

Self-hosting the Envision Visual Programming Environment

Master Thesis Report

Patrick Lüthi

Supervised by Dimitar Asenov, Prof. Dr. Peter Müller
ETH Zürich

November 11, 2015



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

Envision is a visual programming environment written in C++. In this thesis we are working on self-hosting Envision thereby aiming to enable development of Envision in itself. Our main contributions are creating a code generation framework that can represent macros in Envision, translating C++ macros to our code generation framework and analyzing dependencies between code fragments to enable exporting code from Envision to C++. The code generation facility is able to encode almost all macros used in Envision's C++ code. We are also able to automatically import most of the essential macros used in Envision and the most challenging part of exporting, the dependency analysis, has been tackled.

Table of Contents

List of Figures	3
List of Tables	5
List of Code Listings	6
1 Introduction	7
1.1 Challenges	7
2 Background	9
2.1 Structural differences between Envision trees and C++ source code . . .	9
2.2 Code generation	10
2.3 Existing C++ support	11
2.4 Import-Export Cycle	12
3 Code generation framework	13
3.1 Meta-programming systems in existing languages	13
3.2 Tool support	14
3.3 Syntactic macros	15
3.4 Importance of code context	16
3.5 Meta definition	17
3.6 Meta call	18
3.7 Splicing	19
3.8 Bindings	19
3.9 Predefined meta definitions	20
4 C++ macro import	24
4.1 Integration	24
4.2 Original nodes and clones	25
4.3 Macro reconstruction from the expanded Envision AST	26
4.4 Meta definition reconstruction	27
4.5 Lexical transformation	29
4.6 Macro arguments	31
4.7 Partial macros	31
4.8 X-Macros	33

5	Export to C++	36
5.1	Dependency analysis	36
5.2	Converting a dependency composite to C++	37
6	Implementation details of C++ macro import	39
6.1	Clang	39
6.2	Mapping between Clang and Envision ASTs	40
6.3	Lexical transformation	40
6.4	MacroImporter Components	41
7	Evaluation	44
7.1	Code generation framework	44
7.2	C++ macro import	44
8	Future Work	48
8.1	Known issues	48
8.2	Finalizing the import-export cycle	49
8.3	Further extensions	49
9	Related work	51
9.1	Macro and code generation systems	51
9.2	C++ Preprocessor	51
10	Conclusion	53
	References	54
	Appendices	55
A	Import guide	56

List of Figures

2.1	Tree structure of an example method declaration.	9
2.2	In general it is not clear how to create an Envision AST from C++ code.	10
2.3	X-Macro example showing how a data list can be used to create different code with similar structure.	11
2.4	We want to import C++ code into Envision and export Envision code to C++. This forms the import-export cycle effectively enabling self-hosting of Envision.	12
3.1	Example showing a macro that create a method called <code>ExampleMethod</code> with a print statement expressed both declaratively and imperatively.	14
3.2	Example showing how a <i>syntactic macro</i> and its syntactically complete subelements can be translated to forests of trees in the tree model of Envision.	15
3.3	Example showing a <i>non-syntactic macro</i>	15
3.4	The macro <code>MEMBER_DECLARATION</code> creates a code element that can end up being a field or a variable declaration depending on the context it is expanded in.	16
3.5	Visualization of a <i>meta definition</i> called <code>ATTRIBUTE</code> with arguments and a <i>declaration context</i> of type class.	17
3.6	<i>Meta call</i> to <i>meta definition</i> with stringification and identifier concatenation with resulting generated code.	18
3.7	<i>Meta call</i> to a <i>meta definition</i> called <code>ATTRIBUTE</code> shown in figure 3.5.	18
3.8	A <i>meta definition</i> argument spliced inside a method definition body.	19
3.9	A <i>meta definition</i> creating two different code structures from an argument list. Without specific support for this case the argument list needs to be duplicated.	20
3.10	A meta definition which uses meta-bindings to reduce code duplication.	21
3.11	A macro from Envision's C++ code base with an argument called <code>OVER-RIDE</code> which can toggle the override flag for several methods in the macro body.	22
3.12	A <i>meta definition</i> using the <i>predefined meta call</i> <code>SET_OVERRIDE_FLAG</code>	23
4.1	Integration of the macro import system in the existing C++ import system.	25
4.2	General approach on reconstructing standard <i>meta definitions</i> and <i>meta calls</i> from the <i>expanded Envision AST</i>	26
4.3	Example showing that <i>lexical transformation</i> is needed in order to reconstruct information on stringification or identifier concatenation.	30

4.4	Example <i>begin partial macro</i> specialization using a simplified real example from Envision's C++ source code.	33
4.5	X-Macro example from Envision's C++ code base showing the relationship between partial macros and <i>X-Macro children</i>	33
4.6	Example <i>X-Macro meta definition</i> created when importing Envision's C++ source code (manually fixed two missing <i>lexical transformations</i>).	35
6.1	Components of the macro import system and their dependencies.	42
6.2	Information flow from the existing C++ import system to the macro import system components.	42

List of Tables

3.1	Comparison of properties of meta-programming systems of different languages.	13
7.1	Macro import evaluation: <code>nodeMacros.h</code>	45
7.2	Macro import evaluation: <code>typeIdMacros.h</code>	46
7.3	Macro import evaluation: <code>attributeMacros.h</code>	46
7.4	Macro import evaluation: <code>itemMacros.h</code>	46
7.5	Macro import evaluation: <code>shapeMacros.h</code>	46
7.6	Macro import evaluation: <code>StandardExpressionVisualizations.h</code> . . .	47
7.7	Macro import evaluation: <code>StandardExpressionVisualizations.cpp</code> . .	47

List of Code Listings

4.1	Pseudo code of the main method for macro reconstruction invoked after every translation unit.	26
4.2	Pseudo code showing the recursive method <code>handleMacroExpansion</code> invoked by the code in listing 4.1.	28
4.3	Pseudo code of the <i>meta definition</i> creation method invoked in the code of listing 4.2.	28
4.4	A real code example for a <i>begin partial macro</i> from Envision's C++ source code.	32
4.5	A real code example for an <i>end partial macro</i> from Envision's C++ source code.	32
A.1	Options to be added in <code>common.pri</code> to enable importing.	56
A.2	Script used for importing.	56

1 Introduction

In this thesis we are working on self-hosting the Envision visual programming environment[1]. Envision is a visual code editor aiming to increase productivity of software developers by visualizing code not only using text but also with graphical objects.

The term self-hosting refers to a software's ability to create versions of itself. For example a compiler can compile a different version of itself and a development environment can let the user open and modified its own code. The goal of this thesis is to make Envision self-hosting.

It is an important milestone to attain self-hosting stage when developing a programming environment. Envision is a complex program with a sizeable code base. Being able to handle such a complex real-world program shows Envision's applicability.

Developing Envision in itself will drastically increase usage and problem discovery. That in turn will lead to improving robustness and usability of the program. As a result Envision will become more fit for interactive user studies and programming in general.

Furthermore we are not aware of any other self-hosting visual programming environment. Self-hosting a visual programming environment is a challenging tasks. Existing visual programming environments that we are aware of are either domain specific (for example Labview[4]), or meant for educational and smaller tasks(for example Scratch[7] or MIT app inventor[8]). Envision on the other hand is meant for general purpose programming and targets professional developers.

1.1 Challenges

To be able to edit Envision's code in Envision we first need to be able to load Envision's large C++ code base into Envision. This is a challenging task because of the difference of code representation between Envision and C++. Envision uses trees to store the program structure whereas C++ code is just text. Particularly challenging is the heavy usage of macros in the C++ code base. Not only is it hard, or even impossible in some cases, to convert textual macros into trees but we have to develop a code generation framework for Envision in order to represent them.

After Envision's C++ code has been imported, all the code is in one single tree (no code duplication) and another challenge is to split the code into header and source files when

exporting to C++. Typically the header file contains the public interfaces of declarations whereas the definitions of said declarations reside in the corresponding source file. This means that the dependencies are not the same for both parts. Therefore we have to design a dependency analysis that calculates the set of dependencies given the code of one file.

2 Background

In this chapter we discuss structural differences between Envision and C++, the existing C++ support system and give a short introduction to code generation in order to prepare the reader for the contents of this thesis.

2.1 Structural differences between Envision trees and C++ source code

Envision structures code using a tree model similar to an Abstract Syntax Tree (AST). All code elements are represented by nodes and their properties are captured in child nodes. For example a node representing a method has a child node for the name of the method and another child node representing its body. The corresponding tree is illustrated in figure 2.1.

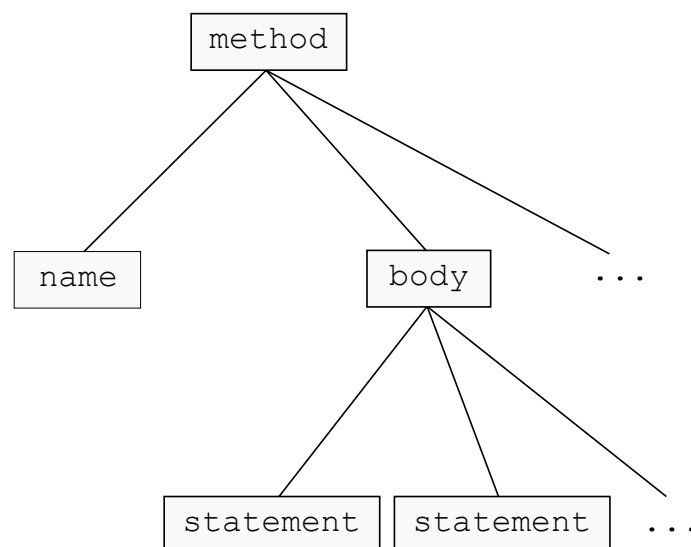


Figure 2.1: Tree structure of an example method declaration.

In contrast to Envision's tree model C++ code is represented by text. The tree consists of nodes whereas the text consists of lexical tokens. This difference results in unequal expressive power of the two domains. It is possible to write just half a class in the textual

domain but you cannot create half a class node in the tree model. This is relevant when writing macros in C++ where it is common to write partial declarations. In general it is hard to map between the two domains (see figure 2.2).

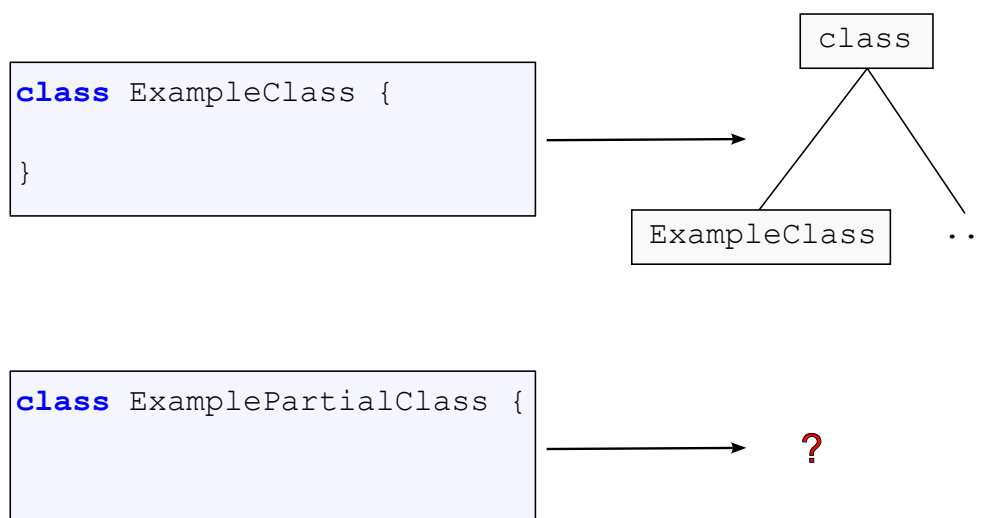


Figure 2.2: In general it is not clear how to create an Envision AST from C++ code.

Another structural difference is that in Envision one code element, for example a declaration, is represented by exactly one node. C++ structures code using text split in header and source files. This split together with the way modularity is achieved in C++ (by text inclusion) means that logically the same declaration might occur in many different places in the expanded code of a translation unit.

2.2 Code generation

Code generation refers to the procedure of creating code as a result of a computation which in most code generation systems takes is done at compile-time. It has many useful applications such as creating code from data or automatic naming. An example of the first is the creation of visualizations for expressions in Envision where the data provided is the operands and operator names and order. Code generation can help to reduce code size, because describing code often takes less space than repeatedly writing code with similar structure, and as a consequence there is less code to maintain.

A more concrete concept of code generation are macros. A macro is a program or piece of code that given some input produces another program or code fragment.

The C++ preprocessor processes code and the result is used as the input to the C++ compiler. C++ macros are implemented in the preprocessor and together with C++ templates they form the code generation possibilities for C++. The preprocessor is unaware of any syntax or semantics of the resulting code. It uses its own language where every preprocessor operation is described starting with a preprocessor directive. The preprocessor can do simple text manipulations (copy, replacement, concatenation

etc.) providing features for C++ like modularity by file inclusion, macros or conditional compilation.

Envision's C++ source code uses a well-known technique called X-Macro to generate different visualizations for expressions provided a list of data defining the expressions. In general X-Macros are used to generate several variants of code with similar structure from the same dataset in C++. It is a very textual mechanism and features usages of macros which are not syntactically complete (see section 3.3) presenting a special difficulty for the code generation framework and when importing code containing X-Macros.

An X-Macro usage consists of a list providing structure and data to be used and the transformation for each list element. Every element in the data list is wrapped in a macro call. The definition of the wrapping macro calls is changed depending on how the data should be used for code generation. Figure 2.3 shows an example of a simple application of the X-Macro technique where we generate an enum class and a list of constants from the same data list.

```
#define DATA_LIST
    DATA_ENTRY(cat)
    DATA_ENTRY(dog)
    DATA_ENTRY(car)

enum ExampleEnum {
    #define DATA_ENTRY(name) name;
    DATA_LIST
    #undef DATA_ENTRY
}

void ExampleMethod {
    #define DATA_ENTRY(name) const int name = 1;
    DATA_LIST
    #undef DATA_ENTRY
}

enum ExampleEnum {
    cat,
    dog,
    car
}

void ExampleMethod {
    const int cat = 1;
    const int dog = 1;
    const int car = 1;
}
```

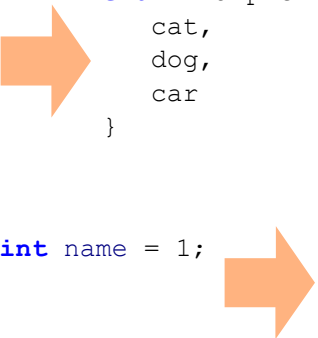


Figure 2.3: X-Macro example showing how a data list can be used to create different code with similar structure.

2.3 Existing C++ support

Prior to this thesis, Lukas Vogel worked on adding C++ support for Envision during his Bachelor's Thesis[10]. The framework he created uses Clang, an open-source front end for the LLVM compiler, to import C++ code into Envision.

The existing C++ import system is not complete. It does not handle macros and other preprocessor directives. Clang first processes the C++ code to be imported and then provides an AST representing the preprocessed code. The resulting imported code thus lost all information about macros and other preprocessor directives. To enable self-hosting we need to retain all information about preprocessing directives and macro usage.

Furthermore some C++ constructs are not yet correctly imported by the existing import system but fixing these issues should be trivial. Clang introduces some implicit nodes that have no corresponding source text and we would like to detect and remove them since they are not part of the original source code directly.

In our work we focus on supporting C++ macros. Other issues are only addressed as time allows but fixing them is generally not considered to be in the scope of this project.

In chapter 3 we discuss the code generation framework we designed in order to encode the necessary macros in Envision.

2.4 Import-Export Cycle

In order to make Envision self-hosting we need to be able to open, edit and save C++ programs. We plan on achieving this by designing and implementing an import-export cycle. The import-export cycle is a concept where we can move between C++ source code and an Envision tree model by means of automatic conversion. There is an import system taking C++ code files as input and producing an Envision tree model and an export system creating C++ code files from an Envision tree model. We plan on reaching our goal of self-hosting Envision by extending the existing import system to be able to import macros, adding a code generation framework and an export system to Envision. These systems combined can then form an import-export cycle as illustrated in figure 2.4.

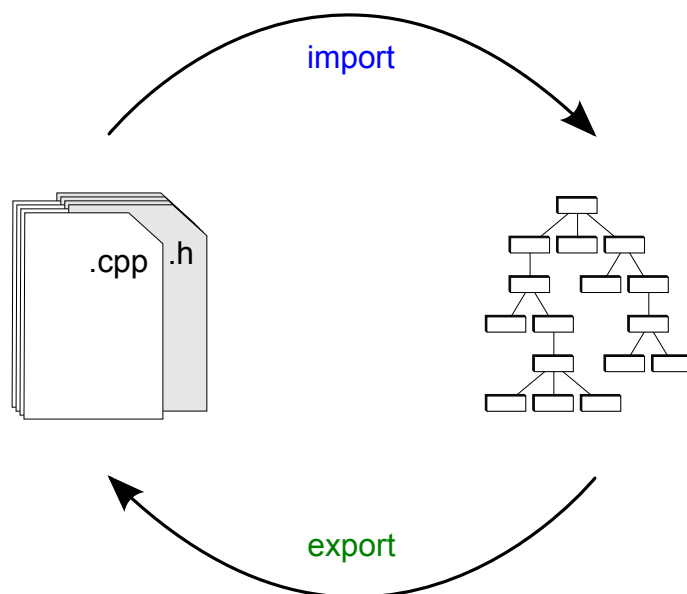


Figure 2.4: We want to import C++ code into Envision and export Envision code to C++. This forms the import-export cycle effectively enabling self-hosting of Envision.

3 Code generation framework

3.1 Meta-programming systems in existing languages

To inform the design of our code generation framework we looked at the solutions other languages use to realize code generation. Table 3.1 compares properties of various languages.

	C++	Scala	Nemerle	CommonLisp
Medium	text	tree	tree	list
Representation	declarative	mixed	mixed	declarative
Meta language	Preprocessor	Scala	Nemerle	CommonLisp
Typed	no	yes	yes	no
Hygiene	no	yes	yes	no (Scheme yes)
Modularity	inclusion	import	using	load

Table 3.1: Comparison of properties of meta-programming systems of different languages.

The *medium* property refers to the structure of code pieces a meta-programming system works with.

The property of being *declarative* or *imperative* refers to the way a macro expresses the computation it performs. A *declarative* approach focuses directly on what the result should be whereas an *imperative* approach focuses on describing how to get to the desired result. Figure 3.1 shows an example of how a macro creating a method with a print statement could look like in each of the two representations. We decided to use a *declarative* approach for our code generation framework because it provides enough flexibility for our needs. Furthermore it allows directly editing macros with existing code editor functionality and the representation is more concise, direct and familiar due to the similarity to C++ macros.

Scala's and Nemerle's meta-programming systems are *typed*. This is achieved by declaring the types of trees which have to be equal to the types of the expressions one would get by evaluating a tree. Our code generation framework is currently not *typed*, because it was not needed to reach the main goal of this project, but we are confident that it is possible to do so.

```

macro DeclarativeMacro()
{
    void ExampleMethod
    {
        print();
    }
}

macro ImperativeMacro()
{
    exampleMethod = new Method("ExampleMethod");
    exampleMethod.addStatement(new PrintStatement());
    return exampleMethod;
}

```

Figure 3.1: Example showing a macro that create a method called `ExampleMethod` with a print statement expressed both declaratively and imperatively.

In meta-programming systems the term *hygiene* refers to a meta-programming system's ability to avoid accidental capturing of names in macros. This means that a *hygienic* meta-programming system avoids capturing of names from surrounding code at the location where a macro is expanded automatically unless instructed otherwise. Our code generation framework is not *hygienic*.

The term *modularity* describes how a developer using a language is able to use symbols in his code which are defined somewhere else (other code files, libraries, etc.). This property is interesting because in C++ modularity is achieved by including code files with declarations which is a preprocessor operation that works very similar to C++ macros.

3.2 Tool support

We want to enable tool support for generated code. In particular tools such as reference resolution or auto-completion are very important when working in a code editor and they should be able to process generated code as well. We found two ways of enabling tool support for generated code. One way is to generate the actual code in form of an Envision AST and append it to the code at the location of the macro expansion enabling tools for reference resolution or auto-completion to work on the expanded AST like if it was normal code. The other approach would be to not generate any actual code but make the tools aware of the code generation framework. Tools then would need to know how to incorporate generated code in their analysis directly, for example by interpreting the corresponding macro bodies.

The interpretation approach could use less memory and would not require any caching of generated code. However it is not clear how complex this approach would be and the design of some tools like Envision's reference resolution system is not yet final. A

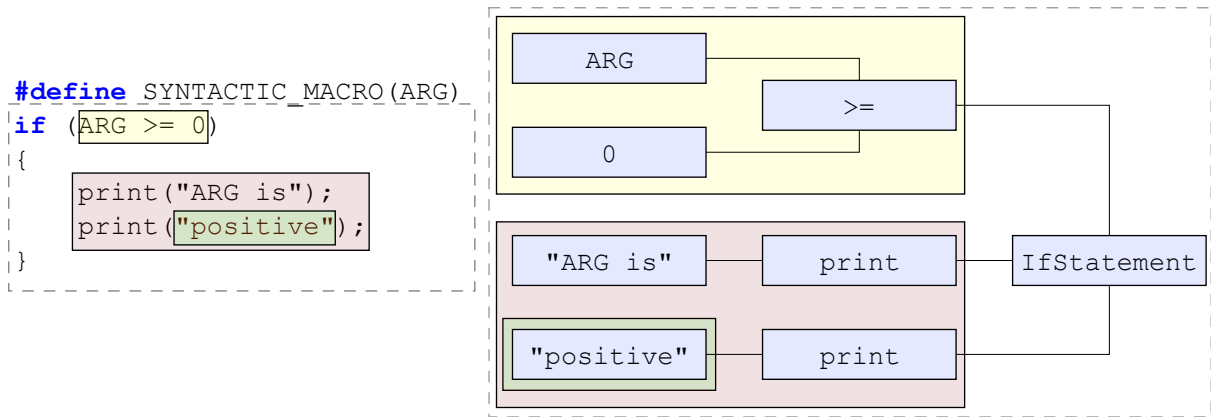


Figure 3.2: Example showing how a *syntactic macro* and its syntactically complete subelements can be translated to forests of trees in the tree model of Envision.

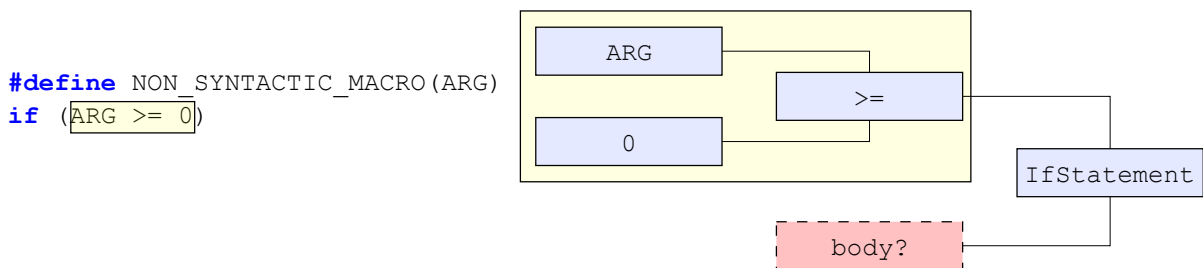


Figure 3.3: Example showing a *non-syntactic macro*.

generative approach on the other hand requires more memory and the system has to keep track of whether previously generated code is still valid. In turn the complexity of the reference resolution system barely increases and other tools can work with the generated code directly.

We decided to use a generative approach since it seems less complex to do overall and will be more performant. Generating code is potentially a costly operation. We therefore cache generated code in the node corresponding to the macro call which generates said code.

3.3 Syntactic macros

Syntactic macro systems work on the level of ASTs unlike the C++ preprocessor which works on the level of lexical tokens. As a consequence macros in syntactic macro systems, called *syntactic macros*, generate syntactically complete code. Here syntactically complete means that the resulting code can be fully described by a forest of ASTs. Figure 3.2 illustrates this concept by comparing C++ code regions with corresponding ASTs. Figure 3.3 shows an example of a *non-syntactic macro* where we cannot fully describe the macro body with a forest of ASTs.

A macro can be directly represented in a tree structure if it is a *syntactic macro*. Therefore *syntactic macros* directly fit Envision’s model. It is not clear how *non-syntactic macros* can be represented by trees in general. The C++ code base of Envision uses *non-syntactic macros* in the context of X-Macros. The code generation framework and macro import system we design therefore has to be able to deal only with certain kinds of *non-syntactic macros*.

3.4 Importance of code context

The representations in C++ and Envision for the same code element encode different information. A field declaration in Envision is a field node. The same piece of code in C++ is only a group of lexical tokens. The field node has a type while the corresponding code in C++ does not. Only by knowing the context of the C++ text the code becomes meaningful. This effect is especially important when working with macros.

A C++ macro can generate code that even provided the same input can change semantics depending on where the macro is called. Figure 3.4 shows an example of this. The pre-processor pastes lexical tokens and is unaware of semantics or the programming language used. The flexibility this provides sometimes helps in further reducing code size which is one of the main goals when using code generation. Additionally for many developers thinking in terms of code as text and not as code elements can make it feel natural that a macro system behaves in this manner.

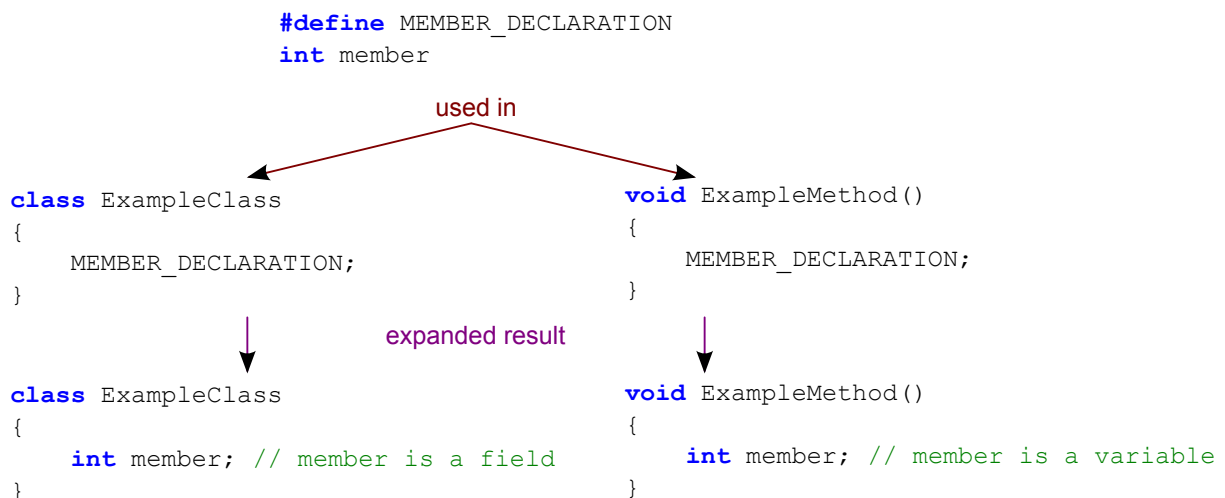


Figure 3.4: The macro `MEMBER_DECLARATION` creates a code element that can end up being a field or a variable declaration depending on the context it is expanded in.

In Envision a field and a variable declaration are two different code element types. As a consequence the code generation framework in Envision has to know exactly what type of nodes it has to create as a result of a macro call. This information could be computed from the macro call site providing the context of the generated code. However in non-trivial cases where the macro call resides inside another macro definition this computation could become complicated. Even if such a computation was done there would still be a

problem with creating and editing macros in Envision. The code editor has to be able to create the right node types when editing. For those reasons we decided to introduce the concept of context as a main part of all macro definitions in our code generation framework which is described in the next section.

3.5 Meta definition

In our code generation framework the equivalent of a C++ macro is a *meta definition*. A *meta definition* has a similar structure to a method. Every *meta definition* has a name, formal arguments and a body. The body consists of a *context declaration* that in turn contains the actual macro code. Figure 3.5 shows a *meta definition* as visualized in Envision.

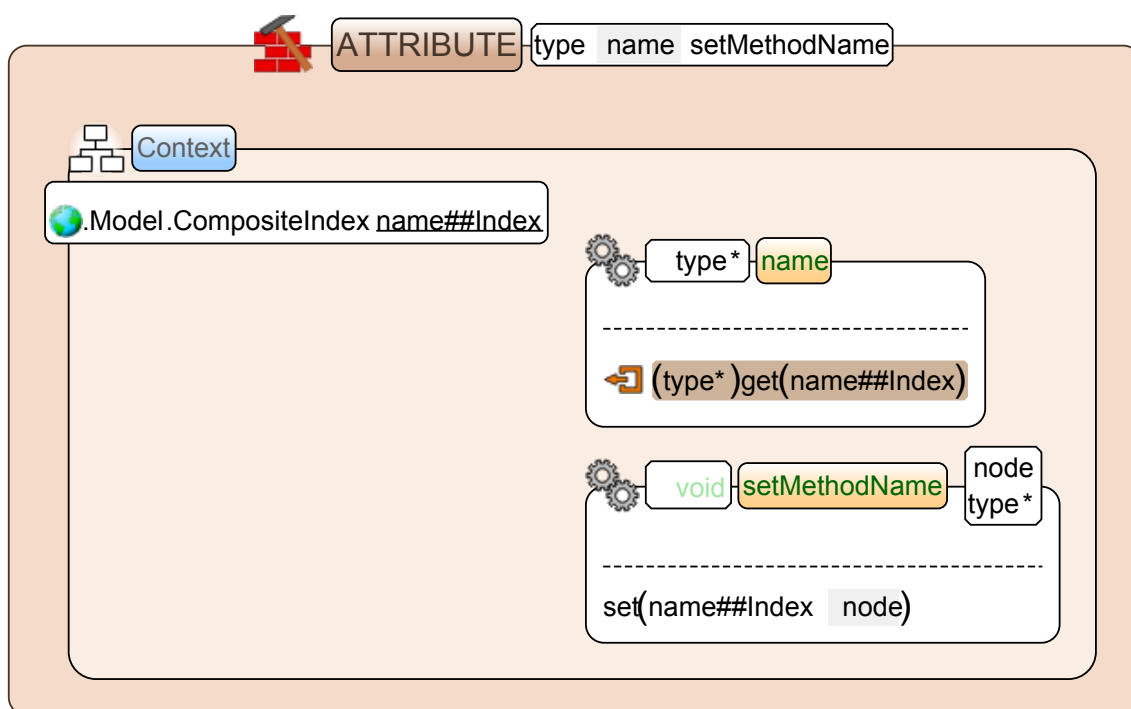


Figure 3.5: Visualization of a *meta definition* called ATTRIBUTE with arguments and a *declaration context* of type class.

The *context declaration* in the meta definition body defines the context type where this definition can be used. The *context declaration* at the location of the usage is called the *actual context*. The *actual context* type has to match the context type of the *meta definition*, body to perform code generation. Having a *context declaration* disambiguates cases as discussed in section 3.4 and allows editing macros directly in the code editor.

Meta definitions are stored in declarations (Projects, Modules, Classes, Methods, etc.). More specifically they can be located in any declaration type that can also be the *context declaration* type.

Figure 3.6 shows how our code generation framework supports features like stringification

and identifier concatenation similar to C++. The syntax is identical to the C++ variants where a double hash (##) is used to concatenate identifiers and a single hash (#) followed by an identifier can be used to stringify the provided identifier.

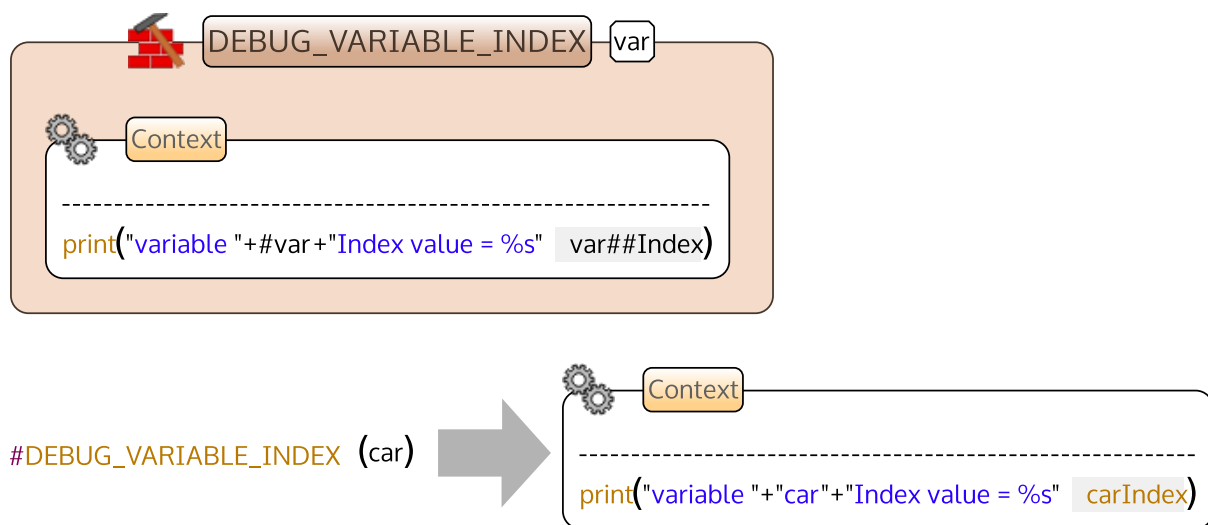


Figure 3.6: *Meta call to meta definition with stringification and identifier concatenation with resulting generated code.*

3.6 Meta call

In our code generation framework the equivalent of a C++ macro call is a *meta call*. *Meta calls* provide the name of the *meta definition* to be used as well as parameters to generate code at their location. Figure 3.7 shows an example of a *meta call* to `ATTRIBUTE` shown in figure 3.5. The arguments of *meta calls* are trees. In case a forest is required as an argument the forest is to be provided in a list node.

Meta calls are expressions and as such can be used anywhere an expression can be used. Additionally *meta calls* can be used directly in all types of declarations that can be a *context declaration* type of a *meta definition* (see section 3.5). This means that the locations a *meta call* can be used in Envision is more restrictive than the locations a macro call can be used in C++ (no restrictions). This is fine because the locations we support is all the locations we need to handle in order to make Envision self-hosting.

`#Model.nodeMacros.ATTRIBUTE(Text name setName)`

Figure 3.7: *Meta call to a meta definition called ATTRIBUTE shown in figure 3.5.*

3.7 Splicing

Similar to the concept of lexical token pasting of macro systems such as the C++ preprocessor the concept of *splicing* is used to insert trees at a certain location of a macro body in many syntactic macro systems. At the place where a tree is to be inserted (spliced) there is a splice identifier, from now on called the *splice*. Usually the *splice* consists of the name of a formal macro argument where the actual argument provided would be the tree to be spliced in. An example of argument splicing is shown in Figure 3.8.

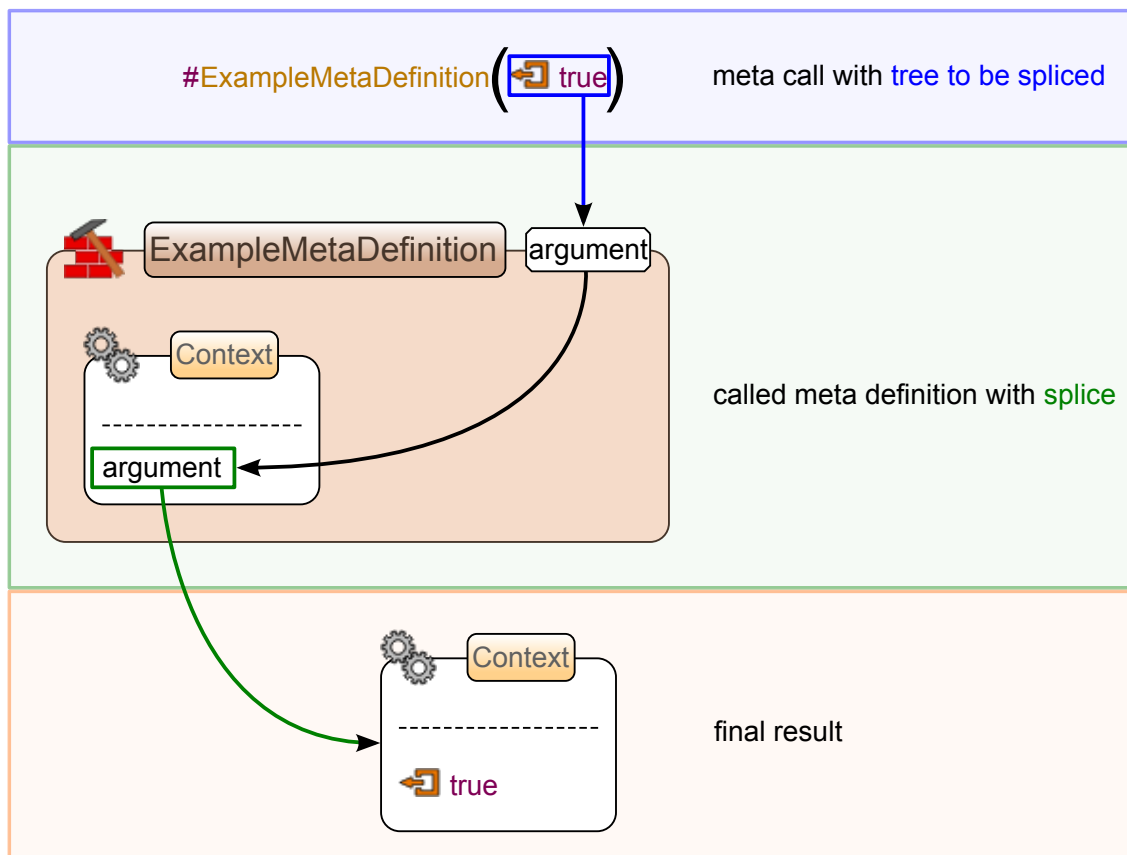
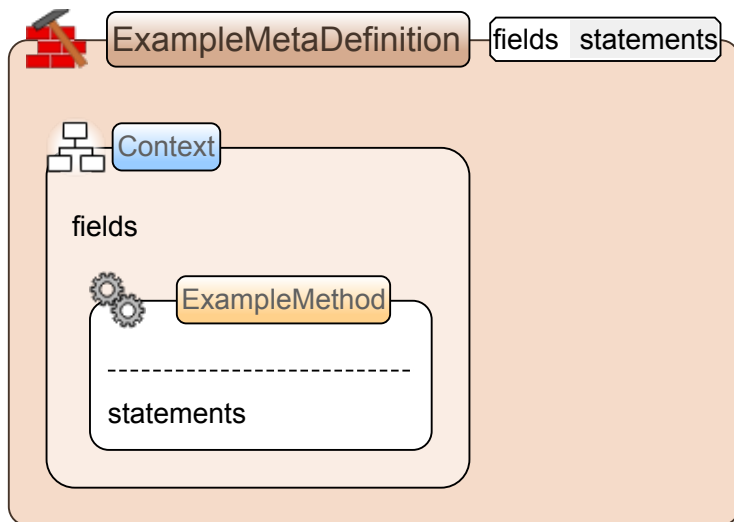


Figure 3.8: A *meta definition* argument spliced inside a method definition body.

3.8 Bindings

Sometimes it is useful to generate different code with similar structure from the same input. Figure 3.9 shows an example where we want to create a list of fields and do something with the same fields in a method body. Code similar to the one shown in example 3.9 exists in Envision's C++ code base and is handled using X-Macros (see section 2.2). Unlike in the example shown in figure 3.9 where we have to provide a similar list twice we ideally only want to specify the list once.

Inspired by X-Macros we developed the concept of *meta bindings*. *Meta bindings* provide a way to generate different code with similar structure from the same input. Like in



```
#ExampleMetaDefinition (
  #CreateField(fieldNameA) #CreateStatement(fieldNameA)
  #CreateField(fieldNameB) #CreateStatement(fieldNameB)
)
```

Figure 3.9: A *meta definition* creating two different code structures from an argument list. Without specific support for this case the argument list needs to be duplicated.

X-Macros, where all entries are wrapped in macro calls, the entries in the data list are wrapped in *meta calls*. *Meta bindings* enable *meta definitions* to provide custom meaning to the wrapping *meta calls* provided in arguments. Figure 3.10 shows the same example as in figure 3.9 but using *meta bindings*. A *meta binding* consists of an input referencing a formal *meta definition* argument, a *meta call* mapping and a name for the result of the *meta binding*. All *meta calls* inside the input are transformed using the mapping and the resulting tree is available as local variable with the name of the *meta binding* in the *meta definition* body.

3.9 Predefined meta definitions

The designed code generation framework is limited to creating trees. There is no way to modify existing nodes by using just the introduced concepts up to this point. Figure 3.11 shows an example from Envision’s code where modifying existing nodes is necessary. The override flag is not stored in a separate node but together with all modifiers for a declaration in one node. Setting the override flag thus requires us to modify that node, which exists for all declarations as soon as the declaration itself exists, by using a *meta call*. While this is not possible using the standard *meta definitions* we provide such functionality by means of *predefined meta definitions*. *Predefined meta definitions* are *meta definition* with reserved names. When the code generation framework encounters a *meta call* to a *predefined meta definition* it does not look for a *meta definition* node in the current project but instead performs a special operation depending on the reserved name

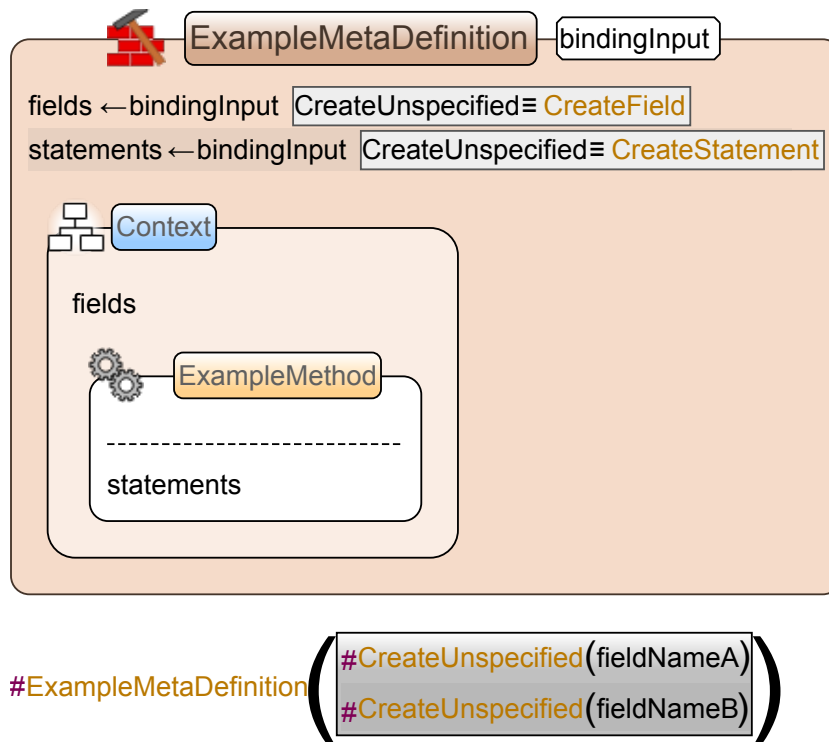


Figure 3.10: A meta definition which uses meta-bindings to reduce code duplication.

used. This allows us to encode any transformation on a node but all effects of *predefined meta definitions* have to be hardcoded in the code generation framework.

In order to solve the problem posed by the real example from Envision's code in figure 3.11 we added a predefined meta definition called `SET_OVERRIDE_FLAG` which toggles the override flag of the surrounding declaration based on a boolean argument provided in the predefined meta call. Figure 3.12 shows how the macro from figure 3.11 is represented in our code generation framework.

```

#define DECLARE_TYPE_ID_COMMON(OVERRIDE)
public:
    virtual const QString& typeName() const OVERRIDE;
    virtual int typeId() const OVERRIDE;

    /* Returns an ordered list of all ids in the type hierarchy of this class. */
    /* The most derived id appears at the front of the list. */
    virtual QList<int> hierarchyTypeIds() const OVERRIDE;
    virtual bool isSubtypeOf(int type) const OVERRIDE;
    virtual bool isSubtypeOf(const QString& type) const OVERRIDE;

    static const QString& typeNameStatic();
    static int typeIdStatic() { return typeId_; }
    static void initType();

private:
    static int typeId_;

```

Figure 3.11: A macro from Envision’s C++ code base with an argument called `OVER-
RIDE` which can toggle the override flag for several methods in the macro body.

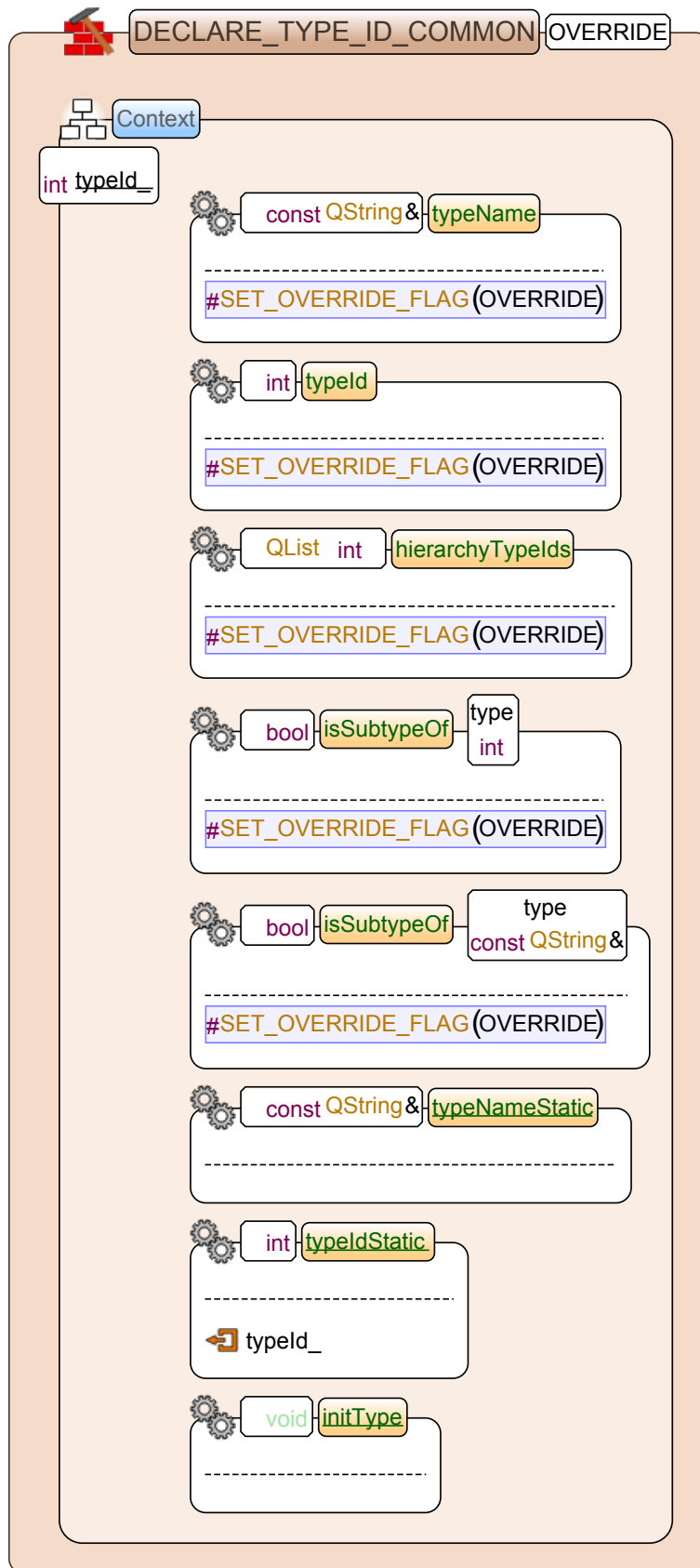


Figure 3.12: A meta definition using the predefined meta call SET_OVERRIDE_FLAG.

4 C++ macro import

In this chapter we discuss the design and integration of the macro import system.

4.1 Integration

The macro import system works together with the existing C++ import system. Figure 4.1 provides an overview of the combined import system.

We first let Clang translate the C++ source code we wish to import into a Clang AST. In order to do that Clang first preprocesses the C++ code and then creates an AST representing the expanded source code. The macro import system uses Clang preprocessor callbacks to collect preprocessing information. In the current design we only collect macro expansion information because we do not yet have a representation for other preprocessing directives in Envision. The process of collecting information repeats itself after every translation unit that Clang processes. The information is then just stored until after the existing import system is done processing the translation unit.

After preprocessing, the existing C++ import system translates the Clang AST of the fully expanded C++ code into an *expanded Envision AST*. During this conversion the macro import system gathers information about both the Clang and Envision ASTs in order to be able to map between them later on. In particular we calculate a range of the source text that corresponds to a Clang AST node and the Envision AST corresponding to the Clang AST node to get a mapping from Envision AST node to source range. This is important because all the information from the Clang preprocessor callbacks references source locations and we need a way of associating Envision AST nodes with the collected preprocessing information.

After the *expanded Envision AST* for one translation unit was generated by the existing C++ import system the macro import system uses the preprocessor information collected during this translation unit to reconstruct macros. In general what we want to do when reconstructing macros is to identify the parts of the expanded Envision AST that originate from a macro expansion. The identified subtrees together with the information collected are then used to create *meta definitions*. At the location where the expanded code used to be we then insert an appropriate *meta call* akin to the macro call that used to be at the same location in the C++ source code (see figure 4.2). The following sections discuss necessary concepts and describe the process of reconstructing *meta definitions*.

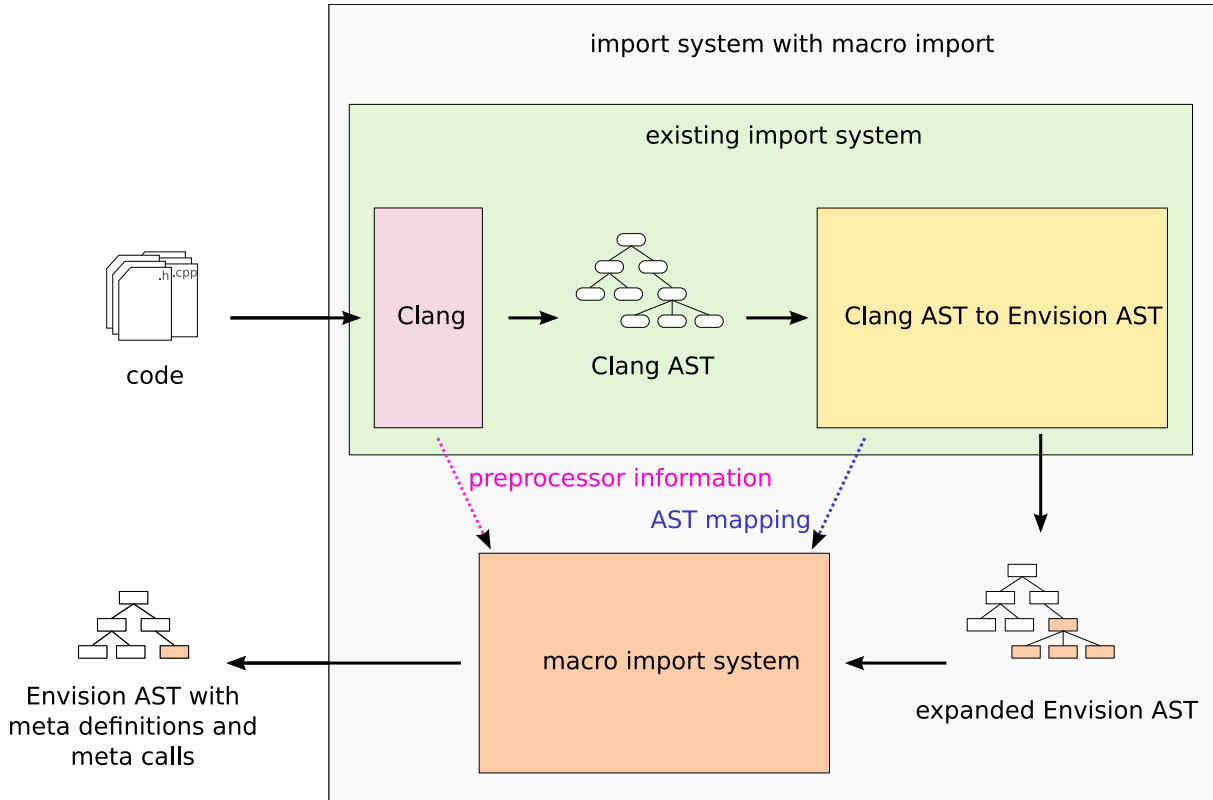


Figure 4.1: Integration of the macro import system in the existing C++ import system.

4.2 Original nodes and clones

One node in the *expanded Envision AST* can occur in many different translation units. For example a method which is declared in a header file that is included in many source files will exist only once in the *expanded Envision AST* but occurs in multiple translation units (usually one source file corresponds to one translation unit). If we modified the original *expanded Envision AST* during the processing of one translation unit then information could be missing the next time the same region of the tree is used by another translation unit. An example for this would be when replacing the expanded code generated by a macro with a *meta call* to the reconstructed *meta definition*. Therefore we have to leave the original *expanded Envision AST* unmodified until all translation units have been processed. We achieved this by introducing the concept of cloning with a mapping. When cloning a tree it returns the cloned tree together with a mapping that can be used to map from original nodes to clones and vice versa. Using cloning with mapping we can clone the parts of the original *expanded Envision AST* that we need to modify and use them, for example in a *meta definition* body, together with the mapping to relate back to the original *expanded Envision AST* without modifying it.

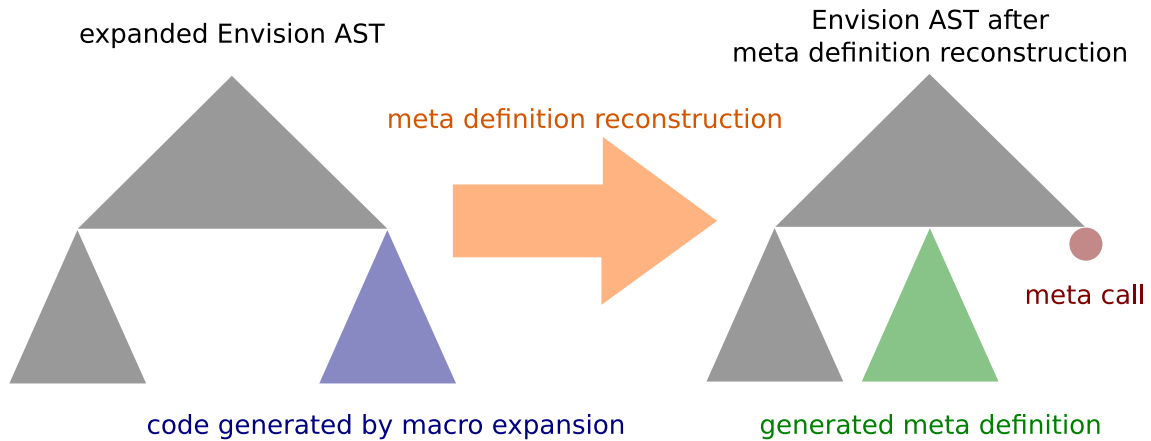


Figure 4.2: General approach on reconstructing standard *meta definitions* and *meta calls* from the *expanded Envision AST*.

4.3 Macro reconstruction from the expanded Envision AST

In this section we discuss how we reconstruct macros, given the collected information as described in section 4.1, from the *expanded Envision AST* resulting from the existing import system.

Listing 4.1 shows the main routine of the macro import system invoked after every translation unit in pseudo code (implementation details omitted).

```

1   for (expansion : topLevelExpansions)
2   {
3       generatedNodes = topLevelNodes(expansion);
4
5       handleMacroExpansion(generatedNodes, expansion);
6
7       if (!alreadyInsertedMetaCall(expansion))
8       {
9           finalizationMetaCalls.insert(expansion.expandedNode, expansion);
10      }
11
12      calculateFinalizationNodes(generatedNodes, mapping);
13  }
```

Listing 4.1: Pseudo code of the main method for macro reconstruction invoked after every translation unit.

We first calculate the *top level macro expansions* of the current translation unit. A *top level macro expansion* is a macro expansion which does not occur inside another macro expansion. All macro calls in non-generated code therefore result in a *top level macro expansion*. The following steps are repeated for every *top level macro expansion* found:

1. Line 3: We calculate all the top level nodes belonging to (that are generated by) the current *top level macro expansion*. A node is a *top level node* iff the node belongs to the current expansion but its parent does not. The result of the computation is a list of nodes with subnodes forming the forest generated by the current *top level macro expansion*.
2. Line 5: Next we process the current *top level macro expansion* and all *child expansions*, the macro expansions inside another macro expansion. This process takes care of creating a *meta definition* for the current *top level macro expansion* and all its *child expansions* and is discussed in detail later in this section.
3. Line 7: We check whether a *meta call* to the *meta definition* representing the current *top level macro expansion* should be inserted in the non-generated Envision code. In particular we want to prevent adding a *meta call* that represents a macro call in the C++ source code that has already been added in another translation unit. Such effects occur due to code file inclusion in the preprocessed source code where some code fragments that are logically only present once in the original source code get duplicated and processed in multiple translation units.
4. Line 9: In the case where we want to insert a *meta call* to the *meta definition* representing the current *top level macro expansion* we do not directly modify the original *expanded Envision AST*. Instead we remember what node needs to be replaced by the meta call (the node to be replaced, called *expansion node*, was computed in the call in Line 5). After all translation units have been processed the `finalizationMetaCalls` list is used to do a final transformation on the *expanded Envision AST* by replacing the nodes with appropriate *meta calls*. Section 4.2 discusses in more detail why we cannot directly edit the *expanded Envision AST* at this point in the import.
5. Line 12: Similar to Line 9 we do not directly remove the forest belonging to the current *top level macro expansion* from the *expanded Envision AST*. Instead we remember all the nodes belonging to the current *top level macro expansion*. These nodes are removed from the *expanded Envision AST* after all translation units have been processed.

4.4 Meta definition reconstruction

In this section we discuss how we reconstruct *meta definition*, given the collected information as described in section 4.1, from the *expanded Envision AST* resulting from the existing import system.

Listing 4.2 shows the recursive method `handleMacroExpansion` invoked by the code in listing 4.1 (implementation details omitted). It handles the computation of the *expanded node* as well as the creation of a *meta definition* for the macro expansion. The *expanded node* of a macro expansion is the node in the original *expanded Envision AST* that should be replaced by a *meta call* to the *meta definition* representing the macro expansion.

```

1 void handleMacroExpansion(nodes, expansion)
2 {
3     for (childExpansion : expansion.children)
4     {
5         handleMacroExpansion(topLevelNodes(childExpansion), childExpansion);
6     }
7
8     calculateExpandedNode(expansion);
9
10    createMetaDefinition(nodes, expansion);
11 }

```

Listing 4.2: Pseudo code showing the recursive method `handleMacroExpansion` invoked by the code in listing 4.1.

Listing 4.3 shows a simplified version of the *meta definition* creation method invoked in the code of listing 4.2.

```

1 void createMetaDefinition(nodes, expansion)
2 {
3     metaDefinition = createEmptyMetaDefinition(expansion);
4
5     for (node : nodes)
6     {
7         applyLexicalTransformations(node);
8         insertChildMetaCalls(node, expansion);
9         removeUnownedNodes(node, expansion);
10        insertArgumentSplices(node, expansion);
11        addNodeToDeclaration(node, metaDefinition.context);
12    }
13 }

```

Listing 4.3: Pseudo code of the *meta definition* creation method invoked in the code of listing 4.2.

The following steps are performed in order to create a *meta definition* given a set of nodes that were generated by the expansion we want to reconstruct:

1. Line 3: We create an empty *meta definition*. Here empty refers to the *context declaration* content (or body) of the *meta definition*. At this point we add the formal arguments of the *meta definition* and register it in our macro import system so we can retrieve it later if needed.
2. Line 5: We do the next steps for all the nodes we identified to be a top level node of this *meta definition*. The reason we can have multiple nodes is that a *meta definition* body might contain a forest and not just a single tree.
3. Line 7: At this point we do *lexical transformation* of this top level tree in order to recover stringification and identifier concatenation. *Lexical transformation* is discussed in detail in the next section.

4. Line 8: Here we insert all the *meta calls* to macro expansions inside the expansion we are currently reconstructing. The *meta definitions* for these *child expansions* have already been reconstructed by virtue of the recursive caller method shown in the code of listing 4.2.
5. Line 9: We remove nodes that are children of the current tree we want to add to the *meta definition* but do not actually get generated by this *meta definition*. This can happen if for example a field declaration is generated by this *meta definition* but the field also has an initialization which originates from some other part of the code. This initialization is detected at this point as unowned by this *meta definition* and removed accordingly.
6. Line 10: Here we add *splices* (see section 3.7) for arguments of this *meta definition*. In particular this means we identify the potential subtree generated from a *meta definition* argument and replace it with a *splice* to the corresponding formal argument. In section 4.6 we discuss macro arguments in more detail.
7. Line 11: We add the reconstructed tree to the *meta definition* body we are currently reconstructing.

4.5 Lexical transformation

Information about identifier concatenation, stringification and sometimes macro arguments and name qualification is lost during the creation of the Clang AST in the existing C++ import system. Figure 4.3 illustrates the problem. Related information is only easily available in reduced or expanded form. In order to reconstruct macros we need to recover that information. The only way to do this is to go down to the level of source text and parse the text directly.

The macro import component responsible for *lexical transformation* is informed about which Clang AST node corresponds to which Envision AST node in the import phase where the existing import system converts the Clang AST to an *expanded Envision AST*. It then uses various properties of Clang nodes (dependent on the type of the node) to find the source text that best matches the associated Envision AST node. In many cases the whole source text belongs to the node. However some Clang AST nodes (mostly `clang::Type` nodes) do not provide a way to directly find the source text that belongs to them. Hence it is not possible to always exactly match a node with a piece of source text. In that case we are looking for a larger piece of source text that is guaranteed to contain the source text we are interested in. We then use information on surrounding code and node type to choose a regular expression to extract the relevant information. The resulting source text is then stored in a map from Envision AST node to said source text and used later when creating a *meta definition*. This process is not very flexible and there are still a few use cases not properly handled by the current implementation.

C++ macro with concatenation

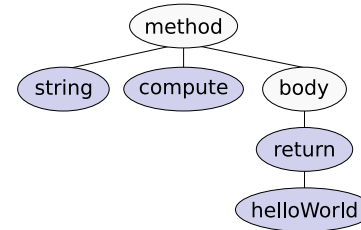
```
#define CONCAT_MACRO(macroArg)
    return hello##macroArg

string compute() {
    CONCAT_MACRO(World);
}
```

existing import system



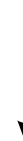
expanded Envision AST



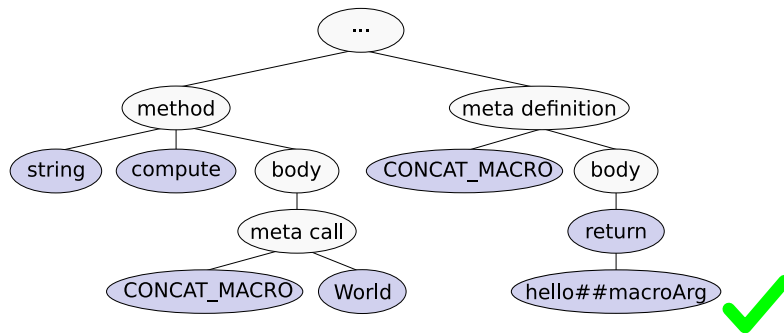
lexical transformation



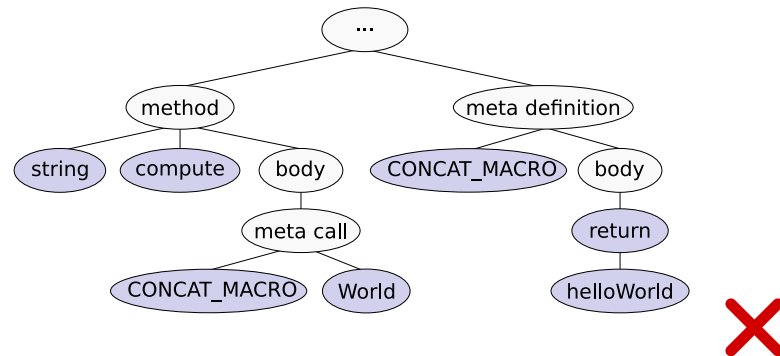
meta definition reconstruction



Envision AST after meta definition reconstruction with lexical transformation



Envision AST after meta definition reconstruction



30

Figure 4.3: Example showing that *lexical transformation* is needed in order to reconstruct information on stringification or identifier concatenation.

Every time a tree is added to a *meta definition* some of its nodes are transformed according to the stored source text from the *lexical transformation* component. Sometimes the text has to be used to not only transform existing nodes but create new trees from the stored source text. We therefore wish to minimize the usage of stored *lexical transformations*. This is achieved by checking whether there is at least one formal argument of the current *meta definition* with a name occurring in the stored source text. If there is no such occurrence we know for sure that we do not have to lexically transform this particular node because it cannot possibly be the result of an identifier concatenation or stringification.

4.6 Macro arguments

When creating *meta calls* all actual arguments are by default instantiated to be reference expressions with a name equal to the spelling of the C++ code macro argument. This means that for example instead of a boolean literal node there would be a reference expression with the name "true" and we do not want that. Instead we wish to identify a forest inside the code generated by a *meta call* corresponding to an actual *meta call* argument. We achieve this by using the gathered information in the Clang AST to Envision AST transformation. It allows us to find a source range for each Envision node. Clang provides a way of checking whether a source location points to the beginning or the end of an expanded macro argument. In all cases where we can map Envision nodes to source locations that are identifiable as the beginning and end of an expanded macro argument by Clang we can use them as actual *meta call* arguments. As a consequence in the previous example with the boolean literal the reference expression with the name "true" is replaced by a node of type boolean literal which was created as part of the *expanded Envision AST* in the existing macro import system.

4.7 Partial macros

Not all macros in the C++ code base of Envision are syntactically complete. In the context of X-Macros there occur syntactically incomplete macros (see section 3.3) that have to be handled in a special manner by our code generation framework. The structure of all used syntactically incomplete macros fall into only a few categories. Only syntactically incomplete macros falling into one of these categories are handled properly by the current code generation framework. If a syntactically incomplete macro does not fall into one of the categories the behavior is undefined yet most likely it will just incorrectly reconstruct the macro.

In the following we refer to syntactically incomplete macros with some extra properties as partial macros. In particular we differentiate between two kinds of partial macros:

1. *Begin partial macros* (*bp-macros*) are all syntactically incomplete macros that start syntactically complete and can be made complete by a following *ep-macro*. Listing 4.4 shows a real example of a *bp-macro* from Envision's C++ source code.

2. *End partial macros* (*ep-macros*) consist of just a closing brace ("}") thus completing a *bp-macro*. Listing 4.5 shows a real example of an *ep-macro* from Envision's C++ source code.

```
1 #define BEGIN_STANDARD_EXPRESSION_VISUALIZATION_BASE(apiSpecification,
2     className, nodeType, styleTypeName)
3 class apiSpecification className
4     : public ::Super<::OOVisualization::VExpression<className,
5         ::Visualization::LayoutProvider<>, nodeType>> {
6     ITEM_COMMON_CUSTOM_STYLENAME(className, styleTypeName)
7     public:
8         className(::Visualization::Item* parent, NodeType* node,
9                 const StyleType* style = itemStyles().get());
10    protected:
11        virtual void determineChildren() override;
```

Listing 4.4: A real code example for a *begin partial macro* from Envision's C++ source code.

```
1 #define END_STANDARD_EXPRESSION_VISUALIZATION };
```

Listing 4.5: A real code example for an *end partial macro* from Envision's C++ source code.

4.7.1 Identifying partial macros

In general it is hard to identify whether a C++ macro is syntactically incomplete without parsing the code in the context it is used. We decided to use a name convention to help identify partial macros. This is acceptable since the main focus of this project lies on importing Envision and not other code with partial macros. By definition all *bp-macro* names must start with `BEGIN_` while all *ep-macro* names must start with `END_`. This way of naming macros is somewhat standard with X-Macros and Envision was already using these names prior to our work.

4.7.2 Begin partial macro specialization

It is possible to create a *bp-macro* by calling another *bp-macro* and then adding code to the open end of the macro body. We call the resulting *bp-macro* a *bp-macro* specialization because it specializes the *bp-macro* called in the macro body. It is necessary to handle such macros because they occur in the C++ code base of Envision in the context of X-Macros.

In order to import specialized *bp-macros* we first find the base *bp-macro* that is the *bp-macro* called inside the specialized macro. The meta definition corresponding to the base *bp-macro* is then transformed to accept one additional argument which is spliced in at the end of the *meta definition's* body. After this transformation the meta definition of the

specialized macro consists just of a call to the *meta definition* of the base macro call with the specializations added as the additional actual arguments as illustrated by figure 4.4.

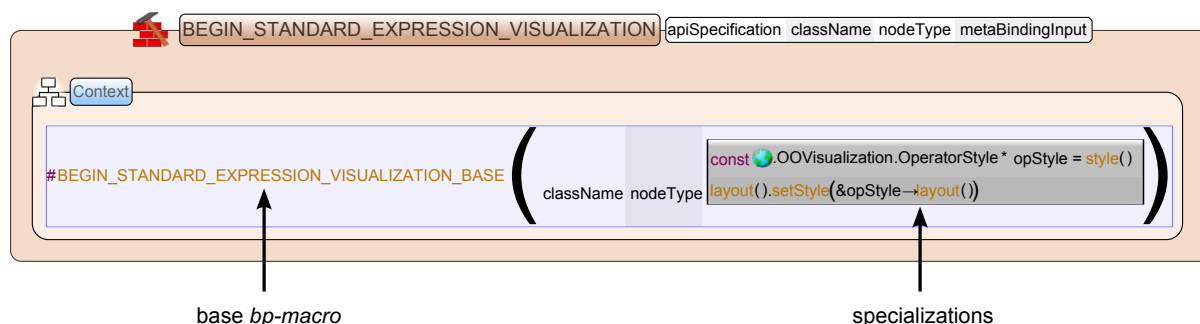


Figure 4.4: Example *begin partial macro* specialization using a simplified real example from Envision’s C++ source code.

4.8 X-Macros

In this section we describe the design of the part of the macro import system responsible for handling X-Macros.

In the previous section we introduced the concept of *bp-macros* and *ep-macros*. Their purpose in importing the C++ code base of Envision is to surround other macro calls as shown in Figure 4.5. In the following we refer to the surrounded macro calls as *X-Macro children*.



Figure 4.5: X-Macro example from Envision’s C++ code base showing the relationship between partial macros and *X-Macro children*.

Macro expansions are identified to be *X-Macro children* if they follow a *bp-macro* expansion and precede an *ep-macro*. This is enabled by using the fact that the preprocessor processes the code top to bottom.

4.8.1 X-Macro meta definition

X-Macros are often (in the case of Envision always) used together with *bp-macros* and *ep-macros*. In particular there is a *bp-macro* in a header file that has a counterpart (also a *bp-macro*) in a source file and the same for *ep-macros*. In the following we discuss how we use *meta bindings* and the relationship between *bp-macros* and *ep-macros* to generate a *X-Macro meta definition*.

To enable the usage of *meta bindings* (see section 3.8) we need to create a *meta definition* for X-Macros. In that *X-Macro meta definition* we bind the wrapping *meta calls* of *X-Macro children* to some specific *meta definition*. In order to do that we need to identify the header and source *bp-macros*. The two *bp-macros* are matched by assuming that the generated code from the header *bp-macro* contains the *meta call* to the source *bp-macro*. This is very specific to Envision and in Envision's current C++ code base header *bp-macros* always introduce a new declaration. Everything following a header *bp-macro* or belonging to the source *bp-macro* counterpart is going to be inside that declaration. Therefore once we find a *bp-macro* we just assume that it is a header *bp-macro* and look for another *bp-macro* in the code generated by it. If we can find such an other *bp-macro* then that means that it is the corresponding source *bp-macro*.

Given two matching *bp-macro* calls we first find the two *base partial macros* (see section 4.7.2). The two *base bp-macro meta definitions* are then merged into one *X-Macro meta definition*. We add an extra formal argument to the merged *meta definition* to provide potential *X-Macro children* and add bindings for both a list of declarations and a list of statements. The list of declarations can be filled by *X-Macro children* surrounded by the *bp-macro* of the header file while the statement list can be filled by *X-Macro children* surrounded by the *bp-macro* of the source file. The *splice* for the statement list is inserted at the end of the last method in the meta definition body of the source *bp-macro*. This transformation is very specific to Envision but sufficient because it allows us to handle all the patterns appearing in Envision's C++ source code is all we need to do.

The mappings for the two *meta bindings* are inserted upon discovering *X-Macro children*. Every time an *X-Macro child* is added to a *meta definition* argument we look up the corresponding *X-Macro meta definition* and check whether there exists a mapping for both *meta bindings* for the *X-Macro child*. If that is not the case we add new mappings to both *meta bindings* by adding a fresh *meta binding* name that binds to each of the actual *meta definitions* (for header and source part of the *X-Macro child*). Figure 4.6 shows a *X-Macro meta definition* created when importing Envision's C++ source code.

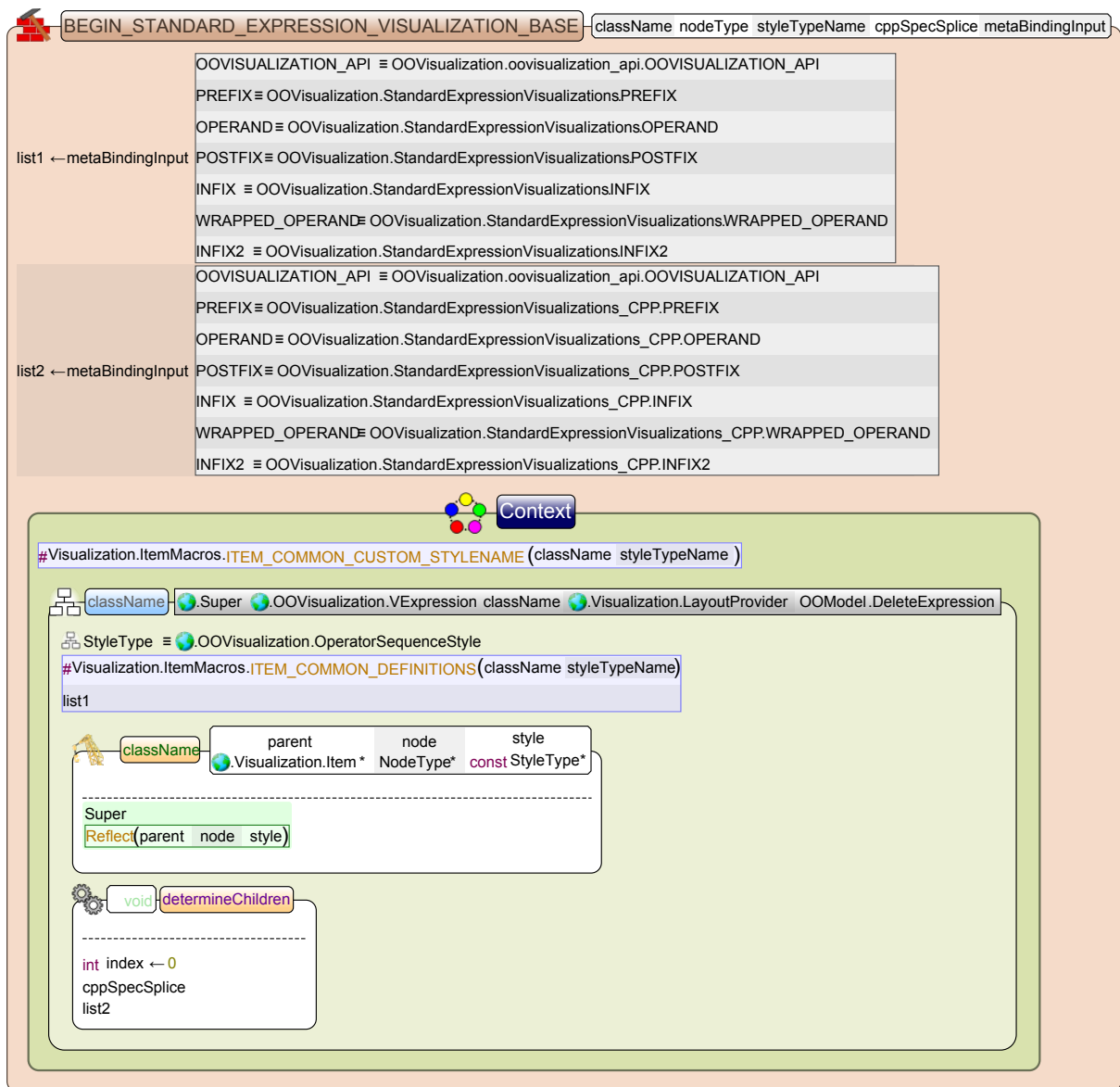


Figure 4.6: Example *X-Macro meta definition* created when importing Envision's C++ source code (manually fixed two missing *lexical transformations*).

5 Export to C++

The last remaining task of the import-export cycle is exporting Envision's tree model to C++. The main challenges here are splitting Envision code in header and source files as well as analysing dependencies to properly add includes or forward declarations to the exported C++ files. Another challenge is converting *meta definitions* and *meta calls* back to C++ macros. Furthermore our goal is to not only export code which is runnable but we want to get C++ code that is as close as possible to the C++ code we imported.

5.1 Dependency analysis

In this section we discuss the design of the *dependency analysis* part of the export mechanism. The *dependency analysis* uses Envision's reference resolution system to find reference targets.

A *dependency reference* is a data structure holding the target of a reference and whether this reference requires a hard dependency on the target or whether a name-only dependency is enough. A *soft dependency* or *name-only dependency* is a dependency where we only need to know that the name of a declaration targeted by a reference exists (forward declaration in C++). A *hard dependency* is a dependency which is not a *soft dependency* but needs to know about size and structure of the type targeted by a reference as well. *Hard dependencies* are translated to include statements while name-only dependencies are translated to forward declarations when exporting to C++.

A *dependency unit* is an inseparable forest of Envision trees that belong together when exporting. Ultimately all code in a *dependency unit* is guaranteed to be in the same C++ file after exporting. Currently every Envision class as well as every module containing no classes are treated as one *dependency unit* each.

Sometimes it makes sense to put multiple classes in the same C++ file. *Dependency composites* are groups of *dependency units* which allow multiple *dependency units* to be put in the same C++ file. If a *dependency unit* is to be put in the same file as another *dependency unit* or the resulting file should be named differently from what the *dependency unit* name suggests this can be manually specified by virtue of a configuration file.

5.1.1 Dependency composite order

In this subsection we discuss an approach on topologically ordering dependency composites in order to export required *dependency composites* (*dependency composites* another *dependency composite* depends on) first. Near the end of this thesis we realized that this approach is not going to work. We outline the problem and how we intend to fix it at the end of this subsection.

Dependency composites contain *dependency units* that depend on other *dependency units*. Those *dependency units* in turn belong to a *dependency composite*. Following this relationship *dependency composites* depend on other *dependency composites*.

In any valid assignment of *dependency units* to *dependency composites* the resulting dependency graph is a directed acyclic graph (DAG). More importantly the graph corresponding to Envision's code must be acyclic because otherwise the C++ code base would not compile.

Nodes in a DAG can be topologically ordered. The topological ordering provides a hierarchy over dependencies allowing us to process dependency composites higher up the hierarchy independently from the ones lower down the hierarchy. We use Kahn's algorithm¹ to compute a topological ordering over all *dependency composites* using the dependency relation. The algorithm also provides a way to check for cycles indicating an invalid assignment of *dependency units* to *dependency composites*.

Additionally to the processing order when translating *dependency composites* to C++ header and source files the topological sort is used to order *dependency units* in the same *dependency composite*.

Near the end of this thesis we realized that the proposed approach is not going to work. In particular it is okay for two *dependency composites* to depend on each other as long as the header files after splitting the code of the *dependency composites* do not have *hard dependencies* on each other. We therefore have to use *dependency composites* only as containers to group multiple classes in one header/source file pair. The proposed analysis still works as long as it is done on some other *dependency reference* collection that only contains the *dependency references* used in a header or source file.

5.2 Converting a dependency composite to C++

In this subsection we discuss how we intend to output C++ header and source files given a *dependency composite* and the results of the *dependency analysis*.

The header file should contain all public declarations as well as inlined functions whereas the source file contains all definitions. This means that we need to split the code inside a *dependency composite* in a public interface declarations and inline functions parts and print the result in the corresponding header file. Everything else should be output in the source file. This separation is done by the component which actually prints the C++ code since that component must know what part of a tree to print in one file either way.

¹https://en.wikipedia.org/wiki/Topological_sorting#Kahn.27s_algorithm, 07.11.2015

The output files now contain the C++ code but are missing dependency information like forward declarations and inclusion. This information can be requested from the *dependency analysis* system. The only information needed to calculate the appropriate dependencies are the *dependency references* used to output each file. Finally we apply transitive reduction on the calculated set of dependencies and print the result on top of the respective files.

The dependency relationship is transitive. The list of all dependencies of a file can have redundant entries if some of the dependencies are reachable via others. The process of removing redundant edges in a graph to nodes that can be reached via another path is called *transitive reduction*. In the context of dependencies this means that we reduce the amount of dependencies such that we cannot remove any further dependencies from the list but the transitive closure of the dependencies remaining after the transitive reduction is equal to the transitive closure of the original dependencies.

6 Implementation details of C++ macro import

In this chapter we discuss implementation details of the C++ macro import system we designed.

6.1 Clang

The information we collect during the existing import system's work largely relates to data over Clang source locations so we need a way of working with them.

A *source location* in Clang is an opaque pointer into the source code managed by a *source manager*. Usually information about a *source location* is retrieved by providing it as an argument to a function of the corresponding *source manager*. The function `isMacroID()` is the only interesting function existing directly in *source location*. It can be used to check whether the given *source location* represents a location inside a macro expansion. A *source range* is a pair of *source locations* representing the beginning and the end of the range respectively.

The *source manager* provides functionality for finding the location where a certain source range is expanded to (expansion range) or the location of the source text of a token (spelling location). It is important to note that there exist two forms of such queries: Plain ones (for example `getExpansionRange`) and immediate ones (for example `getImmediateExpansionRange`). The difference between the two is easily understood by using an example on expansion ranges: For example the function `getExpansionRange` returns the range describing the terminal destination of where a source location is expanded to. `getImmediateExpansionRange` on the other hand just returns the very next expansion step location.

Source locations are commonly associated with Clang node properties (for example the source range of the name text of a declaration node). Clang nodes are separated in different categories and do not have a common base class. There are two kinds of Clang nodes we treat specially. The statement type nodes inheriting from `clang::Stmt` and the declaration nodes inheriting from `clang::Decl`. Those kinds of nodes are used in mapping between the Clang and Envision ASTs. Statement and declaration nodes have a variety of *source ranges* associated with them which vary depending on subtype. The

following list compiles a few of the most interesting *source range* and *source location* queries on such nodes:

- All statement and declaration nodes have a member called `getSourceRange()` returning the smallest *source range* encompassing itself and all subnodes.
- Named declarations (`clang::NamedDecl`) have a member called `getLocation()` returning a *source location* pointing to the beginning of the name.
- Operator type nodes have a member called `getOperatorLoc()` providing the *source location* of the operator symbol.

Another category of Clang nodes are types inheriting from `clang::Type`. These nodes often do not have *source location* information. Instead they point to the declaration of the type. This can be problematic when reconstructing macros because the information it provides is not the information we need. In that case we are looking for a larger piece of code that at least contains the source text belonging to the type node and then find the relevant part using the node type and node context information (see section 4.5).

6.2 Mapping between Clang and Envision ASTs

Unlike Clang, all nodes in Envision have a common base class. Since it would make the implementation cleaner if we could store them in one map we do not store the actual Clang nodes when mapping between Clang and Envision ASTs. Instead we only store the *source range* of Clang nodes because that is all the information we are going to need in the macro import system (with the exception of the *lexical transformation* component). The *source range* used for this is dependent on the type of the Clang node and the appropriate *source range* is extracted when adding an entry to the map.

6.3 Lexical transformation

6.3.1 Identifier concatenation and stringification

Clang preprocesses the code before generating an AST. As a consequence all identifiers are already concatenated and all stringification operations are already performed. The *source locations* of nodes originating from such constructs thus point to an instance where the concatenation or stringification already took place. We have to be able to handle such cases specifically by checking whether a given *source location* originated from a concatenation or stringification. To determine whether a given *source location* originated from a concatenation or stringification we do the following:

1. Check whether the *source location* points into a macro expansion. If it does not point into a macro expansion it cannot originate from an identifier concatenation or stringification.

2. Calculate the immediate expansion (see section 6.1) of the location to check. In the following we call this *immediate expansion*.
3. Check whether the beginning of the immediate expansion range (see section 6.1) of the location to check equals the beginning of the *immediate expansion* we computed previously. If it does then it cannot be originating from an identifier concatenation or stringification because there would be an intermediate expansion step for those operations. If the two locations were not equal then the *source location* must point into an identifier concatenation or stringification.

6.3.2 Unexpanded spelling

When calculating the source text for an AST node for *lexical transformation* (see section 4.5) we are interested in the source text that was not yet expanded in any way (not even preprocessed). We call this particular spelling the *unexpanded spelling* of a node.

Given a source range *SR* its *unexpanded spelling* is calculated as follows:

1. Check whether the beginning of *SR* is part of a concatenation or stringification. If yes then use the beginning of the immediate expansion range (see section 6.1) of the beginning of *SR*.
2. Check whether the end of *SR* is part of a concatenation or stringification. If yes then use the end of the immediate expansion range (see section 6.1) of the end of *SR*.
3. Use the calculated beginning and end *source locations* as input to the standard `getSpelling` function of `ClangHelper` which returns the source text for a *source range*.
4. Clean up the result. It might contain trailing line separation or whitespace characters determined by the form of the replacement text specified in the `#define` directive of the macro.

6.4 MacroImporter Components

This section discusses the components of the macro import system. The main entry point for the macro import system is a class called `MacroImporter`. It has several child components and uses information provided by the existing C++ import system. Figure 6.1 shows the dependencies between the components and figure 6.2 shows the information flow from the existing C++ import system to the various components. The `ClangAstVisitor` is a component of the existing import system and the parent component of `MacroImporter`. It provides the `MacroImporter` with information on preprocessor, source manager and Clang nodes processed by it. The `TranslateManager` is also a component of the existing import system that handles certain kinds of Clang nodes

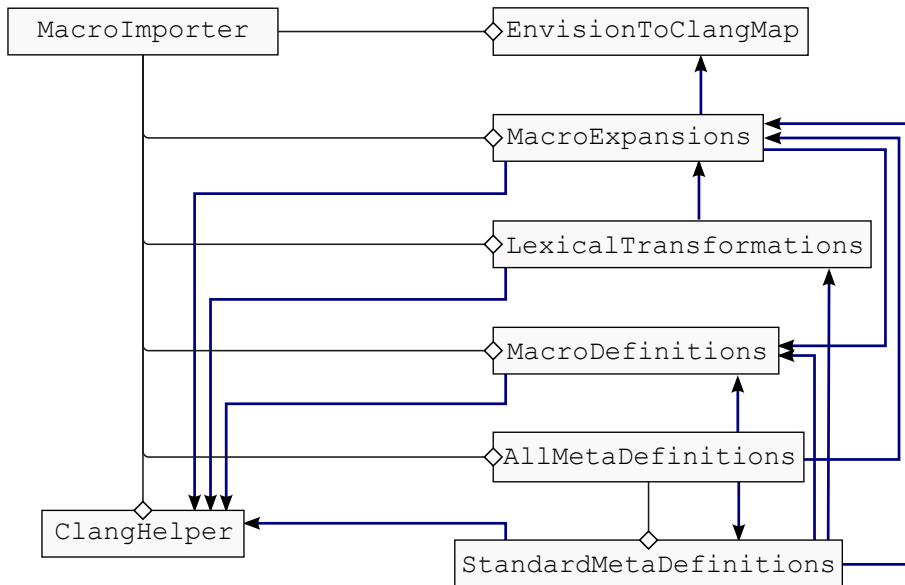


Figure 6.1: Components of the macro import system and their dependencies.

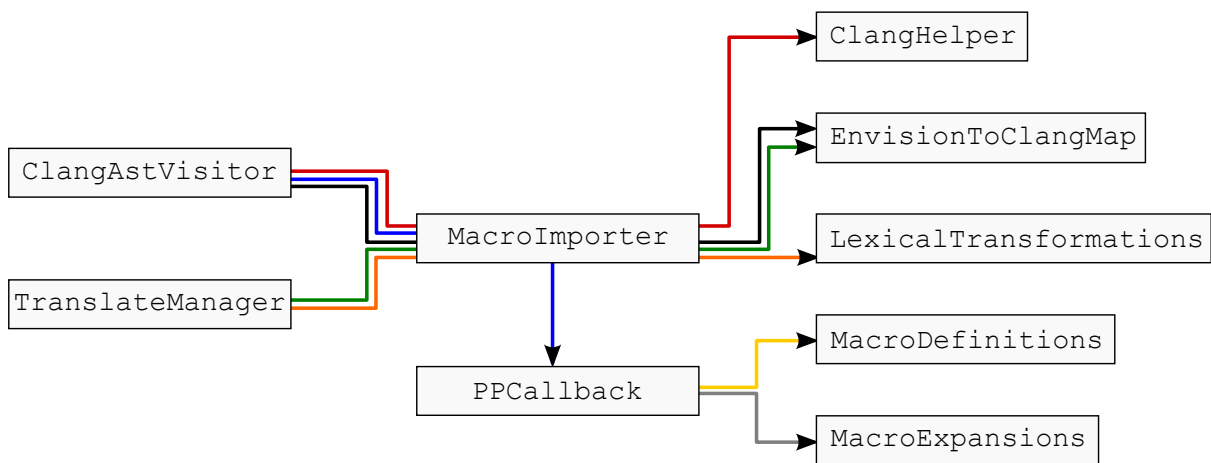


Figure 6.2: Information flow from the existing C++ import system to the macro import system components.

delegated by `ClangAstVisitor` and used to fill the `EnvisionToClangMap` discussed in the next paragraph.

The `EnvisionToClangMap` logically stores the mapping between the Clang and Envision ASTs. Because we only ever care about the *source range* for Clang nodes we do not store the actual Clang nodes here but only the relevant *source range* for a Clang node. What *source range* is relevant for a given Clang node is decided upon adding a mapping to this component.

The `ClangHelper` is a helper class that stores a reference to the preprocessor, language options and source manager. It provides helper functions to interact with the stored objects.

The `MacroDefinitions` component keeps track of all macro directives (`clang::MacroDirective`) and their names. Additionally it provides useful functions that only require information on either the name or the properties of macro directives. The state of this component is cleared after every translation unit.

The `MacroExpansions` component stores all macro expansions and provides functions to query information about expansions.

The `AllMetaDefinitions` component handles the creation of all *meta definitions*. Special cases like *bp-macro* specialization or *Y-Macro meta definitions* are processed directly by `AllMetaDefinitions` while the standard *meta definition* creation is delegated to a child component called `StandardMetaDefinitions`.

The `StandardMetaDefinitions` component creates and stores *meta definitions* that are not begin partial *meta definitions* or *X-Macro meta definitions*. It uses the information collected by the `LexicalTransformations` component to perform *lexical transformation* on the *meta definition* bodies.

The `LexicalTransformations` component calculates and stores a piece of source text for every Envision AST node in order to recover information on stringification or identifier concatenation (see section 4.5). The stored source text is then used for *lexical transformation* when creating a *meta definition*.

7 Evaluation

In this chapter we evaluate each one of the three major components of this thesis and discuss known issues.

7.1 Code generation framework

The code generation framework can represent all the macros used in Envision's C++ source code. In particular it can represent and create code for macros using stringification or identifier concatenation and in can represent X-Macros using *meta bindings*. It does not yet support code generation for the following two usages:

1. Expression to statement conversion is currently not supported because it was not used in the earlier stages of the project when the code generation framework was implemented. Adding support for such a use case requires minor changes in the `CodeGenerationVisitor`. In particular one would have to check whether one tries to replace a statement node with an expression and in that case wrap it in a `ExpressionStatement` node before replacing it. And the other way around when trying to replace an expression node with a statement one would have to check whether the expression to be replaced is indeed inside a node of type `ExpressionStatement` and replace the parent node with the new statement.
2. Arguments of type `Model::List` should be handled differently than other nodes. `Model::List` represents a forest of nodes in the code generation framework. This means instead of using the list node itself as expanded node one would have to use all its subnodes and use them like a collection. To enable treatment of `Model::List` like any other node one would have to introduce a special node type for splicing forests.

7.2 C++ macro import

The macro import system was evaluated manually by looking at the Envision AST created after importing the Envision C++ code. There are seven files which contain most of

the macros in Envision’s C++ code base. Tables 7.1 to 7.7 contain the results of the evaluation for each of the macro files.

All macros are recognized when importing provided they are at least used once in the imported code. Most of the remaining issues originate from missing *lexical transformations*. These can be resolved by handling more cases of *lexical transformations* by adding more regular expressions based on a node type and its context as discussed in section 4.5. Perhaps the hardest to resolve are the issues originating from implicit casts added by Clang during the conversion from C++ source code to Clang AST. This is an issue that affects all of the import system not just our contributions.

nodeMacros.h

Macro	Issues
NODE_DECLARE_STANDARD_METHODS	
DECLARE_TYPED_LIST	unsupported node: forward declaration
ATTRIBUTE	
PRIVATE_ATTRIBUTE	
COMPOSITENODE_DECLARE_STANDARD_METHODS	
DECLARE_EXTENSION	
SET_EXTENSION_ATTR_VAL	lexical transformation missing
SET_ATTR_VAL	lexical transformation missing
DEFINE_TYPED_LIST	lexical transformation missing
NODE_DEFINE_TYPE_REGISTRATION_METHODS_COMMON	unsupported node: lambda expression lexical transformation missing
COMPOSITENODE_DEFINE_TYPE_REGISTRATION_METHODS_COMMON	unsupported node: lambda expression lexical transformation missing
NODE_DEFINE_TYPE_REGISTRATION_METHODS	
REGISTER_ATTRIBUTE	
REGISTER_EXTENSION_ATTRIBUTE	lexical transformation missing
ATTRIBUTE_VALUE_CUSTOM_RETURN	lexical transformation missing
ATTRIBUTE_VALUE	lexical transformation missing
PRIVATE_ATTRIBUTE_VALUE	lexical transformation missing
EXTENSION_ATTRIBUTE_VALUE	lexical transformation missing
COMPOSITENODE_DEFINE_EMPTY_CONSTRUCTORS	
COMPOSITENODE_DEFINE_TYPE_REGISTRATION_METHODS	
NODE_DEFINE_EMPTY_CONSTRUCTORS	
DEFINE_EXTENSION	
NODE_DEFINE_TYPE_REGISTRATION_METHODS_WITH_DEFAULT_PROXY	not used
EXTENSION_PRIVATE_ATTRIBUTE	not used
EXTENSION_PRIVATE_ATTRIBUTE_VALUE	not used
EXTENSION_ATTRIBUTE_VALUE_CUSTOM_RETURN	not used
COMOSITENODE_DEFINE_TYPE_REGISTRATION_METHODS_WITH_DEFAULT_PROXY	

Table 7.1: Macro import evaluation: `nodeMacros.h`

typeIdMacros.h

Macro	Issues
DECLARE_TYPE_ID_COMMON	
DECLARE_TYPE_ID_BASE	
DECLARE_TYPE_ID	
DEFINE_TYPE_ID_COMMON	Clang implicit cast issue
DEFINE_TYPE_ID_DERIVED	
DEFINE_TYPE_ID_BASE	

Table 7.2: Macro import evaluation: typeIdMacros.h

attributeMacros.h

Macro	Issues
ATTRIBUTE_OOP_NAME_NOSYMBOL	
ATTRIBUTE_OOP_NAME_SYMBOL	
ATTRIBUTE_OOP_ANNOTATIONS	
REGISTER_OOENAME_NOSYMBOL_ATTRIBUTE	
REGISTER_OOENAME_SYMBOL_ATTRIBUTE	

Table 7.3: Macro import evaluation: attributeMacros.h

itemMacros.h

Macro	Issues
ITEM_COMMON_CUSTOM_STYLENAME	
ITEM_COMMON	
ITEM_COMMON_DEFINITIONS	

Table 7.4: Macro import evaluation: itemMacros.h

shapeMacros.h

Macro	Issues
SHAPE_COMMON_CUSTOM_STYLENAME	
SHAPE_COMMON	
SHAPE_COMMON_DEFINITIONS	

Table 7.5: Macro import evaluation: shapeMacros.h

StandardExpressionVisualizations.h

Macro	Issues
BEGIN_STANDARD_EXPRESSION_VISUALIZATION_BASE	lexical transformation missing
BEGIN_STANDARD_ENUMERATION_EXPRESSION_VISUALIZATION	lexical transformation missing
EXPRESSION_PART	lexical transformation missing
PREFIX	
OPERAND	
POSTFIX	
INFIX	
WRAPPED_OPERAND	
INFIX2	
BEGIN_STANDARD_EXPRESSION_VISUALIZATION	
BEGIN_STANDARD_FLAG_EXPRESSION_VISUALIZATION	
SHAPE_COMMON_DEFINITIONS	

Table 7.6: Macro import evaluation: StandardExpressionVisualizations.h

StandardExpressionVisualizations.cpp

Macro	Issues
PREINPOSTFIX	lexical transformation missing
PREFIX	
OPERAND	lexical transformation missing
POSTFIX	
INFIX	
WRAPPED_OPERAND	lexical transformation missing
INFIX2	

Table 7.7: Macro import evaluation: StandardExpressionVisualizations.cpp

8 Future Work

In this chapter we discuss possible future work on the subject.

8.1 Known issues

This section contains known issues of the existing design or implementation.

8.1.1 Invalidating code generation cache

The cache used for code generation is never invalidated as of the end of this thesis. Invalidating a cache holding outdated code is mandatory for analysis working with generated code to function properly.

The cache has to be invalidated if any of the child call *meta definitions* or the *meta definition* of the cache's *meta call* change. Additionally the cache has to be invalidated if any of the parent *meta calls* or the cache's *meta call* change their parameters.

8.1.2 Merging context declarations

Currently all the *meta calls* inside a *top level meta call* also generate code with their respective *context declaration*. If the resulting tree was used then not only the first node would be a *context declaration* but there would be a child node that is a *context declaration* for every child *meta call*. This can interfere with analysis components and might not be the result one expects to see when visualizing the generated code.

By means of merging child *context declarations* into the parent one could create a cleaner result that is more easily analyzable and more convenient to look at. The process of merging is non trivial. Similar to cloning of composite nodes one could potentially merge two composite nodes of the same type. Additional complexity is added to the problem by the requirement of detecting logically equivalent declarations in both trees and merging them into one declaration in the result.

8.2 Finalizing the import-export cycle

Apart from the known issues which have to be fixed to make Envision self-hosting there are a few other things that need to be improved or implemented:

- **Unsupported nodes:** The existing import system does not yet fully support importing all types of Clang nodes. The macro import system does not yet support lambda expressions and forward declarations.
- **Exporting declarations:** Currently Envision's C++ source code uses macros for this. We intend to enable to set an export flag for a declaration node in Envision's tree model.
- **Other preprocessing directives:** Apart from macros there are other preprocessor directives for example directives for conditional compilation. There is currently no support in the code generation framework or the import for such features.
- **Qt framework and implicit Clang nodes:** The Qt framework used in Envision as well as Clang itself introduce code elements in the imported code that we ideally do not want to have since they are only implicitly added and not present in Envision's C++ source code.
- **Export to C++:** We have to split code into header and source file and print C++ code.
- **Special macros:** Macros such as `__LINE__` or `__FILE__` have special behavior and would have to be realized in the form of *predefined meta definitions*.
- **Stringification:** For stringification we currently use the binary plus operator to indicate string literal concatenation. We have to use a separate operator and implement code generation framework logic which actually concatenates all the string literals surrounding the new operator to a final string literal.

8.3 Further extensions

8.3.1 Support QMake

The QMake build system is used to compile the C++ code base of Envision. This means that when self-hosting Envision should be aware of QMake and make use of options provided by QMake (for example to report error messages when compiling). The QMake build process uses QMake files listing files to be compiled so when changing Envision's code by adding or removing something that is exported to a file (for example a class) then the QMake files have to be modified to account for the changed element.

8.3.2 Preserve documentation

Documentation in Envision is not limited to textual comments. The user can choose from a variety of elements (image, table, internet resource, etc.) to annotate code. This information should be retained during an import-export cycle round trip. One way to achieve this is to use special syntax in the form of C++ comments referencing more complex resources when exporting code to C++. Complementary one has to extend the current import system to process comments when importing code from C++ and use the special syntax and referenced resources to reconstruct the original documentation.

8.3.3 Improve library support

When using a code editor features such as jumping to the definition of a declaration or auto-completion are very important to enable efficient development. Those features are also useful when working with external libraries. To provide such functionality we would have to process library headers when importing a C++ project. This can be very challenging due to libraries having potentially a lot of macros in them which may have forms that are not correctly handled by the current macro import system (*non-syntactic macros* with a structure different from the ones used in Envision). However typically we do not care about the macros in libraries. Instead we can use the expanded code which contains all the symbols used in the client application.

8.3.4 Intermediate language

Near the end of this thesis we identified an alternative approach which might make it easier to make Envision self-hosting. The idea is to use an intermediate language that uses a text format but closer to Java and Envision in structure. Allowing for easier mediation between said intermediate format and Envision's tree model as well as enabling the use of git for version control. The code generation framework is still necessary independently of this idea. The initial import from C++ has to be done either way and potentially multiple times until the intermediate language is working properly. The intermediate language would lack a compiler so exporting to C++ to use the C++ compiler for running modified code would still be required.

9 Related work

9.1 Macro and code generation systems

Syntactic macro systems reach back to Lisp[11] where quotation (a " ' " in front of a common Lisp expression) could be used to indicate that one wants to use the syntactic form of an expression, instead of the evaluation, to build new expressions. Other programming languages, such as Nemerle, continued evolving quoting and syntactic macro systems. Nemerle is a functional language using .NET and closely resembles the syntax of Java or C#. Its meta-programming-system uses Nemerle itself as meta language[9], supports quasi quotation and provides hygienic macros. The Scala macro system[2] was inspired by the meta-programming system of Nemerle. Their ideas on quasi quotes and splicing inspired the design of our code generation framework.

Intentional Programming[3] aims to express intentions instead of imperatively describing the way a result is obtained. The idea of aiming to express intentions inspired the way we are handling X-Macros in our code generation framework and lead the way of developing the concept of meta bindings.

9.2 C++ Preprocessor

The presence of preprocessor directives hindered development of refactoring tools for C++ for quite some time. In their work on rejuvenating C++ programs through demacrofication[6] A. Kumar et al. describe ways of refactoring certain kinds of preprocessor directives. They aim to use newer language features like templates to replace former preprocessor macros. In the earlier stages of this project we considered rewriting macros to templates but due to the use of stringification and identifier concatenation as well as the fact that macros are evaluated before the code we were unable to do so. The macro reconstruction we perform now is very similar to refactoring the code of the expanded program. A. Kumar et al. suggest a way of tackling syntactically incomplete macros in their analysis by grouping up following code until the block is syntactically complete again and perform refactoring on such syntactically complete blocks instead.

Our work ended up heavily targeting Envision's code. We informed ourselves of common usages of the C++ preprocessor[5] in other applications to judge the restrictiveness of

some design decisions targeting the preprocessor use in Envision's code specifically.

10 Conclusion

We came several steps closer to our goal of making Envision a self-hosting programming environment. In this thesis we designed and implemented a code generation framework in order to represent macros in Envision when importing code from C++. Furthermore we designed and implemented an additional stage in the existing C++ import system to enable macro import by reconstructing them from expanded code using preprocessor information provided by Clang. Finally we started the export part of the import-export cycle by adding the dependency analysis needed to enable modularity when exporting to C++. The issues in existing components as well as time shortage prevented us from completing our goal but we provide guidance for future work.

10 References

- [1] D. Asenov and P. Müller. Envision: A fast and flexible visual code editor with fluid interactions (overview). In *Visual Languages and Human-Centric Computing (VL/HCC)*, pages 9–12, 2014.
- [2] E. Burmako. Scala macros: Let our powers combine!: On how rich syntax and static types work with metaprogramming. In *Proceedings of the 4th Workshop on Scala, SCALA '13*, pages 3:1–3:10, New York, NY, USA, 2013. ACM.
- [3] K. Czarnecki and U. W. Eisenecker. *Generative Programming*, chapter Generators, pages 151–169. 2000.
- [4] C. Elliott, V. Vijayakumar, W. Zink, and R. Hansen. National instruments labview: a programming environment for laboratory automation and measurement. *Journal of the Association for Laboratory Automation*, 12(1):17–24, 2007.
- [5] M.D. Ernst, G.J. Badros, and D. Notkin. An empirical analysis of c preprocessor use. *Software Engineering, IEEE Transactions on*, 28(12):1146–1170, Dec 2002.
- [6] A. Kumar, A. Sutton, and B. Stroustrup. Rejuvenating c++ programs through demacrofication. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 98–107, Sept 2012.
- [7] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The scratch programming language and environment. *Trans. Comput. Educ.*, 10(4):16:1–16:15, November 2010.
- [8] S. Pokress and J. Veiga. Mit app inventor: Enabling personal mobile computing. *arXiv preprint arXiv:1310.2830*, 2013.
- [9] K. Skalski, M. Moskal, and P. Olszta. Meta-programming in nemerle.
- [10] L. Vogel. C++ support in envision. Bachelor thesis, ETH Zürich, 2013.
- [11] P. H. Winston and B. K. Horn. *Lisp*. 1986.

Appendices

A Import guide

In this chapter we provide a step by step guide on how to import Envision. This is a more up-to-date version of the guide written by Lukas Vogel in his bachelor's thesis report[10].

1. Clone the Envision repository to some path *PATH*.
2. In *PATH/Core* the `common.pri` add the code from listing A.1.

```
1 QMAKE_CXX=clang++-3.7
2 QMAKE_CXXFLAGS_WARN_ON += -Wno-unused-private-field
3                               -Wno-inconsistent-missing-override
```

Listing A.1: Options to be added in `common.pri` to enable importing.

3. In *PATH* run `qmake -r CONFIG+=debug`.
4. In *PATH* create and run a shell script with the following contents shown in the code of listing A.2.

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

Listing A.2: Script used for importing.

5. In the directory of your working copy of Envision change directory into `CppImport/test`, locate the `testSelector` file and add `spath:REL_PATH` where *REL_PATH* is the path to *PATH* relative to the `CppImport/test` directory. Make sure that all lines before `spath:REL_PATH` start with a `#`-sign.
6. Run Envision with the `--test cppimport` argument to start importing.



Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Self-hosting the Envision Visual Programming Environment

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):

Lüthi

First name(s):

Patrick

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

Zürich, 11.11.2015

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.