this translation, the visitors use a custom strategy to visit the children. All visitors inherit directly from Clang's `RecursiveASTVisitor`, which is explained in the following subsection.

3.5.1 Recursive AST visitor

The `RecursiveASTVisitor` is the parent class of all our visitors, it is from Clang's library. This visitor uses the curiously recurring template pattern [13].

The visitor does a depth-first order traversal of the whole Clang AST, and provides 3 groups of methods. The first group is the `Traverse` method group, those methods just dispatch to the `Traverse` method which matches the dynamic type of a node. From there the `WalkUpFrom` method is called which goes up the class hierarchy until a top-level node, e.g. a declaration, is reached. While 'walking up' the method calls a user overridden `Visit` method – if there is one. Those methods are in a hierarchy, in which the `Traverse` methods are highest, the `WalkUpFrom` are in the middle, and the `Visit` methods are lowest. A method can only call methods from the same, or lower levels.

The visitor as provided dispatches to the correct user defined method, this behavior is best explained with an example. If `TraverseStmt()` is called on a `LabelStmt`, the `TraverseStmt` will dispatch the call to `TraverseLabelStmt()`. If there is no such method in the derived class, the visitor will instead call the corresponding `WalkUpFrom` method, in our case the `WalkUpFromLabelStmt()`. The `WalkUpFrom` method first calls the same method on the parent in the node hierarchy, in our case `WalkUpFromStmt()`, and then the corresponding `Visit` method, here `VisitLabelStmt()`. This results in a call chain up from the dynamic type of the node until the top-level type like statement or declaration, and then from there again down the hierarchy with the `Visit` methods.

Once understood this system provides several advantages. We use `Traverse` methods for almost all nodes, this is the easiest way to define a custom visiting order and is also fast to dispatch, as there is only one method call in between. For example for an if statement we use the `TraverseIfStmt()` method. For nodes where the common ancestor node provides enough information for our purposes we use the `WalkUpFrom` methods. This saves us to imple-

ment identical methods, or even a separate dispatch to the correct method. For type aliases there exists the new C++11 form (`using Alias = type`) and the previous form (`typedef type Alias`), both forms have a separate Clang node but a common ancestor, therefore we use the `WalkUpFromTypedefNameDecl()` method which deals with both nodes.

Using only those 2 method groups has another advantage, we can use the `Visit` methods to find unsupported nodes. In our derived visitors, we define the `Visit` methods for top-level nodes, such as statements and declarations. Whenever such a method is called, we know that this node is not supported, since otherwise our defined `Traverse` or `WalkUpFrom` method would have been called.

Now that we know the general structure of the visitors, we can go into the details of our visitors.

3.5.2 Declaration and Statement visitor

The `ClangAstVisitor` is the main visitor, it instantiates the other 2 visitors, which are used to help. It is instantiated with a pointer to an Envision root node, i.e. a project to which it will add all the translated nodes.

The visitor uses two stacks that help the translation process. One stack is for declarations and statement item lists and is used for visiting bodies. For example to translate a method the visitor pushes the statement item list on the stack and calls `TraverseStmt()` to traverse the body, in this way each node which is in the body, gets added to this list. For classes we push the class node on the stack and then visit all children of the class. When we visit a method declaration we can check what is on top of the stack and in the case of a class just add the method to the method list of the class. Because in Envision the children of declarations are structured, e.g. a class has lists for methods, fields and more, it easier to append a node to the correct list directly when visiting it. When we would do it the other way around, and push the children on the stack, we would also first have to check what node is on the stack, that would include a lot of casts.

The expression stack on the other hand, works in the opposite direction. For expressions the caller calls the `TraverseStmt()` method on the expression, and then pulls the translated expression from the stack. Whenever we visit a node which has expressions as children we know exactly where those expressions belong to, for example when visiting a loop statement we can visit the condition and then pull the last expression from the stack and set this as the condition. This way we have no overhead with casting or determining where to put the expression.

We implemented the `TraverseStmt()` method such that it dispatches to the `TraverseStmt()` method of the expression visitor if the argument is an expression node.

3.5.3 Expression visitor

The `ExpressionVisitor` translates all kinds of expressions. It can be used by calling the `TraverseStmt()` method, which automatically dispatches to the correct method. While this method also allows statements to be passed in, the caller has to make sure that only expressions are passed. The visitor has a stack of the last translated nodes, and provides a method for the client to get the last translated node. If the node could not be translated, the method will

return an `ErrorExpression`. We use this stack also for intermediate results, for example for overloaded operator calls we first translate all arguments and then, dependent on the operator, create the correct expression using the translated arguments.

While the visitor is designed for the sole purpose of visiting expressions, it still needs the `ClangAstVisitor` because of the way lambdas are modelled in Clang's AST. As we have seen in section 2.2.3 lambda nodes have certain information in class and method declarations. We therefore use the `ClangAstVisitor` to visit the body of the lambda.

3.5.4 Template argument visitor

This visitor is very simple and only translates template argument declarations. Unlike the expression visitor, only a field is used to store previously translated nodes, because we only translate one argument at the time. The class provides a method which takes a declaration as an argument and directly returns the translated node. If the node could not be translated the method returns a `FormalTypeArgument` which has "#ERROR" as name, this should make the error visible in Envision. Additionally an error message is printed in the console.

While it would be possible to include the methods of this visitor in the `ClangAstVisitor` we decided that separating this out provides a nicer structure of the plug-in. This is possible because templated nodes in Clang's AST provide an iterator over template arguments. We explicitly loop through template arguments and translate them with this visitor.

3.6 Handling of unsupported code

C++ has a very large amount of features, we only support a subset thereof. For the user we have to provide feedback when something is not supported. Therefore we implemented the `CppImportLogger` class which keeps tracks of unsupported features and prints warnings for the user. Feedback is structured such that the user knows exactly where the problem is i.e. we provide a file name and a line number of the problem. The feedback is only given for C++ code and does not warn about preprocessor statements or comments. As an example we try to import the code listed in listing 3.1. If one just looks at the output (figure 3.3), it seems on the first glance correct, however the label is missing.

```
1 int main() {
2     label: std::cout << "print";
3     return 0;
4 }
```

Listing 3.1: Unsupported code with a label

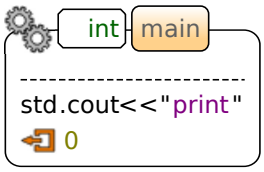

Figure 3.3: Translated sample

The console output reflects this observation and prints the warning as in listing 3.2. It is obvious that on line 2 there is a LabelStmt which is not supported. We implemented a set of error codes that are appended in the end of the not supported node type. Those will be explained in the following subsection.

```
1 ERR/WARN:  In class : ClangAstVisitor
2            reason : Not supported
3            in stmt class node : LabelStmt_NS
4            in file : test.cpp
5            on line : 2
```

Listing 3.2: Error note for the label statement

3.6.1 Error codes

We found that problems while importing code often have similar reasons and therefore created 4 groups of errors:

Not supported (_NS): Nodes which the plug-in at its current state simply does not support. For expressions we create an error expression which is visible in Envision and also output the corresponding warning. (Example: user defined literals) For declarations and statements which are not supported we only output a warning. (Example: labels)

No parent (_NP): Nodes for which the plug-in finds the Clang parent node, but this node is not yet translated. This would happen if a class method is declared before the declaration of the class. During development we had this error for variables which were from an implicit instantiated template class, which is now checked.

Insert problem (_IP): This is the case when either the imported C++ code is invalid, or it is not supported by Envision. Such an error happens for example when the visitor visits a statement, but the current node on the stack is not a statement item list. This happens if we forget to push the statement item list previously for example when visiting a method body. The visitor therefore does not know where to put the statement. For invalid C++ code the Clang parser already prints the Error and does not create a corresponding node in the AST, therefore whenever this Error occurs it points to a bug in the implementation of the plug-in.

Other (_O): This is for problems which do not fit into the other groups. Those cases provide a detailed message of the reason.

Thanks to the suffixes, which are presented in the brackets after the name, it is easy to get an overview of which problems occurred in the statistics output. The statistics table is presented in the end and contains the node name with the suffix, plus the number of times this error happened. An example of such statistic output is shown in listing 3.3.

```
=Statistics of warnings and errors==

General errors
UnaryOperator_NS:              50

Types not supported by envision
TypeOfExpr:                    25
unsigned char:                  9

Unary operations not supported
UO_Extension:                  50

Storage class specifiers not supported
SC_Extern:                      2

===========Statistics End===========
```

Listing 3.3: Sample statistics output

3.7 Using the plug-in

This section discusses the usage of the new plug-in. As for using the plug-in you will, in most cases, need a compilation database file. We explain why this file is needed in section 3.7.1. We present how to test the plug-in with test cases in section 3.7.2 and in section 3.7.3 we explain the steps needed to import C++ projects.

3.7.1 Compilation databases

Compiling C++ often needs additional information passed in as command line arguments. Those arguments can be used to specify the version of the standard (for example `-std=c++0x` to specify the C++11 standard) or to include libraries with the `-I` switch. There are even more such switches and they may differ from source file to source file. It is therefore important that the plug-in, i.e. the Clang interface, is aware of such arguments. Clang tools take a compilation database file, that conforms to the format described in the Clang documentation [9]. Those files will contain the whole command which would be used to compile a certain source file and also the complete path to the source file. Providing such a compilation database file is the easiest way to specify compilation arguments for the plug-in.

3.7.2 Testing

When implementing support for new nodes, or when checking the behaviour of the plug-in for a certain construct it is often desirable to have a small test case and an easy way to provide this case. Therefore we provide the `setupTest()` method in the `CppImportManager`. This method will import the test selected with the `testSelector` file.

To create a test, just create a directory (inside the test directory of the plug-in) with the name of the test and inside it a file called 'test.cpp'. In the `testSelector` just write the name of the directory, and make sure there is no previous line without a '#' character. Then simply run Envision with the `--test cppimport` parameter.

We generate a generic compilation database file for the test cases and use `/usr/bin/clang++ -std=c++0x -Irelative -I/usr/lib/clang/3.3/include/ -c -o test.o test.cpp` as compile command. If those arguments do not fulfill your needs you have to use a project as described in the next section.

3.7.3 Importing C++ projects

For larger projects importing is a bit more complex, but luckily most steps are automated and do not need much interaction. As opposed to the test cases, you have to provide a compilation database file for projects. We will explain how you can generate them automatically.

Before creating the compilation database file you may have to adapt a few settings in your project.

- You have to set the compiler to `clang++`.
- You have to include the include directory of Clang, on Linux systems this is typically

  `/usr/lib/clang/3.3/include/`

  This is needed to find Clang's system headers such as `stdarg.h`
- If you use C++11 features such as list initializers and libraries which check for the compiler version such as Qt 4 you have to fake a more up-to-date gcc version since Clang currently masquerades itself as gcc version 4.2.1. This can be achieved with first undefining the version with `-U__GNUC_MINOR__` and then define it to the latest version i.e. `-D__GNUC_MINOR__=8`.

Now that the set up is done you can proceed with generating the compile commands file. If you use one of the following build systems you can just follow the described steps to generate such a file:

CMake: As of version 2.8.5 CMake supports generating compilation databases for UNIX Makefile builds with the option `CMAKE_EXPORT_COMPILE_COMMANDS` [9].

Qt qmake project: For qmake project files just run `qmake` on them which generates a Makefile and then proceed with the steps explained in the make description.

make Makefile: For make files you can use the Bear tool [18]. As soon as you have installed the tool you can call `bear -- make`. This will compile your project and generate the `compile_commands.json` file. You have to make sure that your project is not yet compiled otherwise the database will be incomplete or empty.

After you have your compilation database file you are ready to import the code. We provide a way for users to easily import their project for testing purposes and also an API for other plug-ins:

Users Just specify the path in the `testSelector` file (this file is in the test folder of the plug-in) begin the line with `path:` and then write the path. After having set this up you can run Envision with the `--test cppimport` parameter. If you have a project with sub-projects you can specify the path in `testSelector` file with the `spath:` prefix. For this you have to have a compile commands file for each sub-project as described below.

API For other plug-ins we provide a method to import code. You can interface with the `CppImportManager` class. First you have to set the import path with the `setImportPath()` method, you have to pass the path of the project you want to import as argument. If you have a project with sub-projects set the second parameter to true, and make sure you provide a compile commands file for each sub-project as described below. After you have set the path you have to call the `createModel()` method which returns the root of the translated project.

If you want to use the tool on a project with sub-projects it is best to create the compile commands file for each sub project this can be achieved with a small script as in listing 3.4.

```
1 #!/bin/bash
2
3 for dir in ./*/
4 do
5   (cd $dir && make clean && bear -- make)
6   echo "Processed $dir"
7 done
```

Listing 3.4: Script for sub projects

This special treating for projects with sub-projects is needed because of an issue in Clang's tooling interface. If we would just use the top-level `Makefile` Clang would use invariant include paths. In particular we found that Clang replaced `-I/src` with the complete path while processing the first sub-project. Therefore the include path is wrong for the other sub-projects and causes errors because header files can not be found. We reported this issue on the Clang mailing list but did not yet get any feedback on this.

4 Evaluation

Now that we have seen how the plug-in is built, we present an overview of supported features in the implementation. As one of our main goals was to import Envision's code, we analysed how well the plug-in performs in doing that. To check general performance we ran our tool on other C++ software and report the results in section 4.2.

4.1 C++ support status

The initial goal of this thesis was to support C++ code which is semantically comparable to what the Java import plug-in supports. We outreached this goal and support even more constructs. In the following subsections we present the status of some interesting features and constructs.

4.1.1 Feature completeness

In this section we present an overview of C++ constructs and their support status. The table contains the name of the feature, the corresponding node name in Clang if there is one, and in the comment column you can see current state of the feature. The overview concentrates on features, which are only partially or not at all supported. For some features we provide additional information in the following subsections.

Feature	**Clang Node**	**Comment**
Preprocessor		
Preprocessor	–	Minor support, see section 4.1.3
Template code		
Template code	–	Mostly supported, see section 4.1.4
Partial template specialization	ClassTemplatePartial-SpecializationDecl	Not supported
Template specialization	ClassTemplate-SpecializationDecl	Supported
Templated methods	FunctionDecl	Supported
Templated reference expressions	–	Supported including method calls

Templated friend declarations	FriendTemplateDecl	Not supported
Declarations		
Namespace declarations	NamespaceDecl	Supported including merging
Class declarations	CXXRecordDecl	Supported, also unions and structs
Method declarations	FunctionDecl	Supported, also constructors and destructors
Variable declarations	VarDecl	Supported, also static fields (modeled as VarDecl in Clang)
Field declarations	FieldDecl	Supported
Alias declarations	various	Supported
Friend declarations	FriendDecl	Mostly supported, see section 4.1.5
Access specifiers	AccessSpecDecl	Supported indirect (method & fields are queried)
Dependent using declaration which is not marked with typename	UnresolvedUsing-ValueDecl	Not supported
Statements		
Statements	Stmt	All statements used in Envision are supported
Expressions		
Lambda expressions	LambdaExpr	Supported except for capture lists
Overloaded operator calls	CXXOperatorCallExpr	Most supported, except for overloaded memory operator call (new, delete)
Literals	–	All supported, except user defined ones
C++11 noexcept expression	CXXNoexceptExpr	Not supported
Pseudo destructor	CXXPseudoDestructor-Expr	Not supported
C++11 Pack expansion expression	PackExpansionExpr	Not supported
Size of pack expression	SizeOfPackExpr	Not supported
Parenthesized list expression	ParenListExpr	Not supported, see section 4.1.6
Types		
Function type	FunctionProtoType	Supported except for argument names
Decltype	DecltypeType	Not supported
Member pointer	MemberPointerType	Not supported
Various elements		
Member initializers	CXXCtorInitializer	Partial support, see section 4.1.6

Extern keyword	–	Not supported
Labels	LabelStmt / LabelDecl	Not supported
Body of anonymous outlined functions	CapturedStmt / CapturedDecl	Not supported
Inline ASM code	ASMStmt	Not supported
Go to	GotoStmt / IndirectGoto	Not supported
Attribute applied to a statement	AttributedStmt	Not supported

4.1.2 C++11 features

The latest C++ standard brings many valuable additions to the C++ language. Some of those features are already used quite often in the code of Envision. As our goal is to import Envision we aimed to support all the needed C++11 features to achieve this. Envision's model already supported a load of features, examples include: lambda expressions, null pointer constant, initializer lists, in-class member initializers, and range based loops. Yet there were some features that needed an adaptation of the model. We added support for the auto type, which is a very commonly used feature. Furthermore we had to introduce member initializers for methods, which we designed in a way that they directly support delegating constructors.

4.1.3 Preprocessor

Preprocessor directives are a very important construct for C++ programmers. One very basic directive, the `#include` directive, is crucial for every C++ program. Thanks to Clang which provides preprocessed sources this construct is partially supported out of box. Partially supported means here that files which are included are also considered for translation, but we lose the information that they were included. This loss of information is unfit, especially in the case of system headers because those are not translated to Envision's model, and it should be considered to translate includes to a corresponding Envision node in the future. The `#pragma once` directive will guarantee that a header file is included only once for each translation unit.

For macros and conditional compilation we only provide limited support. As the tool runs on preprocessed sources we only get the macros expanded, for the defined values as given. This is enough to test the importing of C++ code, but needs to be extended to really be able to develop C++ code in Envision. We outline an approach to improve preprocessor support in section 5.2.

4.1.4 Templates

For templates the initial goal was to support constructs similar to Java, this means templates used like Java generics. The plug-in supports normal template classes, template methods, as well as template specialization.

Support for the recently introduced variadic templates [2][6, pp. 327–329] is yet missing. Supporting variadic templates would need changes to Envision's model in various nodes, some of them are stated in the following.

First of all we would need a way to model variadic templates in the `FormalTypeArgument` node of Envision. Then we would need a node to represent the ellipsis operator (...). This ellipsis operator will have to contain a child expression, which will be used as the argument. For example for a method which takes a variadic template as parameter i.e. when the call looks like `func(args...);` the parameter in Envision's model would be an ellipsis operator with a reference-expression, which refers to args, as the operator argument.

4.1.5 Friend methods

While support for friend classes is easy, because they can only be declared, there are more issues for friend methods. Friend methods can be defined inside a class, but even if done so they are still global [6, p. 243]. Therefore we would have to split the method into declaration and definition, because Clang models it as one single node.

While this is feasible, by propagating the definition to the correct place, Envision in its current state still lacks the possibility to refer to this method. Presently we are using a method call expression to refer to the method, which is uniquely used for debugging and displaying purposes. This helps as a place holder in the implementation, as soon as Envision supports referring to a method that can easily be adapted in the current implementation.

4.1.6 Member initializers

We provide support for most member initializers but there are some cases which do not work.

If we have a call to a super constructor from a templated class there will be a `ParenListExpr` as initializer expression which is not supported by our plug-in. An example of such a case is shown in listing 4.1.

```
1  class Pair {
2    int a,b;
3  public:
4    Pair(int f,int s){
5      a=f;
6      b=s;
7    }
8  };
9
10 template <class T> class A : public Pair {
11   A(int f, int s) : Pair(f,s){}
12 };
```

Listing 4.1: Member initializer which does not work

The interesting thing is that if the class `A` is not templated the Clang AST is different and supported. Below you find the AST dumps of the sample with and without template argument (only the member initializer is shown).

```
1     |-CXXCtorInitializer 'class Pair'
2     | |-CXXConstructExpr 0x1aeb940 <col:21, col:29> 'class Pair' 'void (int, int)'
3     | | |-ImplicitCastExpr 0x1aeb910 <col:26> 'int' <LValueToRValue>
4     | | | `-DeclRefExpr 0x1aeb4d8 <col:26> 'int' lvalue ParmVar 0x1aeb2c0 'f' 'int'
5     | | `-ImplicitCastExpr 0x1aeb928 <col:28> 'int' <LValueToRValue>
6     | |   `-DeclRefExpr 0x1aeb500 <col:28> 'int' lvalue ParmVar 0x1aeb330 's' 'int'
```

Listing 4.2: Working AST

```
1       |-CXXCtorInitializer 'class Pair'
2       | |-ParenListExpr 0x2bff7e8 <col:25, col:29> 'NULL TYPE'
3       | | |-DeclRefExpr 0x2bff798 <col:26> 'int' lvalue ParmVar 0x2bff580 'f' 'int'
4       | | `-DeclRefExpr 0x2bff7c0 <col:28> 'int' lvalue ParmVar 0x2bff5f0 's' 'int'
```

Listing 4.3: Templated sample, not working

This parenthesized list of expression is introduced whenever Clang can not determine which method will be called. In fact in this case it would be possible to determine it, but there is no implementation for this decision. You can find the discussion on the mailing list [15].

4.1.7 Implicit nodes

Clang creates an AST which is ready to be compiled. Therefore the tree contains more information than what is written in the source code. For Envision we just want the parts written in the source code. While many implicit nodes can be ignored and are not visited at all, there are some cases where implicit nodes contain information relevant to our translation. In this section we present some cases where the translation adds implicit code and explain why this happens.

In figure 4.1 you can see the translated code of listing 4.4. The difference is obvious: The plug-in has introduced a `QFlags` constructor call which is undesired because this does not represent the written source code.

```
1 Modifiers get() const {
2     return modifiers_;
3 }
```

Listing 4.4: Implicit code sample

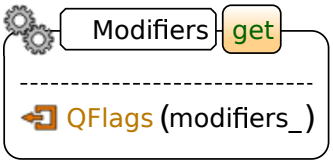

Figure 4.1: Translated sample

To find out why this happens we looked into the AST Clang generates for the code as written in listing 4.4. There we found an explicit construct-expression (i.e. `CXXConstructExpr`) node. To further investigate this we changed the code to match the translated code and compared the two ASTs. But unfortunately both ASTs looked identical and therefore this did not provide any more insight on how to deal with this problem.

However one possible solution to solve this problem is to compare the beginning and end source location of the construct expression. If those locations are identical we can conclude that this expression is not written. In that case we just have to find a strategy on how to deal with

multiple children. Other than that such an expression can be ignored and the child is to be visited.

Another problem arises for implicit conversion operator calls. In figure 4.2 the call to a conversion operator (`operator int()`) has been inserted when comparing to listing 4.5. In the AST one can discover an implicit cast expression node which has the operator call as a child. Using this implicit cast as an indicator of the implicit operator call expression might be possible. While we observe that this cast is not needed when explicitly writing the operator call, there might be cases where this assumption can be misleading. Another solution is to do the same as before and just compare the source locations.

```
1 void save(Model::PersistentStore& store)
      const
2 {
3     store.saveIntValue(modifiers_);
4 }
```

Listing 4.5: Implicit code sample

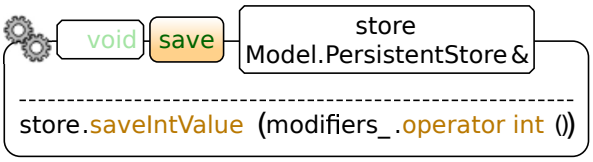

Figure 4.2: Translated sample

4.2 Performance

This section presents statistics on time and space consumption of our plug-in during practical test runs. As importing is done only once for a certain project, later we can use Envision's persistence model, performance is not a very important factor for this plug-in. The values we present are just approximations and are only to give a rough idea of time and space consumption.

The time was measured using the `QTime` class. We only measured the time from the start of the C++ import until the end. This means the time to resolve the references and to visualize the model is not included.

The memory was measured with the `top` tool [1]. We used the tool by observing the memory usage during the importing. This is not very accurate but provides a rough approximation. We only show the highest usage during the plug-in run. The plug-in frees all resources except from the created model as soon as it is finished.

The line number were determined with the CodeAnalyzer tool [20]

Test	**Total Lines**	**Code lines**	**Time used [min:sec]**	**Memory [Mb]**
Envision	85363	42691	1:55	176
cppcheck gui	23678	16440	0:27	100

We imported the cppcheck tool [17] to compare the results against those from Envision.

Memory is not a very big concern because after the plug-in is finished with creating the model, Envision uses a lot more memory than what is needed during the importing. Memory usage is low thanks to translation units, because we only have rather small Clang ASTs which are deleted after the translation unit is processed.

Time usage is rather high but at an acceptable level, when considering that importing is done only once. In section 5.4 we present how to make this plug-in work in multiple threads, which should lower the time used.

5 Future extensions

The plug-in provides support to import a variety of C++ constructs as explained in the previous section. However import functionality is far from being complete and there are still interesting things which can be added. In section 5.1 an approach on how to support the import of comments is outlined and discussed. Section 5.2 is about advanced preprocessor directives such as macros and similar constructs. Even if there is a high coverage of supported AST nodes there might be the need to support newly introduced nodes. Therefore we present how to achieve this in section 5.3. We outline the needed adaptations to make the plug-in multi threading capable in section 5.4.

5.1 Comments

Comments are a crucial feature to document and explain code. Therefore it is of high interest to deal with source code comments in the future. This section lays out approaches on how to deal with comments used for documentation, but also how to deal with explanatory comments.

Given that documentation comments, are bound to a declaration, adding support for them is very easy. Clang's framework parses comments and binds them to declarations. To get the comment of a declaration the `getASTContext()` method can be used to get the context and from the context you then get the comment. The `ASTContext` class provides several methods to get the comments reaching from raw comments to pre-parsed full comments.

While the documentation comments are quite easy to get, some more work is required to get comments, which for example explain some complicated code in a method. We outline one possible approach to deal with them. Our approach is based on source locations, because for statements Clang does not provide an AST Context or anything similar. Therefore we first have to implement a comment handler. The comment handler should use the `Clang::CommentHandler` as a base class. The only method this handler has to provide is the `HandleComment` method. In our approach this method is used to save the source ranges in a list. The source range is enough to get the comment later, as long as we are in the same translation unit.

Once we have the comment handler, we have to make sure Clang is aware of it. So we add the following line to the `CreateASTConsumer()` method in the `ClangConsumerCreator` class :

```
CI.getPreprocessor().addCommentHandler(//pointer to our handler);
```

This will register the comment handler to the preprocessor, thus the `HandleComment` method will get called whenever a comment is read. Note that the `ClangConsumerCreator` class provides a reference to the `CppImportLogger` class which could be helpful to log comment activities.

Now our comment handler will collect all locations of comments. What remains to be done is to adapt the visitor to query the comment handler, if there is a comment near the statement we are currently visiting. If we find a statement which has a comment, we can get the comment using the source manager.

During development we implemented a sample comment handler to test the behavior. The handler we had, just printed every comment it received. Because Envision has currently no support for comments we removed this handler shortly after our experiment.

5.2 Preprocessor directives

Some directives such as `#include filename` and `#pragma once` are partially supported as discussed in section 4.1.3. The source being preprocessed may cause a loss of information we are interested in. As Envision is a code editor it needs to be aware of all possible cases of a conditional compilation. However such functionality is currently missing and therefore we present a rough overview of steps that are necessary to add this functionality. This guide will only include the steps needed in the CppImport plug-in. How to model such a construct in Envision's model has yet to be determined.

The approach for preprocessor directives is, like the approach with comments, based on source locations. Clang allows to set preprocessor callbacks to easily interact with the preprocessor, without reimplementing the `Preprocessor` class. A class which implements the `PPCallbacks` interface provided by Clang [11] has to be implemented to get information on what the preprocessor does. The interface provides methods for both conditional compilation, and macros which is well suited for our desires. For the constructs of interest we can store the source location, and if needed additional information. The callback method for `#ifdef` constructs for example contains not only the source location but also the token which is tested. This token will be of interest if we want to support conditional compilation. There might be additional steps needed to always get valid source locations, this was discussed on the mailing list [14] [19].

After having the class implemented we have to register it to the preprocessor. This is done by manipulating the `ClangConsumerCreator` class. In the `CreateASTConsumer()` method, we have to add the following line:

```
CI.getPreprocessor().addPPCallbacks(//pointer to our callbacks);
```

This will add our callback class to the preprocessor callbacks.

Now that the callbacks are registered we have to adapt our visitors. The visitors have to query if on a certain source location there is a preprocessor construct which is of interest.

5.3 AST nodes

An important and often needed extension is to add support for new Clang AST nodes. This is rather simple by extending the existing visitors. We demonstrate the process of adding support for a new node with examples, but try to keep explanations as general as possible. In the example on expressions, the focus lies on user defined literals, a C++11 construct, which is not yet supported. The second example will be about declarations. First off we present initial common steps that need to be done first.

5.3.1 General steps

At the beginning a test case, which contains the new feature, has to be written. The test case should be as minimal as possible. This helps to have an overview and simplifies the debugging process. With the test case given, it is possible to try and see how the plug-in deals with it. This should already provide an error message with the node name that is not supported.

If this is not the case, or more detailed information is needed, the AST dump will help. This can either be achieved programmatically inside the visitor by calling `dump()` on a node or otherwise by calling `clang -cc1 -std=c++11 -undef -ast-dump test.cpp`. The first method makes it possible to choose very precisely which node to print, whereas the second method prints the whole AST, but is easier to do since there is no need to recompile anything. In both cases it is advised to have a short and concise test cases because the AST quickly gets large which makes it more difficult to find a specific location in it.

With the node name we can find the rich documentation of a node on the llvm website, a search on Google with the node name should directly lead to it.

5.3.2 Expressions

In our example we use the test case as listed in Listing 5.1. In its current state the plug-in translates this sample into the representation showed in Figure 5.1. From the console output, which is shown in listing 5.2, we can see that the missing node is a `UserDefinedLiteral`.

```
1 int operator"" _toInt(long double n) {
2   return int(n);
3 }
4
5 int main(){
6   int m = 5.0_toInt;
7 }
```

Listing 5.1: User defined literals, test case

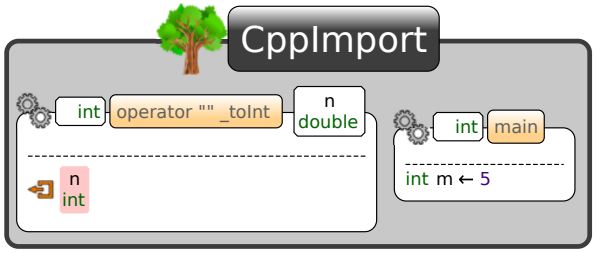

Figure 5.1: Translated test case

```
1 ERR/WARN:   In class : ExpressionVisitor
2     reason : Not supported
3     in stmt class node : UserDefinedLiteral_NS
4     in file : test.cpp
5     on line : 6
```

Listing 5.2: Error when importing a user defined literal

As we have to visit children in our specified order, our method will be a Traverse method, and we can call Traverse for the children (remember the different methods as explained in section 3.5).

The user defined literal has two parts which we have to consider: first we have the arguments and second we have the literal suffix. The arguments can be traversed by the expression visitor and the suffix we receive with the `getUDSuffix()` method.

Applying all this, we get the method as shown in Listing 5.3. This method is quite simplified for presentation purposes and would need error checking in the real version. Since Envision has currently no support for user defined literals we inserted comments on line 10 and 11 to specify which additional steps would be needed.

```
1  bool TraverseUserDefinedLiteral(Clang::UserDefinedLiteral* literal)
2  {
3      QString suffixName = QString(literal->getUDSuffix()->getNameStart());
4      for(auto argIt = literal->arg_begin(); argIt!=literal->arg_end(); ++argIt)
5      {
6          TraverseStmt(*argIt);
7          OOModel::Expression* arg = ooExprStack_.pop();
8          // do something with arg
9      }
10     // create the translated node.
11     // push the created node on the expression stack
12     return true;
13 }
```

Listing 5.3: Pseudo code of the new method

5.3.3 Declarations

As for declarations we also have to check for duplicates, the process contains some more steps than the process for expressions. The example, which we use for this explanation is a type alias with type arguments. We use an example from Envision's code, we modified the code from the `Reflect.h` file slightly, to the code in listing 5.4. If we run the plug-in on this code we receive the Envision model as shown in figure 5.2. There you can see, that the translated version of the type alias contains no type arguments which is wrong.

```
1  namespace Core {
2
3  template <class Base>
4  class Reflect : public Base
5  {
6      protected:
7          using Super = Reflect<Base>;
8  };
9  }
10
11 template <class Base> using Super =
       Core::Reflect<Base>;
```

Listing 5.4: Type alias with template argument

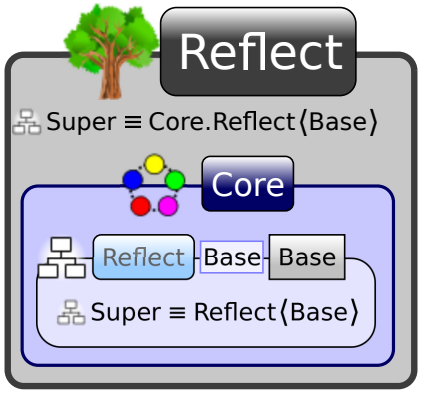

Figure 5.2: Translated sample, the type alias without type arguments

From the plug-in output, as in listing 5.5, we can see that a `TypeAliasTemplateDecl` node which was not supported. There is a second node the `TemplateTypeParmDecl` that is reported.

We translate `TemplateTypeParmDecl` nodes in the `TemplateArgumentVisitor` therefore there is no method for this node in the `ClangASTVisitor`. Because the visitor visits childs automatically for unsupported nodes this `TemplateTypeParmDecl` is also reported with the `TypeAliasTemplateDecl`. With our presented method we can eliminate both errors.

```
ERR/WARN:   In class : ClangAstVisitor
    reason : Not supported
    in clang node : TypeAliasTemplateDecl_NS
    clang node name : Super
    in file : test.cpp
    on line : 11
ERR/WARN:   In class : ClangAstVisitor
    reason : Not supported
    in clang node : TemplateTypeParmDecl_NS
    clang node name : Base
    in file : test.cpp
    on line : 11
```

Listing 5.5: Error when importing a templated type alias

With the name given by the error message, the documentation [12] can be found.

Now we implement the check for duplicates, to achieve this we first implement a function which creates a unique hash of the declaration. For this we add a function `hashTypeAliasTemplate` to the `NodeHasher` class. We can reuse the method for type aliases without templates, and to this append the hash of the template arguments. The method implemented will look like in listing 5.6.

```
const QString hashTypeAliasTemplate(const Clang::TypeAliasTemplateDecl*
    typeAliasTemplate)
{
    QString hash = hashTypeAlias(typeAliasTemplate->getTemplatedDecl());
    auto templateParamList = typeAliasTemplate->getTemplateParameters();
    for( auto templateParam = templateParamList->begin();
         templateParam != templateParamList->end(); ++templateParam)
    {
        hash.append("_");
        if(auto templateType =
            llvm::dyn_cast<Clang::TemplateTypeParmDecl>(*templateParam))
            hash.append(hashTemplateTypeParm(templateType));
        else if(auto nonTemplateType =
            llvm::dyn_cast<Clang::NonTypeTemplateParmDecl>(*templateParam))
            hash.append(hashTemplateTypeParm(nonTemplateType));
    }
    return hash;
}
```

Listing 5.6: Pseudo code of the new method

Now we have to modify the `TranslateManager` class. The method `insertTypeAliasTemplate` will look identical to the `insertTypeAlias` method, except for the adapted hash method.

Next we implement the `TraverseTypeAliasTemplateDecl` method in the `ClangAstVisitor` class. The name we receive from the `getNameAsString()` method. The aliased type we receive with the following line:

```
typeAliasTemplate->getTemplatedDecl()->getUnderlyingType();
```

Then we just have to loop over the template arguments and translate them with the `TemplateArgumentVisitor`. At the end we have to insert the newly generated type alias in the model. Due to the size the finished code of the method is omitted here, but it can be found in the source code.

5.4 Multi threading

The extensions presented previously concentrate on adding features. Another interesting improvement is to find on the performance side. Even if the import currently is rather fast, it should be as fast as possible. This section therefore presents some points to consider to make this plug-in multi threading capable.

Be aware that Clang works in translation units, which means for each source file that we have a separate run of the tool. This offers a great possibility to parallelize the importing.

There are some classes which should be shared amongst all translating threads. The most important one is the `TranslateManager` class, which was presented in section 3.4. For the maps used by this class we have to implement thread safety. This means we have to lock access to a certain map each time a thread writes to this map. For reading we do not need to change anything since the Qt-framework provides this already. The logging class is another class that is shared and since it uses maps it should be changed to be thread safe.

As every translated node gets added to an Envision node we have to make sure that we provide thread safety for such insertion as well. This can either be done by implementing an own thread safety guarantee for the inserted nodes or by using Envision's locking facility as provided by the Envision model.

Given thread safety we can create several threads and split the sources to process amongst them. Each thread can then create its own tool and translate the given sources.

6 Conclusion

We presented a plug-in for Envision which is able to import C++ code to Envision. The plug-in relies on the Clang front-end, we translate from Clang's AST to Envision's model. Most of the code constructs found in Envision's source code we are able to import with this new plug-in. We found ways to merge ASTs from different translation units into the united concise Envision model. Our implementation is able to detect certain AST nodes which are not written in the source code and handles them appropriately. Some parts are still missing to be able to fully import C++ projects, especially preprocessor constructs such as macros are not yet supported. We provided detailed guidelines, which we hope will help to implement the missing functionality.

The plug-in we presented, is a first step towards developing Envision using Envision itself. To achieve this there is still some work to do:

- We have to extend this plug-in to also support preprocessor directives and comments.
- As soon as the plug-in is complete, we should adapt the visualizations to fully support the newly introduced C++ constructs. This is not very crucial but still needed for day to day usage.
- The last and probably the biggest task is then to export Envision's model to C++ code to be able to compile the edited project.

The ability to develop Envision using itself, will hopefully further improve the quality of Envision.

A Importing Envision's code

This section explains how to import Envision's code with the new plug-in. The steps here are very specific to Envision's code, for other code bases you find explanations in section 3.7.

- First clone the Envision repository [4] to some path $PATH$.
- In `PATH/Core` you will find the `common.pri` file which you have to adapt:
  - To `QMAKE_CXXFLAGS` add the parameter `-U__GNUC_MINOR__`
  - In the next line write `DEFINES += __GNUC_MINOR__=8` this fakes gcc version 4.8
  - Specify Clang as the compiler with `QMAKE_CXX = clang++`
  - Add the `include` directory from clang to the `INCLUDEPATH` variable In Linux this may look like this: `INCLUDEPATH += /usr/lib/clang/3.3/include/`
- Now cd to the $PATH$ directory again and run `qmake -r` eventually you have to specify the qt version so you may want to use `qmake-qt4 -r`
- Now you should have a `Makefile` in $PATH$ and in the sub directories which contain code. Create a `maker.sh` file as in listing A.1
- Make sure you have the bear tool [18] installed.
- Run the `maker.sh` script, it will create the needed `compile_commands.json` files
- In the directory where you have your installed version of Envision cd into `CppImport/test`
- In this directory is the `testSelector` file, open it, and add `spath:PATH` where `PATH` is the $PATH$ (the cloned Envision directory) For example this can look like this:

  `spath:/home/luke/BachelorThesis/TestEnvision/Envision` (note that there are no spaces)
- You can now run Envision with the `--test cppimport` argument.

```
#!/bin/bash

for dir in ./*/
do
  (cd $dir && make clean && bear -- make)
  echo "Processed $dir"
done
```

Listing A.1: Script for sub projects

If you observe that not all code was translated, you may want to make sure that the generated `compile_commands.json` files are complete and contain no errors. This was often the case during our development.

References

[1] Unix top utility. http://www.unixtop.org/, July 2013.

[2] Variadic templates. http://en.wikipedia.org/wiki/Variadic_template, July 2013.

[3] Dimitar Asenov. Design and implementation of envision - a visual programming system. Master's thesis, ETH Zürich, 2010.

[4] Dimitar Asenov. Envision on github. https://github.com/dimitar-asenov/Envision, July 2013.

[5] Dimitar Asenov. Envision's project page. http://www.pm.inf.ethz.ch/research/envision, July 2013.

[6] The C++ Standards Committee. Latest publicly available draft - n3690 edition, May 2013.

[7] clang. clang: a c language family frontend for llvm. http://clang.llvm.org/, July 2013.

[8] clang. Frontendaction documentation. http://clang.llvm.org/doxygen/classclang_1_1FrontendAction.html, July 2013.

[9] clang. Json compilation database format specification. http://clang.llvm.org/docs/JSONCompilationDatabase.html, July 2013.

[10] clang. Libtooling documentation. http://clang.llvm.org/docs/LibTooling.html, July 2013.

[11] clang. Ppcallbacks class reference. http://clang.llvm.org/doxygen/classclang_1_1PPCallbacks.html, July 2013.

[12] clang. Typealiastemplatedecl class reference. http://clang.llvm.org/doxygen/classclang_1_1TypeAliasTemplateDecl.html, July 2013.

[13] James O. Coplien. Curiously recurring template patterns. *C++ Rep.*, 7(2):24–27, February 1995.

[14] George Kastrinis Eli Friedman. Mailing list conversation about macros and declarations. http://lists.cs.uiuc.edu/pipermail/cfe-dev/2013-August/031444.html, August 2013.

[15] Eli Friedman. Mailing list conversation about member initializers. http://lists.cs.uiuc.edu/pipermail/cfe-dev/2013-August/031347.html, August 2013.

[16] Manuel Klimek. The clang ast - a tutorial. http://youtu.be/VqCkCDFLSsc?t=5m40s, May 2013.

[17] Daniel Marjamäki. Cppcheck tool. https://github.com/danmar/cppcheck, July 2013.

[18] Laszlo Nagy. Build ear. https://github.com/rizsotto/Bear, July 2013.

[19] George Kastrinis Sam Parker. Mailing list conversation about macros and source locations. http://lists.cs.uiuc.edu/pipermail/cfe-dev/2013-August/031454.html, August 2013.

[20] Mark Teel. Code analyzer tool. http://www.codeanalyzer.teel.ws/, July 2013.