technische universität
dortmund

Bachelor's Thesis

AspectClang: Moving AspectC++'s Weaver to the Clang C++ Front End

Benjamin Kramer
September 6, 2013

Adviser:
Prof. Dr.-Ing. Olaf Spinczyk
Dipl.-Inf. Christoph Borchert

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl Informatik 12
Arbeitsgruppe Eingebettete Systemsoftware
http://ess.cs.tu-dortmund.de

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 6. September 2013

Benjamin Kramer

Abstract

The source-to-source translator (weaver) that translates code using the aspect-oriented C++ language extension *AspectC++* into standard C++ was built upon a C++ parsing framework called PUMA. The framework is developed alongside the weaver; it is showing its age and the complexity of the C++ language drives up the upkeep of PUMA. This thesis explores the replacement of PUMA with the widely used *Clang* C++ front end. Clang is used in industry and academia and shares many design goals with PUMA, making it a viable successor. A port of the weaver from PUMA to Clang, using the code name *AspectClang*, is developed to show the feasibility of a Clang-based weaver and to evaluate its features in the context of AspectC++ weaving. The result is vastly improved conformance with the C++ language specification and the availability of C++11 support in the parser will benefit the AspectC++ community in the long term.

Das Übersetzungsprogramm (Weaver), das aus *AspectC++*, einer aspektorientierten Spracherweiterung für C++, normalen C++-Quelltext erzeugt baut auf der Programmbibliothek PUMA auf. Die Bibliothek wird zusammen mit dem Weaver entwickelt und zeigt Alterungserscheinungen durch den hohen Aufwand um die Kompatibilität mit der komplexen C++-Sprache zu wahren. Diese Arbeit untersucht die Austausch von PUMA durch den weit verbreiteten C++-Parser *Clang*. Clang wird sowohl in der Industrie als auch in der Forschung eingesetzt und hat viele Ähnlichkeiten mit PUMA, was es einen würdigen Nachfolger macht. Im Rahmen dieser Arbeit wurde ein Weaver auf Clang-Basis mit dem Namen *AspectClang* entwickelt um die Machbarkeit des eines Clang-basierten Weavers zu untersuchen und Clangs Funktionen im Kontext der *AspectC++*-Sprache kritisch zu betrachten. Die Ergebnisse zeigen eine deutlich verbesserte Kompatibilität mit dem C++-Sprachstandard und die Verfügbarkeit von C++11 im Parser wird *AspectC++* auf lange Sicht nützen.

Contents

1 Introduction

AspectC++ is an aspect-oriented extension for the C++ programming language. The exntension is implemented with a source-to-source translator which transforms code and aspects into standard C++. The output can be used with any modern C++ compiler. This transformation process is called aspect weaving.

Since the beginning, the AspectC++ weaver `ac++` was built upon a C++ parsing framework called PUMA that is developed in tandem with the weaver. Writing and maintaining a C++ parser requires a significant amount of work and the upkeep that has to be paid for PUMA is quite high.

This thesis explores the replacement of PUMA with the widely used *Clang* C++ front end. Clang is widely used in industry and academia and shares many design goals with PUMA, making it a viable successor. A port of the weaver from PUMA to Clang, using the code name *AspectClang*, is developed to show the feasibility of using Clang and to evaluate its features in the context of AspectC++ weaving.

1.1 Motivation and Goals

There are several issues with the current implementation of the `ac++` weaver. Some of them prevent usage of aspects while using certain C++ libraries at the same time; others are less technical in nature or just inconvenient for users of AspectC++. While changing the parser underneath the `ac++` weaver is probably the most severe change in weaver code since its inception, it would solve many of the problems in one go and proof the weaver for coming challenges in the environment surrounding the C++ programming language.

Maintenance

Many problems with the AspectC++ weaver are actually rooted in the PUMA framework. Maintaining an entire C++ front end is not an easy task and drains development resources that could be spent improving the language itself. Using Clang instead of PUMA would remove most of the work needed to keep the parser running as Clang is maintained by an external community. With the freed-up resources, the team developing the weaver can focus on advancing the AspectC++ language.

Speed

Weaving aspects into a couple of C++ files in a large project can often take longer than compiling the entire project without using the extension. A part of this problem is due to the way the weaver handles source code with multiple parsing phases but also because PUMA has to make trade-offs on performance as it struggles to achieve correctness. Clang lists having a fast parser as an explicit design goal and was heavily optimized [1]. It is possible that a faster parser would significantly speed up the weaving process, reducing the time spent waiting for the build to complete.

Support for C++

PUMA does not support the C++ language completely. This is often not an issue with code written for AspectC++ from the ground up but with libraries used commonly with C++ code. Due to the way C++ is compiled, the aspect weaver has to parse all the headers used, which may contain C++ constructs PUMA does not understand in their entirety. Bogus parser errors are the result, necessitating workarounds that should not be needed. Clang claims complete support for the C++ standard so issues of this kind should become a thing of the past.

C++11 and Coming Standards

There is very little support for features of the new C++11 standard in PUMA and improving it is a major effort given the current state of the project. Clang on the other hand was one of the first C++ compilers to claim full support for C++11 and it is likely that future standards will be implemented in a timely manner. The AspectC++ weaver could make use of the new features with only a small amount of additional implementation effort.

1.2 Thesis Structure

Following the introduction chapter, chapter 2 discusses the foundations of AspectC++, the PUMA parser framework and Clang. After this the usage of PUMA in the AspectC++ weaver is analyzed and Clang replacements for the used PUMA components are considered in chapter 3. This leads to chapter 4 which lays down a detailed picture of the work needed to move the weaver to Clang. Its subsequent implementation is discussed in the following chapter 5 and critically evaluated afterwards in chapter 6. There, Clang has to show if it can match the expectations on performance and conformance with the C++ standards. The concluding chapter 7 summarizes the achieved work and goes on to point out directions for the future.

2 Background

This chapter gives a high level overview of the critical parts that make up AspectClang, setting the stage for the following analysis, design and implementation chapters.

2.1 The AspectC++ Language

AspectC++ [2] is a language extension that adds support to the C++ language for writing code in an aspect-oriented manner. It is modeled after the AspectJ extension [3] for Java, sharing a similar design, with differences to accomodate for the distinct features of the underlying C++ language.

Design goals of *AspectC++* [4] are ease of use, in particular a familiar setting for programmers already proficient in AspectJ, and an easy integration into existing projects. Furthermore *AspectC++* should be strong in domains where C++ is already used for its specific feature set. This applies especially to the embedded systems programming and high performance computing areas. It is also the motivation for breaking with AspectJ by not providing extensions which make use of language features with significant runtime overhead, such as exception handling or run-time type information.

AspectC++ is implemented as a source-to-source translator, turning code using the extensions into regular C++ code that can be compiled with any modern C++ compiler. Source-to-source translation allows integrating *AspectC++* with existing C++-consuming tools and platforms with little effort. This translation process is called *weaving* in aspect-oriented programming jargon. The *AspectC++* weaver is also known by its command line invocation `ac++`.

The language extension is based on four basic concepts: pointcuts, advice, slices and JoinPoint API functions. In actual code, all of those constructs are contained in a top-level *aspect* which behaves similar to a C++ class, supporting both inheritance and abstract classes.

Pointcuts

*Pointcut*s are AspectC++'s way to address entities in the input source code. For this a domain specific language is used. Pointcuts can match names, similar to regular expressions.

Listing 2.1: Name pointcut that matches all member functions of the class C that return an int

```
pointcut ints() = "int C::%(...)";
```

A different type of pointcut can use a predefined set of expressions to address other constructs in the code. Name pointcuts and expression pointcuts can be combined with boolean operators.

Listing 2.2: Expression pointcut that matches all subclasses of *Queue*

```
pointcut queues() = derived("Queue");
```

Advice

Advices are the primary way to inject code in *AspectC++*. The actual position for the injection is specified by a *join point* which consists of a pointcut and a location fragment. They come in two flavors.

- *Execution join points* specify a piece of code that is woven before, after or around a function in the input source code.

  Listing 2.3: Execution advice to print a string every time the function *login* is entered.

  ```
  advice execution("void login(...)") : before() {
      cout << "Logging in." << endl;
  }
  ```

  For this kind of advice the weaver emits a wrapper function which adds the new code and then proceeds with calling the old code. The wrapper replaces the original function so it is called instead of the function.

- *Call join points* on the other hand do not affect the function itself but calls to the function.

  Listing 2.4: Call advice to print a newline before all calls to printf

  ```
  advice call ("% printf(...)") : before () {
    printf ("\n");
  }
  ```

  This requires modifying the code itself instead of just adding a wrapper, making the weaver's job more complicated.

Slices

Slices are used to modify classes by adding members to them. This is also the only way to add new functions to the output which can in turn trigger other advice to match on the introduced function, leading to recursive weaving.

Listing 2.5: Slice that adds a new integer member to all nested classes named *Nested*

```
slice class IntoNested {
  int new;
};

advice "...::Nested" : slice IntoNested;
```

Slices can also add new base classes to easily inject a whole class into an existing one.

Listing 2.6: Base class introduction which adds *NewBase* as a base class to all nested classes named *Nested*

```
advice "...::Nested" : slice class : NewBase;
```

JoinPoint API

In advice code a set of predefined functions can be used that provide information about the aspect or the matched function. It is also used to proceed with the original function from the advice code.

Table 2.1: Examples of functions available in advice code

Name	Action
`JoinPoint::Result`	Result type of the matched function
`JoinPoint::ARGS`	Returns the number of arguments of the matched function
`JoinPoint::signature()`	Returns the name of the matched function
`tjp->proceed()`	Calls the matched function

AspectC++ also provides introspection code to query meta data of the compiled program, such as the number of members in a class. This code is dynamically inserted during the weaving process.

A detailed analysis of the AspectC++ weaver can be found in chapter 3. For more information on the *AspectC++* language refer to the language reference on the *AspectC++* homepage. [5]

2.2 Puma

Developing a standard-compliant parser for the C++ syntax is just a nightmare. It is an extremely hard, tedious and thankless task. Additionally, to support a substantial set of join point types, an aspect weaver has to perform a full semantic analysis, and the semantics of C++ is even worse.

(Spinczyk, Lohmann, Urban [4])

The Puma C/C++ parser framework is the C++ front end used by the `ac++` weaver. It is also utilizing an aspect-oriented design [6] making it a user of AspectC++ itself. Puma is developed under the AspectC++ project umbrella and maintained by the same team as the weaver.

The framework itself contains both the classical components used to parse C++, including a scanner, preprocessor, parser and full semantic analysis. It also contains a powerful source code manipulation module which is used by the AspectC++ weaver to implement its source code transformations.

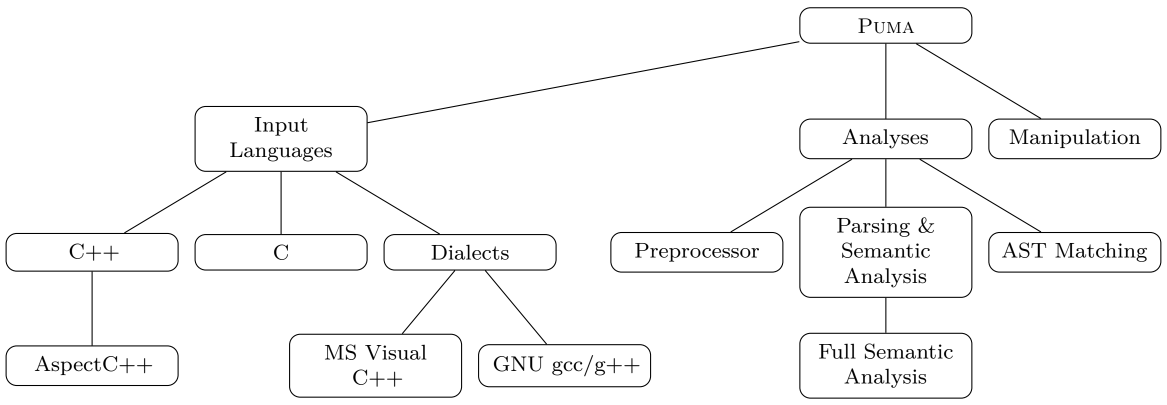

Figure 2.1: Overview of Puma's features. [6]

Language extensions are implemented using aspects that are woven into the parser code instead of adding the code directly to the parser. Among the parser extension aspects are *GNU GCC* extensions, *Microsoft Visual C++* extensions and special AspectC++ extensions. The AspectC++ extensions were used to parse the input aspects in the AspectC++ weaver but are no longer in use as the weaver now implements parsing of those constructs directly.

Sadly, Puma development has slowed down considerably in recent years. There are huge problems in standards compatibility, which have prevented adoption of AspectC++ for code bases depending on modern C++ libraries such as *boost*, which is known to be very challenging for C++ parsers and cannot be parsed by Puma.

It also never gained many users outside of AspectC++ [7] leading to failure in attracting new contributors. The focus on AspectC++ weaving determined the focus of Puma which does little analysis over the basic things needed by a weaver. For example access control is still missing [8]; private class members can be accessed without complaints.

While Puma supports C++03 reasonably well, the new C++11 standard is yet to be implemented. This is a major undertaking as some of the new features are very complex. For example the new `constexpr` feature [9] adds another Turing complete [10] language to C++. Implementing the new standard in Puma requires a significant amount of work and, given the current activity, is unlikely to happen any time soon.

Puma is open source software and available under the GNU GPL license, just like the `ac++` weaver.

2.3 Clang

Clang is a compiler front end for C, C++ and Objective C. It contains a unified parser for the three languages and implements many compiler extensions. The parser is tuned for fast compiles and excellent error messages [11]. Clang is also used as a base for more exotic C dialects such as CUDA [12] and OpenCL, which are also directly supported by the parser.

When Puma was implemented in the early 2000s there was no viable alternative around. Clang is a relatively new development, it was started in 2006 and became a serious competitor in the C++ parser landscape in 2010 [13].

It shares many design goals with Puma, both are designed as a set of libraries to be used by other tools and provide both the usual lexical and semantic analysis and also a sophisticated source code manipulation library.

Contrary to Puma, Clang is used to implement a complete compiler, paired with LLVM as a code generator. The resulting compiler is the default C and C++ compiler on FreeBSD [14] and Apple's OS X operating system [15]. This gives it a large backing in both corporate and academic communities leading to very mature support for the provided programming languages. The parser is exposed to a huge corpus of C and C++ source code through FreeBSD's *ports* system, giving it coverage on many open source code bases.

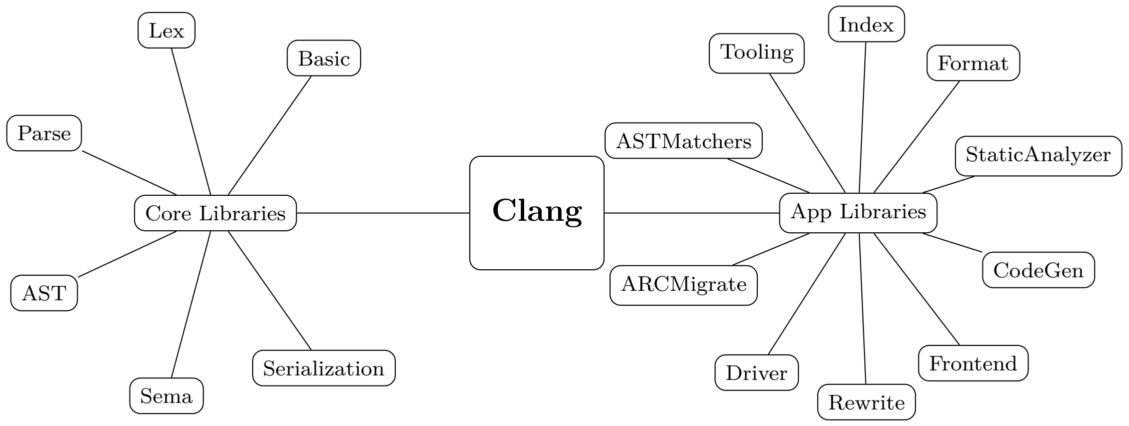

Figure 2.2: Overview of Clang's libraries.

Clang is a very large project and ships with a variety of libraries over the ones provided by PUMA. This includes

- Features for use in integrated development environments. This includes creating an index of the input source so it can be navigated in an editor. Clang also contains support for automatic code completion, which returns a list of valid identifiers at a given location in the source code. Another useful feature is the direct availability of *Doxygen* comments in the syntax tree.
- A static analyzer that can find errors by symbolically evaluating source code and showing code paths that may lead to unintended behavior or crash the program under certain circumstances.
- A GCC-compatible driver library that can parse and understand GCC command lines. This allows Clang to be used as a direct replacement for the GNU compiler without adjustment of build systems. AspectC++ currently relies on GCC to do its command line parsing; this could be replaced with the library and implemented in a cleaner fashion.
- A source code formatting utility which can easily adapt code to existing coding styles. It corrects line breaks and white space for a given input.
- Library support for automatic refactoring of C++ source code. An example is automatic migration of existing C++03 code to C++11, modifying code to use features such as range-based `for` loops and automatic code simplification with the new `auto` keyword.

The Clang project claims to have complete support for the C++03 and C++11 standards [16]. Experimental support for features likely to be included in the next revisions of the standard is also available. It is likely that this will continue and a Clang-based weaver would gain support for new C++ standards essentially for free.

Clang implements almost all GNU extensions and many Microsoft extensions are also supported. This is important for compatibility with existing code bases on Linux and Windows as even basic system header files make heavy use of compiler extensions. The extensions are directly implemented in the parser code; this approach is not as clean as PUMA's aspects and complicates the parser code.

The Clang source code is available under the very liberal *University of Illinois/NCSA Open Source License* which allows free redistribution and modification of the code, including usage in projects whose source code is not delivered with the binary executable files.

2.4 Other C++ Parsers

Of course PUMA and Clang are not the only C++ parsers available. The elephant in the room is GCC which originated in the GNU project. It is freely available under the GPL license but made design decisions which make decoupling the front end from the back end very complicated [17]. It also does not provide any support for source code manipulation.

A major commercial option is the EDG C++ front end which is used by many proprietary source-to-source translators. It is not freely available though, making it impossible to use it in a open source AspectC++ weaver.

There are other, smaller open source parsers such as Elsa and OpenC++. None of these options claim to have a complete implementation of C++ and would provide no advantage over PUMA.

2.5 Summary

This section explored the features of the AspectC++ language and showed the properties of the PUMA parser which the `ac++` weaver is built upon. The Clang front end was introduced, and its advantages over PUMA were pointed out. Clang and PUMA share many design goals, making Clang a good spiritual successor to PUMA while providing a much more complete and mature C++ parser and many additional features.

3 Analysis

In this chapter the differences between Puma and Clang in the context of their use in the AspectC++ weaver are analyzed.

3.1 Structure of the Weaver

The AspectC++ weaver can be seen as a pipeline taking both C++ source code and aspect code and emitting pure C++ code at the end. The transformation is divided into two phases.

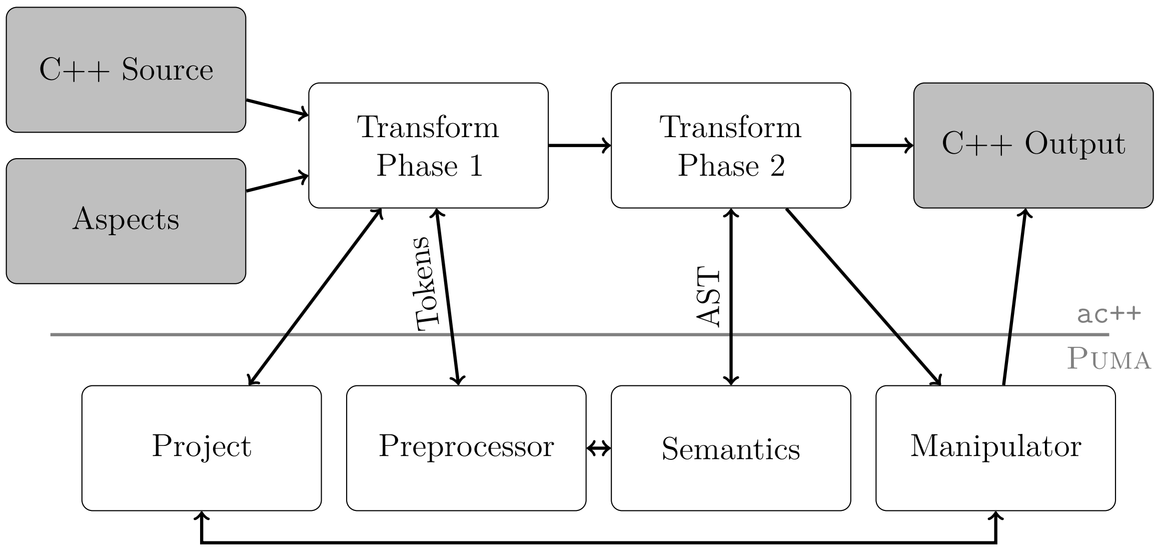

Figure 3.1: High-level overview of the AspectC++ weaver.

Phase 1 takes the input source code and tries to find AspectC++ constructs in them. To do this it asks a Puma component called the `Project` for input files and runs the preprocessor on the code. It then goes on to analyze the resulting tokens, looking for aspects. If it finds one, the aspect is stored on the side and removed from the source code which is subsequently fed into the next phase.

Phase 2 runs the full C++ parser on the output of phase 1. The resulting syntax tree is analyzed and broken down into a simpler representation. On this representation pointcuts are evaluated and code transformations are performed. The code transformation is provided by Puma's manipulator class. Results are written back into C++ files using path information from the `Project`, concluding the weaving step.

3.2 Use of Puma Components in the Weaver

To get a more detailed picture of the dependencies between AspectC++ weaver components and Puma components, the parser usage by individual AspectC++ classes was explored. Figure 3.2 shows which class in `ac++` uses which classes and functions from what sublibrary in Puma.

The used Puma libraries in particular:

aspects
This is a special library for the aspects woven into Puma. A dependency on this library means that a language extension is referenced.

basics
Utility classes used throughout Puma and AspectC++.

common
Components shared by all libraries.

cpp
Implementation of the C Preprocessor.

infos
Definitions for all classes that represent semantic information.

manip
Source code manipulation library.

parser
Contains the code to build syntax trees and extract semantic information from it. This component also covers sublibraries specific to C and C++ code, including a separate sublibrary for template instantiation.

scanner
The code that drives lexical analysis of the input files.

The basics and commons libraries are excluded from the graph because all AspectC++ components depend on it.

The following sections describe major components from all the libraries, their usage in AspectC++, and how they can be replaced with Clang equivalents if one exists.

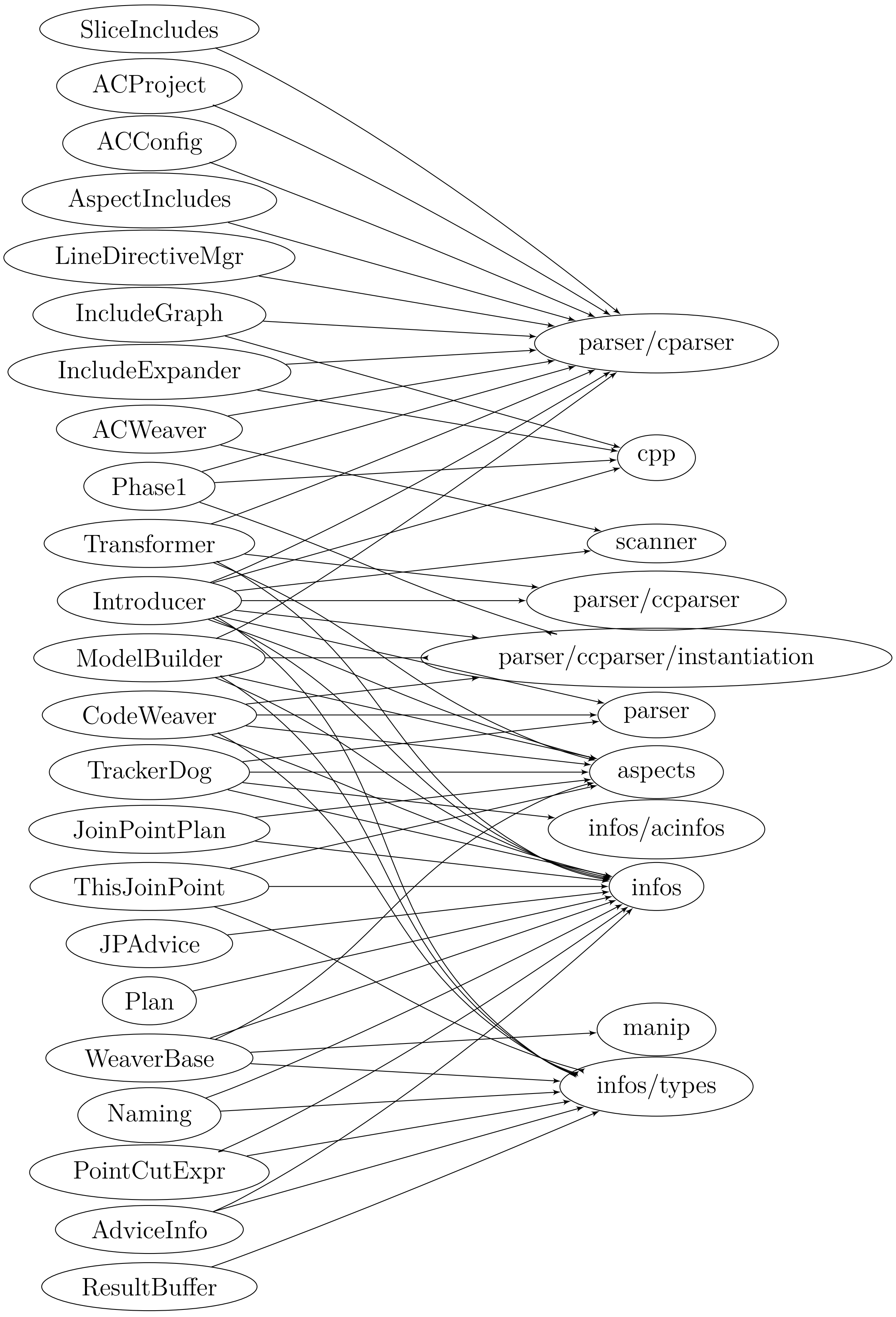

Figure 3.2: Dependencies from ac++ to Puma components

3.3 Project

Contained in Puma *library* ***common***.

Puma's `Project` class is referenced in almost every component of the weaver. It takes on three major responsibilities in AspectC++.

- Configuring the preprocessor. This includes parsing a set of predefined macros from a file that is provided by the `ag++` driver. `ag++` extracts the information from a GCC compiler invocation.

  In this configure phase the `Project` also takes care of enabling language extensions.

- Finding and managing file paths. `Project` contains essential information on whether a file belongs to it. This is used all over the weaver to determine if a file should be modified as part of the manipulation process. Knowing what files belong to the project avoids changing unrelated source code.

  This information also allows the `Project` to emit relative paths to be used in newly added include directives.

- Managing and saving modified files. The `Project` also manages the `Units` (see below) of all parsed files. When weaving is completed the modification are committed back to the `Project`. As a final step the modifications are written to the file system.

In Clang there is no direct equivalent to the Puma `Project` class. Some parts of its responsibility can be taken over by Clang functions though. Clang includes a GCC-compatible driver which can parse a `g++` command line and produce predefined macros, this can potentially replace the configuration phase.

As the Puma project currently provides a location to store the state of the preprocessor and the `Units`, a project for Clang would take ownership of Clang's source management facilities, namely `FileManager` and `SourceManager`. Clang's rewriter has a method to save modified files but it is incompatible with AspectC++'s phased approach with intermittent reparsing, so it cannot be used directly. Before parsing the input a second time the source files would be committed to the `SourceManager`. A separate saving step could fetch the textual data from the manager and store it into a file.

The path management functionality can be used unmodified with a Clang-based AspectC++ weaver. To remove the dependency on Puma the class could be stripped of Puma-specific methods and copied into the AspectC++ source tree. This is not a requirement for a working weaver utilizing Clang, which can still use Puma parts, but an important step for removing the dependency on Puma.

3.4 Preprocessor

Contained in PUMA *libraries* ***cpp*** *and* ***scanner****.*

The preprocessor is in charge of interpreting directives such as file includes and macro expansion. In modern C and C++ parsers, it directly interacts with the lexical analysis for performance reasons instead of being implemented as a separate source code manipulation step. Both PUMA's and Clang's preprocessors take an input file and return a stream of tokens which can be consumed by a semantic analysis step. PUMA provides a visitor mechanism to retrieve information on preprocessor directives. Clang uses a callback interface (`PPCallbacks`), through which the same information can be collected while the preprocessor runs.

In the AspectC++ weaver the preprocessor is run by phase 1 which directly consumes tokens and phase 2 where a full semantic analysis is performed. Since most of the interaction with the preprocessor happens in phase 1 this section focuses on this portion of the weaver.

Phase 1 runs the preprocessor and performs simple semantic analyses on the tokens, identifying classes and all syntactic extensions such as advice and pointcuts. The information on those extensions is stored on the side and the code is removed from the source files, making it invisible to later passes. This makes it possible to parse AspectC++ with a generic C++ parser that does not have to recognize any extensions.

The data stored about aspects is mostly parser-independent with the exception of class slices which are stored as `Units` for consumption by phase 2.

To get a deeper insight on the differences between PUMA and Clang in the preprocessing step a test program was written which runs both preprocessors and prints out the tokens found. It was then executed on various source files from PUMA and AspectC++ for manual inspection of the differences.

- PUMA intentionally does not implement some hard to parse language features such as the GNU `__attribute__` extension. The tokens resulting from those constructs are silently dropped by PUMA's lexical analysis pass. This is not a problem for AspectC++ as the extensions do not affect weaving.
- In the Clang output, tokens derived from macro expansions carry source locations differing from the locations PUMA uses. This has an effect on error messages emitted by the AspectC++ weaver.
- Minor bugs were found in PUMA's preprocessor that since have been fixed. For example, include directives were ignored if preceded by a comment on the same line.
- On the tested files Clang's preprocessor was up to four times faster than PUMA's without significant differences in the parsed tokens. While this is not representative for a full C++ parser it will hopefully have an effect on the performance of AspectClang.

3.5 Units and Tokens

Contained in PUMA *library* ***common***.

One of the fundamental classes used in PUMA's parser is the `Unit`. A `Unit` stores a list of tokens representing a single file in the input. Another use case occurs in macro instantiations where artificial `Units` are created to store the expanded tokens.

The AspectC++ weaver also creates such artificial `Units` to store slices during phase 1, which are eventually inserted into the output in phase 2. Those `Units` contain special marker tokens to indicate insertion positions for names in phase 2. They have no direct counterpart in the original input.

In Clang there is no equivalent concept to a `Unit`. The `SourceManager` class manages the content of input files and all tokens reference data stored in the manager. This means that phase 2 cannot reference any tokens from phase 1. This poses problems when porting the code from PUMA to Clang; a strict separation of tokens between phase 1 and phase 2 has to be maintained to avoid mixing state from both phases. The `Units` used during source-to-source translation can be represented as simple lists of tokens, we will only lose comments and formatting in the injected slices that way.

Tokens also play an important role in phase 2 as they are providing the locations where code transformation should occur. This is very different from Clang's approach where tokens are basically hidden after semantic analysis. On the other hand Clang's `SourceLocation` object provide a very fine-grained way to address locations in the source code. It is an opaque object that represents a byte offset. It can be translated into a file/line/column triple by the `SourceManager` or used to address a position for code manipulation. Any code using tokens in the weaver phase 2 has to be changed to use `SourceLocations` instead. Sometimes this entails managing more state; a token has two addressable positions, before and after, while a location is just a single point in the code with zero length.

Listing 3.1: Example C++ code with multiple calls in one position

```cpp
struct C {
  C(int);
  operator int();
};

int abs(int);

void f(C &c) {
  // Call advices for all three functions above could match
  // here; adding multiple insertions at the same positions.
  c = abs(c);
}
```

For call advice it is possible that multiple code insertions happen at the same location and have to be performed in the right order. The current AspectC++ weaver addresses this problem by introducing ephemeral marker tokens which are only used to denote a position. The code manipulation functions of Clang does not cannot use marker tokens but it allows inserting code before or after a `SourceLocation`, moving existing insertions in the specified direction. This involves complex changes to the weaving code as it has to insert changes in the right order. Marker tokens did not put any constraints on the ordering.

3.6 Parser

Contained in PUMA *library* ***parser***.

A key part of a C++ front end is the parser which performs semantic analysis on the stream of tokens and makes the extracted information available in some way.

The parser is primarily seen as an opaque step producing data structures to be used for further processing of the source code. However, there are critical situations during AspectC++ weaving that require interacting with the parser directly. It can be best explained with the problem of recursive introduction. Take for example a simple segment of AspectC++ code using class slices.

Listing 3.2: Example code for nested insertion

```
advice "Foo" : slice struct { struct Bar { }; }; // nested class
advice "%::Bar": slice struct { int _in_bar; };

struct Foo { };
```

The pointcut matches `struct Foo` so in a first step the nested class is introduced by copying the body of the first slice into the structure.

Listing 3.3: First iteration of insertions

```
struct Foo {
  struct Bar { };
};
```

The nested class introduction now would make the second pointcut match. This requires semantic information about the introduced nested class to be available at the time the slices are expanded. If it is provided the member will be introduced triggering a new iteration of code introduction.

Listing 3.4: Second iteration of insertions

```
struct Foo {
  struct Bar {
    int _in_bar;
  };
};
```

Now no further pointcuts match and the weaver can begin processing execution and call advices as those advice types cannot cause recursive introduction of code and do not need special attention in the parser.

A similar problem can also happen with code that is not parsable without adding the slice code and with the AspectC++ introspection feature. Introspection allows, among other things, querying the number of members in a class. The number of members will change if new members are introduced.

Injecting pieces of text during the parser run is not something commonly supported by C++ parsers. PUMA provides a callback interface at certain points in the parser that calls into `ac++` so the weaver can inject code. The weaver then runs the parser on the injected code and updates both its view of the source code and the PUMA'S data structures. Later the code is also inserted into the resulting files with a textual manipulation action.

In Clang this approach is not feasible as the AST data structure containing all the semantic and syntactic information is not designed to be modified in such a way. To solve this problem with the Clang parser a different method has to be found.

3.7 AST

Contained in PUMA *libraries* ***infos*** *and* ***parser****.*

The semantic information emitted by the parser is the reason why a full C++ front end is needed for an AspectC++ weaver in the first place. It contains the knowledge needed to perform all the transformations happening to the source code when being transformed from AspectC++ to plain C++.

The primary consumer of the information is the `ModelBuilder` which further processes it and extracts facts needed for the source transformations. It is split into two parts, one analyzing semantic information and another class called the `TrackerDog` traversing the syntax tree. This reflects the design of the data structures in PUMA.

In PUMA, the parser builds two independent trees for a translation unit. The `SemDB` contains high-level information on classes and functions, abstracting away many details of the C++ syntax. The nodes of this tree are all derived from the `CObjectInfo` class. On the other hand there is the actual syntax tree containing objects derived from `CTree`. The class hierarchies for a class definition can be seen in figure 3.3. There are pointers on both sides to get to the corresponding object in the other tree.

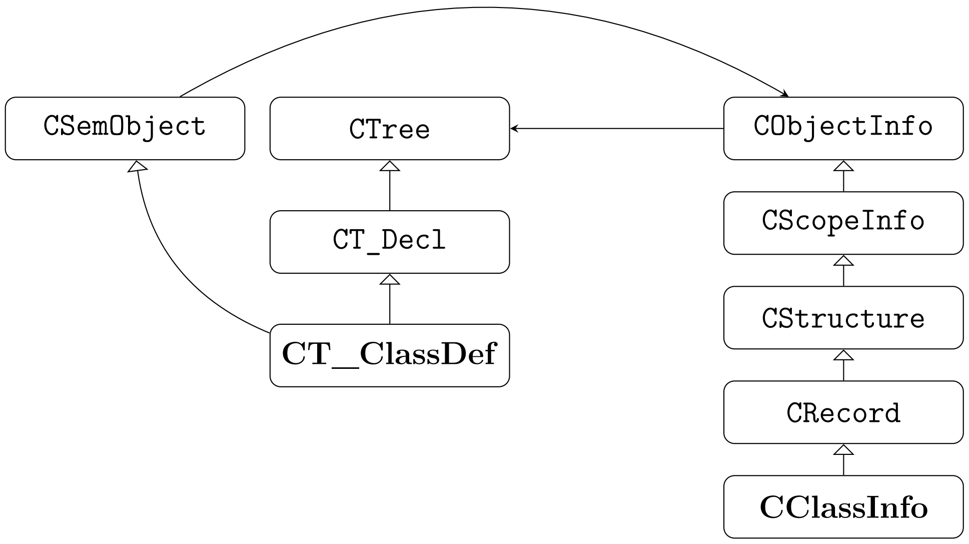

Figure 3.3: Class diagram for syntactic and semantic objects describing a class definition in Puma

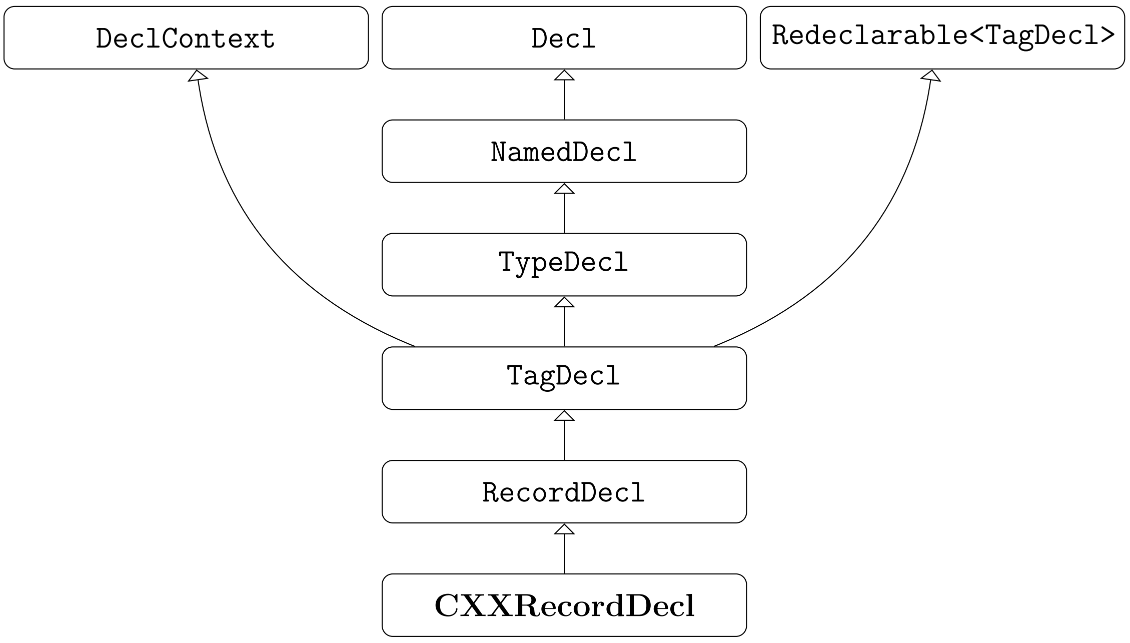

Figure 3.4: Class diagram for objects describing a class definition in Clang

After the semantic database is built, `ModelBuilder` walks the tree and adds information on functions and classes that are candidates for code manipulation. The `TrackerDog` visits the syntax tree of functions, looking for calls that are also added to the model. They will be used for call advice.

Clang pursues a different approach here. There is only one large data structure called the AST (**A**bstract **S**yntax **T**ree) which contains all the information. AST nodes are either derived from the `Decl` or the `Stmt` class hierarchies. Having only a single tree has the advantage that the AST nodes always carry detailed information on what piece of source code they are derived from. On the other hand it adds the burden of dealing with the intricate details of C++ syntax to all users that is invisible in PUMA's semantic database approach. For example a class in C++ can have multiple forward declarations and one definition. When encountering a class declaration it is the consumer's responsibility to pick the desired one. A class information object in PUMA can always be asked for the defining object and provides a simple interface for it.

Figure 3.4 shows the class hierarchy for a class declaration in the Clang AST. It uses multiple inheritance to add information about redeclarations to the classes. A Clang `DeclContext` roughly represents a scope in PUMA terminology and can be used to walk scopes upwards, for example member function to class definition to namespace to file scope.

Clang's AST is not a standalone data structure but designed to be used with an `ASTConsumer` class which provides virtual functions that are called for all top-level declarations. This is different from PUMA where a tree can be traversed after it is completely parsed. This will lead to structural changes in the weaver's `ModelBuilder`.

3.8 Code Manipulation

Contained in PUMA *library* ***manip***.

Both PUMA and Clang provide a low-level code manipulation component that can be used by the AspectC++ weaver to do the actual source-to-source transformation.

PUMA contains a manipulation library that supplies basic insertion, deletion and replacement operations. Locations are given as tokens which limits its precision. If the user wants to manipulate a token she has to do so manually and replace the token through the manipulator. When the manipulation is done changes are directly committed to the project.

In contrast, Clang's `Rewriter` does not use tokens but uses `SourceLocations` which represent a byte offset into the file. This allows more flexibility than the design used by PUMA. When multiple insertions are made at the same point it can be specified if a new insertion should be added before or after existing changes. The `Rewriter` also takes care of mapping the locations to the source even if something was added in front of the location and essentially invalidating the byte offset. At the end all manipulations are rendered into strings, one for every manipulated file.

In the AspectC++ weaver the code manipulation primitives are encapsulated in the `WeaverBase` class. It currently uses a token-based model to address positions similar to PUMA. An interface based on `SourceLocation`s would be a strict superset of the token model.

The PUMA manipulator also provides a setting which automatically expands macros when a transformation affects an instance of it. Clang does not provide similar functionality.

3.9 Summary

This chapter showed that many of the PUMA components used by the AspectC++ weaver have close relatives in Clang, with the notable exception of the `Project` class which does not have a direct equivalent. Significant parts of the `Project` code can be reused with minor modifications though.

The other modules have varying degrees of compatibility, in particular the parts of the weaver that depend on the parser and AST will require complex changes, mirroring the large differences in the parser's data structures.

4 Design

This chapter describes the design of the components used to move AspectC++ to Clang. The design tries to facilitate agile software development by enabling an incremental way for the implementation. This allows testing during the entire process and prevents bugs from accumulating early. The other design goal is reducing changes the existing AspectC++ code to avoid breaking code by accident.

To keep changes to the AspectC++ code base to a minimum, only parts that are directly connected to Puma are rewritten for to use Clang. The goal is to have a working weaver at all times and being able to swap out Puma code for Clang code piece by piece. Some otherwise parser-independent parts of AspectC++ make direct use of Puma's classes, for those cases an abstraction layer is introduced. When both a Puma and a Clang version of a class exist a Clang prefix is added. This can be dropped when the Puma versions aren't needed anymore.

4.1 Code Manipulation

`WeaverBase` and its subclass `CodeWeaver` are the primary providers of source code manipulation in AspectC++. It is built on top of Puma's manipulator classes. `WeaverBase` wraps Puma's interface and extends it with a richer abstraction for addressing specific insertion positions in the source code. Clang provides a similar low-level interface, the Rewriter, which the new class `ClangWeaverBase` is built upon. It is based on `SourceLocation`s instead of tokens so the additional position handling of the old `WeaverBase` is not required anymore. If the Clang version is called with a token it will measure the length of the token and use the location of the start and the end of the token to emulate Puma's functionality. `CodeWeaver` provides higher level manipulation, such as specific transformations for the individual advices of the AspectC++ language and is most interesting for Phase 2. A major difference between the old implementation is that the Clang-based Phase 2 does not have access to any tokens so the token emulation cannot be used. It is necessary to rewrite the code using `SourceLocation`s only.

The Clang-specific code is implemented in the classes `ClangWeaverBase` and `Clang-CodeWeaver` so both implementations can be used at the same time. This also has the disadvantage of a lot of code duplication of the Puma-based `CodeWeaver` in `ClangCodeWeaver` that will go away once `CodeWeaver` falls out of use.

The Puma-based `CodeWeaver` makes use of Puma manipulator's facilities to selectively expand macros if a transformation occurs in a macro instantiation. Clang's rewriter does not provide this feature so it has to be reimplemented manually by checking for macros in every manipulation primitive and storing the location.

At the end of the weaving process it has to call Clang's preprocessor and expand the macro in place. This cannot be done on the fly because there may be other transformations in the same macro and once it is expanded and copied in the target file it cannot be changed anymore.

4.2 Changes to Phase 1

The following sections describe the individual changes to the classes used by phase 1 culminating in `Phase1` itself. Phase 1 and phase 2 share a lot of code, yet almost everything is reinitialized after phase 1 has run. Some duplication and unnecessary processing will be created during the porting process. This can be easily cleaned up when the phase 2 port is done.

ACToken

`ACToken` is a wrapper that can contain either a pointer to a PUMA token or a Clang Token. It defines a set of token kinds that are mapped to the values of the underlying implementation, to minimize changes to the AspectC++ code itself. The token kinds were extracted from AspectC++ with a simple `grep` operation and can be extended if more tokens are needed in the future. It defines a small set of operations that map well to either PUMA or Clang tokens. This is implemented as a non-Clang specific infrastructural refactoring in AspectC++, the primarily affected classes are:

- Phase1
- WeaverBase
- CodeWeaver
- IncludeExpander
- Introducer
- Transformer

First, the existing pointers to PUMA are stored in the `ACToken` instead of using raw pointers. When the token is used in conjunction with a `Unit` the `ACToken` is turned back into a PUMA Token until units are ported as well. Eventually conversions will only occur when tokens are imported from the preprocessor or sent into source code rewriting. The `ACToken` also has a companion class `ClangACToken` that wraps a Clang token instead of a PUMA one. There are slight semantic differences between Clang and PUMA tokens, for example PUMA uses a `NULL` pointer when the parser reaches the end of a file while Clang uses a special `EOF` token. Those special cases require manual changes to the AspectC++ code that uses the tokens as there is no reasonable abstraction for them.

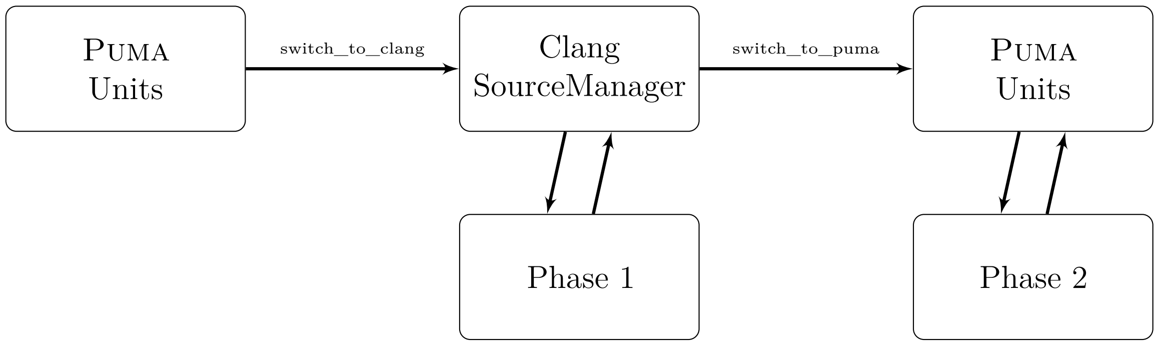

Figure 4.1: Clang phase 1 in a PUMA world.

ACFileID

The basic container for a file in PUMA is the `Unit` class. It consists of a file name, a list of tokens and information about the current state of the file. Clang only provides an ID that can be translated into a file name. The ID can be used in most places where a Unit is used in AspectC++. A wrapper class called `ACFileID` is created, which can contain a Clang `FileEntry` or a PUMA `Unit` but only allows taking the name of the underlying object. This is used to bridge filename-based operations between a Clang-based phase 1 and a PUMA-based phase 2. When an `ACFileID` from a PUMA `Unit` and a Clang one are compared, `ACFileID` does a textual comparison of the absolute paths of the underlying files. Otherwise it simply compares the pointers of the objects the `ACFileID` was initialized with. The textual file name comparison does not work for purely virtual buffers that have no counterpart on disk. Those never cross the boundary between the two phases so they can be ignored here.

Phase 1 contains a recording mechanism that uses `Units` as a temporary container for tokens. Clang does not provide a similar container but the usage in phase 1 is simple enough to be replaced with a `std::vector` of tokens. Phase 1 also makes use of so-called marker tokens to denote names in the recorded token list. Those are replaced with Clang tokens with an invalid ID, those should never occur in real code.

ACProject

`ACProject` is the top-level class containing parser state. In the porting process `ACProject` is extended to hold Clang state next to the existing state. As a temporary measure two new methods are introduced, `ACProject::switch_to_clang` and `ACProject::switch_to_puma`, which transfer state from PUMA to Clang and vice-versa. This is used to have a Clang-based phase 1 coexist in a PUMA-based `ac++`.

For the configuration it will use PUMA's configuration files that contain lists of command line arguments. Not all options are supported by Clang directly so light preprocessing and filtering of the list is required. This allows the Clang-based AspectC++ weaver to be used as a drop-in replacement for `ac++` in the `ag++` driver.

`ACProject` is derived from PUMA's `CProject` class, which also contains useful code to manage the file tree `ac++` sees, there is no direct equivalent for this functionality in Clang. The long-term plan for `ACProject` is to remove any PUMA state from it and lift PUMA code that's still needed after that from PUMA into ACProject.

ClangErrorStream

PUMA provides a way to emit diagnostic messages during the parsing process via the `ErrorStream` class, which follows the C++ iostream pattern. Clang has a similar way via its `DiagnosticBuilder` class. The new `ClangErrorStream` bridges PUMA's interface to Clang's to simplify porting of AspectC++'s code. Instead of PUMA's `Locations` it understands Clang's `SourceLocation` objects and passes them down into Clang's `SourceManager` formatting logic.

Basic Usage:

```
err() << sev_error << token->location()
      << "There's an error near this token."
      << endMessage;
```

Clang provides rich diagnostic options like highlighting of ranges in the source code and hints for code insertion to make it easier for the user to fix an error. Those features aren't currently exposed by `ClangErrorStream` but can be added in the future. One immediate improvement over PUMA's diagnostics is the addition of column numbers to the output.

ACPreprocessor

ACPreprocessor is a base class of `Phase1`. It is responsible for setting up the preprocessor and encapsulates the state of the preprocessor. This reduces dependencies on the actual implementation. The class is small and its methods can be mapped to Clang in a direct fashion.

IncludeGraph

The `IncludeGraph` class uses a callback mechanism to record `#include` directives in the source code as it is getting preprocessed. A graph structure is created that can be queried to reveal the include relation between to files. Clang provides a similar callback mechanism that is used for the Clang version of class and replaces the PUMA-based code in `IncludeGraph`. All uses of the graph in AspectC++ query the output of phase 1 so only one implementation is needed, with `ACFileIDs` bridging the gap between `Units` and Clang's `FileIDs`.

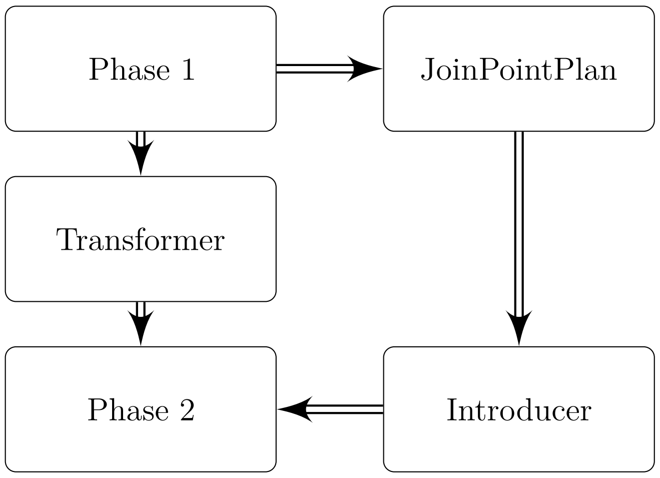

Figure 4.2: Flow of tokens in a PUMA-based phase 1.

Phase1

After all of the above is done, the actual phase 1 transformation pass can be ported to Clang. All uses of PUMA tokens are replaced with `ClangACToken`s. Some tokens are recognized by PUMA as a language extension, for example `advice` and `aspect` keywords. In Clang those are parsed as simple identifiers and their string value is compared instead of the type of the token. This avoids adding new kinds of tokens to Clang.

A complication arises from the recorded tokens that are used to communicate AspectC++ slice classes with phase 2 as seen in figure 4.2. PUMA tokens were recorded in `Phase1`, formatted in `JoinPointPlan` and inserted in the `Introducer`. Clang's tokens are managed by the `SourceManager` class and cannot be shared between the phases. Instead of passing tokens directly, the tokens are formatted into a textual string and passed down in this form. This eliminates the flow of tokens between the phases, leaving only the changes in the `Transformer` class.

Transformer

The `Transformer` is the top-level pass that manages the work of phase 1 and 2. It now calls the necessary operations on `ACProject` before and after the Clang-based phase 1 runs, transferring the contents from PUMA's `Units` to Clang's `SourceManager`. Over time the Clang part can cover larger stretches of the `Transformer` until everything is handled by Clang.

4.3 Changes to Phase 2

While Phase 1 required mostly refactoring and porting work, Phase 2 faces larger differences between PUMA and Clang, making structural changes necessary.

ModelBuilder

The `ModelBuilder` class analyzes semantic information coming from the parser and extracts interesting parts for consumption by the code transformations. The split nature of PUMA's semantic database and syntax trees is reflected in the structure of the `ModelBuilder`. First, it queries the semantic database for all classes and functions and inserts them into the model. Then a separate walk over the syntax tree is started to gather information on function calls. The syntax tree visitor is split out into the `TrackerDog` class.

In Clang there is only one data structure holding all the information of a translation unit, the AST. On the one hand this allows us to unify discovery of classes, functions and calls into one pass, obsoleting `TrackerDog`. On the other hand, `ModelBuilder` now has to deal with the intricacies of redeclarations in C++ and the increased verbosity of a syntax tree versus a simpler semantic database.

Most of the work on `ModelBuilder` itself is finding the right information in Clang's AST that corresponds to the information in PUMA's semantic database. Many checks can be directly replaced with a Clang method but some information (for example whether an element in the AST is derived from a template instance) is spread out over multiple nodes that have to be checked individually.

The generated model consists of two parts. One half is dedicated to parser-independent information such as class names and function call signatures. The other half contains pointers back to the parser data structures so the code transformations can query them later to get additional information and the original locations in the source where modifications should be performed. The parser-independent parts can be serialized into an XML format, leaving a way to compare PUMA- and Clang-derived models by getting two XML files for a given source file and comparing them.

ClangASTConsumer

`ClangASTConsumer` is a new class which implements recursive visitation of Clang's AST. It calls the corresponding methods of the `ModelBuilder` when it encounters an interesting node in the syntax tree. The class is built upon Clang's `ASTConsumer` interface, which provides a callback mechanism that gets activated every time a new top-level declaration is parsed. It then descends into the tree looking at every inner declaration and all function calls.

Because a call expression in Clang does not know its parent function the consumer has to keep this information around when it sees a function declaration. A complication arises in C++ global initializers which also can contain function calls, those have to be marked specially so they do not get attached to a different function by accident.

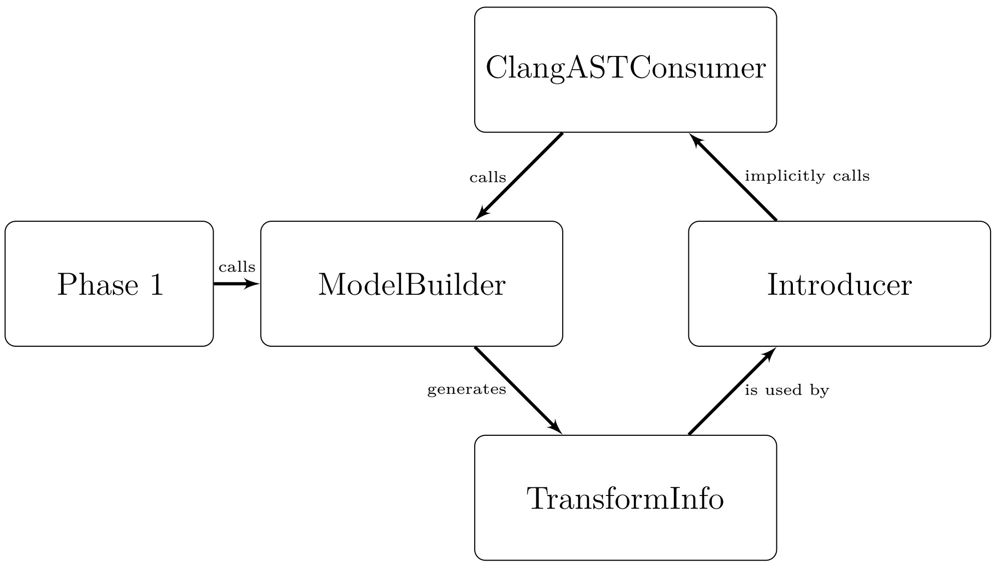

Figure 4.3: Interaction between model and introducer.

The `ClangASTConsumer` determines whether a declaration is interesting by querying the `ACProject`, checking if the file a declaration is in is contained in the project. Otherwise the declaration is ignored.

Introducer

The `Introducer` is the most complicated component of `ac++`, reading data from the model and parser and also injecting code and model information along the way.

One of the reasons why we need the introducer are advices that match inside of class slices. This can happen recursively so we need to parse the injected slices and add information about them to our model. A similar issue exists with inserted introspection code which may be called by a slice and code that is invalid without the added slices.

In the Puma version the `Introducer` is triggered by a callback from the Puma parser itself. The callback mechanism is introduced by an aspect during the compilation of Puma. The `Introducer` then prepares the injection and calls back into Puma to parse the code. The results are appended to Puma's data structures.

In Clang manipulating the AST is discouraged and Clang does not provide a convenient way to do it. For the Clang introducer a slightly different implementation was chosen. Again the parser is patched to provide a callback mechanism to trigger the introducer. But instead of parsing the injected code on the side and manipulating the syntax trees, the introducer for Clang injects the code into the running token stream of the parser. This way the new code gets included into the AST on the fly, and in turn triggers the callbacks in `ClangASTConsumer` to update the model. This design also comes with the downside of added fragility by moving data underneath Clang's parser. Extra care has to be taken so all parser state is set up in the right way when returning into Clang code.

This applies in particular to introduced aspect includes, which contain other code that the introduced code may rely on. Since we cannot paste them into the middle of a class the introducer has to reset Clang's context variables so they point to a top-level scope and call the parser methods manually. The actual `#include` directives are added with an extra code manipulation step.

All other injections are stored in `ClangIntroductionUnit`s which contain some meta data about the injection. Since we cannot manipulate code in text that's inserted with the `CodeWeaver`, any further code rewriting happens in the buffers referenced by the `ClangIntroductionUnit`s. The `ClangCodeWeaver` will insert them in the right order into the final target files when all other code transformations are done.

The Clang-based introducer has three entry points to deal with different kinds of AspectC++ slices.

- `base_specifier_first`: Is called when the parser sees a class with no base classes. The last parsed token is the opening brace. At this position the base class slice is added.

  Listing 4.1: Introduction of first base classes

  ```
  class C {
          ↑
          : public NewBase1, public NewBase2
  ```

  It is important that all aspect includes are resolved here as the base classes may be declared in such a header.

- `base_specifier_next`: Similar to `base_specifier_first` but gets called when the parser sees a class that already has one or more base classes. It also gets called when `base_specifier_first` introduced a base class; those cases are ignored.

  Listing 4.2: Introduction of second base class

  ```
  class C : Base {
                 ↑
                 , public NewBase1
  ```

- `tag_member`: The parser calls this callback when a class definition is parsed completely. The insertion position is before the closing brace. This is used to insert class members in slices and also `aspectof` methods, friend declarations and introspection code.

Listing 4.3: Introduction of inline member

```
class C {
   ⋮
};
↑
void injected_member();
```

For the additional `aspectof` functions, friend declaration and introspection code that are inserted at this stage. `aspectof` does not require parsing but friend declarations and introspection code have to be parsed since other introduced code can rely on it. Friend declarations and `aspectof` functions are added to all classes in the project scope while introspection code is only inserted when requested. Introspection code is also inserted after every member introduction.

- `end_translation_unit`: Finally, when the translation unit is parsed (the parser found an end of file token) this method is called. Special care has to be applied to the callback code because the Clang preprocessor deletes itself when it is done. This is solved by setting it into a special incremental parsing mode and resetting it when the introducer is done. In `end_translation_unit` non-inline member slices are added to the translation unit. To do this it iterates over all classes that we introduced members into earlier. If one or more of the introductions were non-inline and we are in a translation units where the class was defined or some other non-inline members were defined we inject the code here. The check for other non-inline members is important to avoid multiple definitions of a member function.

Listing 4.4: Introduction of non-inline member

```
   ⋮
<EOF>
↑
void C::injected_member() { }
```

On the Clang side there is a new abstract class `ACIntroducer` which is used by Clang's parser and gets implemented by the `Introducer` and Clang's semantic analysis pass stores a pointer to it. In Clang's parser, code calling the three callbacks is added in all places where the piece of code we are looking for is parsed. It also has to save the last lexed token and restore it after the introduction is completed.

The modifications are currently distributed as a patch set against a stable release of Clang. Implementing the changes in aspects that are applied to Clang was considered but using the current design it leads to a lot of code duplication which would not ease maintenance of the code. The code is also very specific to AspectC++ so submitting the patches to be included in Clang's upstream repository is unlikely to succeed.

ClangCodeWeaver

The `CodeWeaver` is the final big piece of phase 2. It does all the hard work of weaving the aspects into the output. It takes the information gathered in the model and applies the advices for every matching pointcut. Some transforms are only using parser-independent model information; those transforms can be directly taken from the PUMA-based `CodeWeaver`. Others query the syntax tree and have to be ported manually. In particular call advice manipulation requires a large amount of code and high precision at inserting the generated bits. It is possible that there are multiple insertions at the same location that have to be executed in the right order.

A shared component of all advice types is the insertion of `ThisJoinPoint` classes. To determine whether a matched joinpoint requires such a class the code weaver performs another walk over the syntax tree of the aspect code. The PUMA-based version does this by looking at `SimpleName` nodes and using the name to set various flags for the generator of those classes. In Clang this information is not available as a separate node and various ways to address the `ThisJoinPoint` object in C++ code must be checked instead.

For execution advice most of the code is parser-independent with the parser-dependent information wrapped in a `SyntacticContext` object. This is reduces the porting effort for this type of advice to providing a `SyntacticContext` object for Clang and minor changes to the actual weaving code. Since `SyntacticContext` is a relatively new development in the AspectC++ weaver, other advice types do not make use of it and the code handling them is deeply tied to PUMA data structures. It may be helpful to port them to `SyntacticContext` instead of directly porting them to Clang. This would require various extensions to `SyntacticContext`.

The last step done by the CodeWeaver before the transformations are committed back into the project is moving all the `ClangIntroductionUnits` into their targets. It is important that this is done in the right order so nothing is lost when introduction were recursive. Figure 4.4 shows a typical tree of `ClangIntroductionUnits` that occurs with recursive introductions. Each unit knows where it should be inserted in its parent unit. Some of the units also have code manipulations done inside them, which means that an advice matched here. The code weaver now does a postorder traversal of the `ClangIntroductionUnits` tree, starting at the leaves, inserting every unit into its parent. If there was code manipulation inside of the unit the manipulated code is inserted. The last step is copying the contents of the top-level introductions back into the translation unit.

IncludeExpander

The `IncludeExpander` is a late transformation pass independent of the `CodeWeaver`. It expands `#include` directives in case the underlying header file was modified. The PUMA version runs the preprocessor again and just replaces the `#include` directive with the tokens of the modified files. In Clang the text can be copied instead, using a lexer in raw mode so includes do not get expanded automatically by the lexer and using the same callback mechanism phase 1 uses.

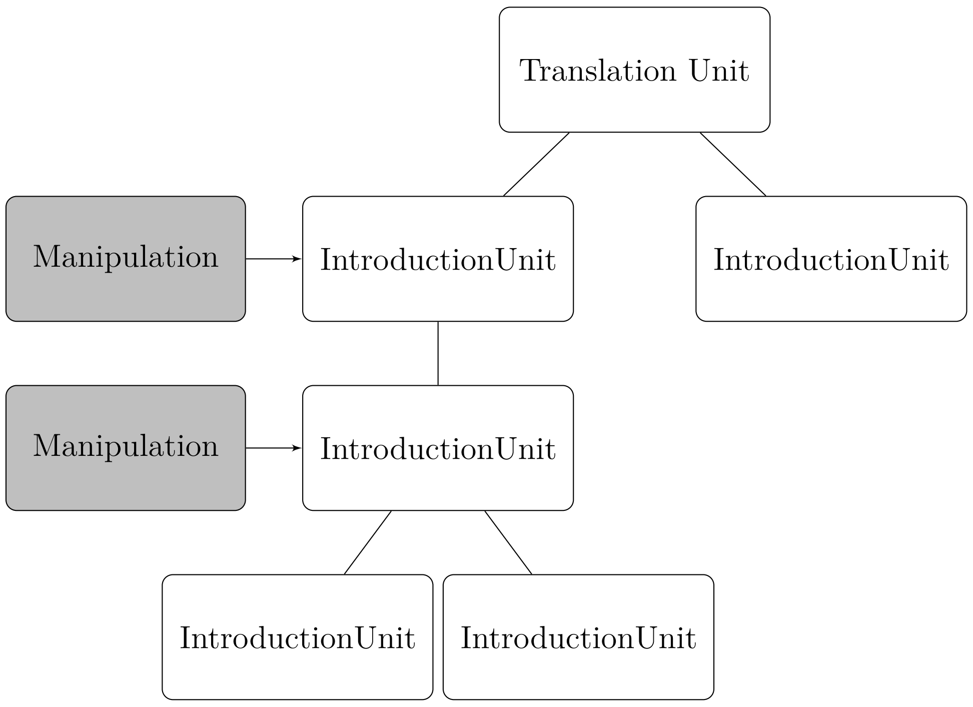

Figure 4.4: Tree of IntroductionUnits. Insertion starts at the leaf nodes of the tree.

4.4 Summary

A strategy for porting the major classes of the AspectC++ weaver was laid down in this chapter. Structural changes to phase 1 could be kept at a minimum which will hopefully lead to a quick and bug-free implementation. Phase 2 is more complicated, requiring deep structural changes to the `ac++` code that will certainly bring new problems.

5 Implementation

This chapter describes the steps that were taken to implement the elements laid down in the design chapter. It also describes the various issues that were encountered while writing the code and how they were solved.

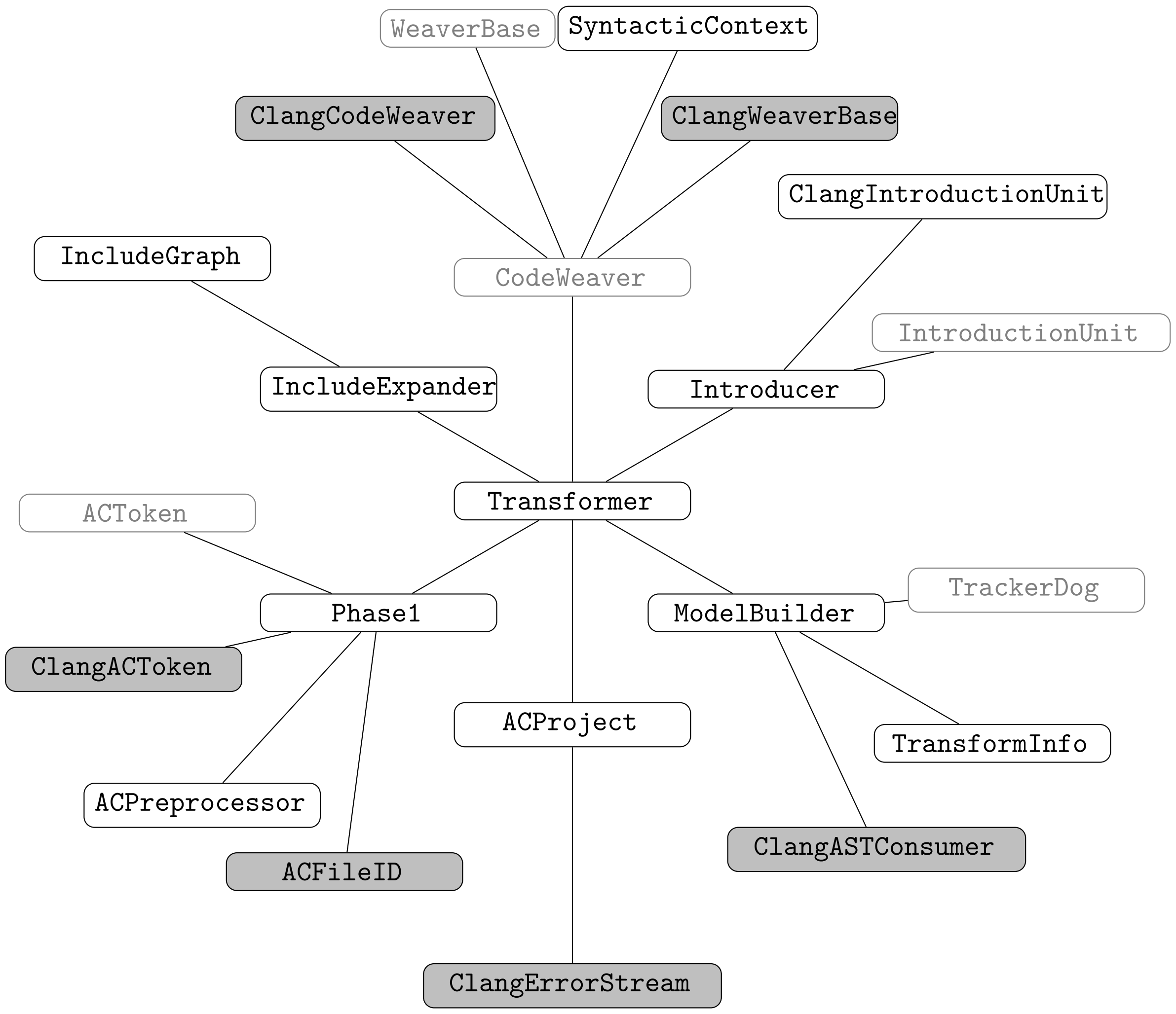

Figure 5.1: Modified classes in AspectC++. New classes are printed with dark background. Obsolete classes are printed in light gray. Related classes are grouped together and inner classes call classes at the edges of the diagram.

For the port seven new classes were introduced and many were significantly changed. Five classes became obsolete and can be removed when all code that uses them is removed. Over the span of four months more than 3000 lines of code were added, 1000 were deleted in AspectC++. The changes to Clang consist of 100 added lines.

Listing 5.1: Statistics for AspectC++ and Clang changes

```
AspectC++:
ACFileID.h              |  60 +++++++
ACPreprocessor.cc       | 128 ++++++++-------
ACPreprocessor.h        |  48 +++---
ACProject.cc            | 138 +++++++++++++++-
ACProject.h             |  18 +++
ACToken.h               |  52 ++++++
ClangACToken.h          | 147 +++++++++++++++++
ClangASTConsumer.cc     | 117 ++++++++++++++
ClangASTConsumer.h      |  51 ++++++
ClangErrorStream.h      |  89 +++++++++++
ClangWeaverBase.cc      | 138 ++++++++++++++++
ClangWeaverBase.h       |  85 ++++++++++
IncludeExpander.cc      |  44 ++---
IncludeExpander.h       |   7 +-
IncludeGraph.cc         |  82 ++++------
IncludeGraph.h          |  63 ++++----
Introducer.cc           | 687 ++++++++++++++++++++++++++++++
Introducer.h            |  58 +++++--
IntroductionUnit.h      |  61 +++++++
ModelBuilder.cc         | 465 ++++++++++++++++++++++++++---
ModelBuilder.h          |  81 ++++++++--
Phase1.cc               | 725 ++++++++++++++++++++++++++++
Phase1.h                |  29 ++--
SyntacticContext.h      | 212 +++++++++++++++++++++++
TransformInfo.h         | 380 +++++++++++++++++++---------
Transformer.cc          | 251 +++++++++++++++++-----------
Transformer.h           |  17 +-
27 files changed, 3294 insertions(+), 939 deletions(-)

Clang:
include/clang/Parse/Parser.h |  8 ++++++
include/clang/Sema/Sema.h    | 37 +++++++++++++++++++++++++
lib/Parse/ParseDeclCXX.cpp   | 57 +++++++++++++++++++++++++
lib/Parse/Parser.cpp         |  9 ++++++
lib/Sema/Sema.cpp            |  3 +-
5 files changed, 110 insertions(+), 4 deletions(-)
```

5.1 Build System

AspectC++ uses a Makefile-based build system. All components are placed in a single directory and are built individually. Dependencies are expressed using relative paths. This can be adapted easily by adding an LLVM and Clang source tree next to the existing directories. Using an external version of Clang is not possible due to the AspectClang-specific modifications to the Clang source code for phase 2. Furthermore, on Unix-based operating systems LLVM has to be built with support for exceptions which are disabled by default by LLVM's build system. This is necessary because `ac++` makes use of exceptions and code built with and without exception support cannot be used in the same binary in general. When a solution for this is found it may be possible to optionally use a Clang library provided by the system (for example a package in a Linux distribution) to reduce the build times and foot print of `ac++`.

LLVM supports two different build systems: one based on the GNU autoconf system and Makefiles, and one using CMake. For sake of simplicity AspectC++ uses the Makefile-based system only. The build settings were chosen to disable as many unneeded components as possible to reduce the time required to compile LLVM and Clang. The specific build parameters are described in the `README-LLVM` file at the top level of the AspectClang source tree. The modifications to Clang are distributed as a patch file that has to be applied before Clang is built.

Clang updates are released every six months [18]. It is planned to update AspectClang every time a new release becomes available. Another possibility was tracking Clang development by taking the code directly from SVN, this option was discarded to avoid changes to Clang that would break AspectClang underneath us. No critical bugs in Clang were found that would necessitate using a newer version.

5.2 Phase 1

After an extensive planning phase, porting of the phase 1 transformations could commence. Surprisingly the most error-prone part of this process was getting a clean separation between PUMA-based and Clang-based parts. This includes turning token-based communication into a string-based approach in the `JoinPointPlanner` but also getting the switching in `ACProject` just right. In particular making sure that no state from the early PUMA-based parts is carried over to phase 2. This lead to dangling pointers to tokens that do not exist anymore.

It is also important to set up everything right when switching back from Clang to PUMA. The tokens must reference the original source files and not any temporary buffers created by Clang. They also must be tokenized with the same parameters used in the old phase 1 or phase 2 will see unexpected token kinds. This was more complicated than expected and took multiple attempts to find a solution that works for all inputs.

The resulting AspectC++ weaver with both Clang- and PUMA-based components was verified using AspectC++'s own regression tests and by building a working PUMA, which itself is a heavy user of AspectC++. All tests except one pass, the single failure is due to not properly updating line numbers in the model, this will be fixed as part of phase 2 porting. Doing a self-hosted compilation with an AspectClang phase 1 compiled PUMA linked against itself also passes the same regression tests.

5.3 Phase 2

As expected, the implementation for phase 2 was a lot more complicated and did not go through as smooth as phase 1. The first major piece was the port of the `ModelBuilder` class, which included setting up a Clang parser and retrofitting the builder to the more incremental way of discovering declarations through the `ClangASTConsumer`.

ModelBuilder

To implement a Clang-based `ModelBuilder` in an incremental way, all of the existing `TransformInfo` data structures storing the parser-specific parts of the model were equipped with the necessary pointers into Clang's AST, right next to the existing PUMA syntax tree pointers. Then the functionality of `ModelBuilder` was replaced method by method. While doing this many points were found where Clang deviates from PUMA's way of describing a C++ source file.

- Through the incremental nature of the `ASTConsumer` it is not possible to know whether a declaration has a definition at the time we see the declaration. This means we have to keep a list of all redeclarations and update it when Clang finds a new one.
- Clang does not have a simple way of telling whether a node in the AST was derived from a template. We have to walk up the surrounding scopes until we find one with the template instantiation flag set.
- Even if a class does not have any user-provided destructors or constructors we still have to add them to the model. Clang stores information about them in the AST which has to be processed separately.
- Names can be spelt differently. For example PUMA calls an anonymous namespace `<unnamed>` while it is named `<anonymous namespace>` in Clang. The `ModelBuilder` has to translate those notions where necessary otherwise a pointcut referencing a PUMA-style name won't match later on.

To verify that the model is emitted correctly the XML serialization capability of `ac++` was used. For various test cases a file was generated with both a Clang-based and a PUMA-based `ac++` and manually compared. This allowed finding bugs in the parser-independent model quickly, some minor differences in line numbers remain that may need investigation later. It was not possible to test the parser-dependent part of the model this way, which meant potential mistakes remain until the consumers for the data are implemented.

Introducer

The `Introducer` is the most complex part of the AspectC++ weaver and thus the most error-prone. The implementation started by patching Clang to provide the needed callbacks and save required state when calling back into `ac++`. Adding new code in the middle of a parsing process when the parser does not expect it leads to subtle and hard to debug issues that remain unnoticed until a very complex source file is processed by the weaver.

- Code insertion as part of weaving is now completely decoupled from injecting code into the parser. This leads to situations where code is inserted in a different position than the parser sees it which can lead to strange error messages on apparently intact code.
- When code is injected it can make the parser call the introducer's methods again with the newly parsed code. Special care has to be taken to not do an introduction twice in those cases.
- Injecting code when the parser does not expect it can confuse the parser and cause it to drop tokens or crash. The Clang side has to be carefully written to avoid going back to the wrong state after an insertion is parsed.
- Parsing introduced includes works very differently from the slice introductions. Here the parser is reset to a global state and called manually on the included source file. This means it also has to manually update the model by calling `ClangASTConsumer`'s methods on the newly parsed AST nodes.
- If there are multiple introductions at the same position they have to be added in the reverse order of appearance in the output, the last injected code is parsed first.
- Added includes can contain classes which again need introductions leading to recursion in the introducer itself. It is important that the `Introducer` does not store any state in the class that would be overwritten by a deeper recursion step.

The new introducer was tested by running AspectC++'s regression tests and then on PUMA, without advice weaving disabled so only slices were added. This found a plethora of subtle implementation mistakes and crashes in the `Introducer`. Given the complexity of this component it will likely stay a major source of problems in AspectClang for some time.

ClangCodeWeaver

The `CodeWeaver` is the largest class of `ac++` and a lot of it is specific to PUMA and requires manual porting. Work was started by copying the code needed to transform execution advice from the old `CodeWeaver` and porting it piece by piece.

- The code emitting wrappers for function definitions could be mostly reused directly by just adding Clang support to the `SyntacticContext`. In the context the only roadblock was finding the right locations in all the Clang AST nodes. The nodes carry many different locations and it is often not obvious which location addresses which part of a declaration.
- Another complication arose from the way Clang prints types. If the type is turned into a string with the obvious `getAsString` method it will assume that the type is a C type and will emit boolean types as `_Bool` instead of `bool`. That is not valid C++ code. To fix this issue, a `PrintingPolicy` has to be passed to `getAsString` everywhere.
- Walking the AST to figure out whether a `ThisJoinPoint` class has to be generated turned out to be difficult to convert to Clang. To gather AST snippets that trigger `ThisJoinPoint` the regression tests were run and failing tests examined for missing `ThisJoinPoint` classes. Code to match the found snippets was added but it is likely that more cases have to be added to reach feature parity with the old weaver.
- Insertion of `proceed` functions which are used to call the original function from advice code was not converted to `SyntacticContext` yet. Since most of it deals with call advice only the execution advice specific parts were directly ported.

With those changes most execution advice regression tests pass with AspectClang. Construction, destruction and call advice transformations were not implemented due to time constraints. Doing that should be relatively straightforward with all the other components being ready.

Another missing feature is the expansion of macros when a transformation occurs inside a macro instantiation. This is not a widely used feature so it was dropped for now.

IncludeExpander

Since the `IncludeExpander` works independently of the rest of the weaver and the `CodeWeaver` was not finished in time the expander was not ported. This leaves a dependency on the PUMA preprocessor in place that still has to be replaced. This has an impact on the overall performance of AspectClang.

5.4 Summary

The implementation of phase 1 was completed without major interruptions and is well tested and stable. Phase 2 is not yet finished. Of the components of phase 2 the model and `Introducer` are almost complete and tested.

Table 5.1: Implementation status of various AspectC++ components

Component	Progress Estimate
Phase 1	100%
Phase 2	
ModelBuilder	95%
Introducer	85%
CodeWeaver	50%
IncludeExpander	0%

- The `ModelBuilder` is feature complete but suffers from occasional bugs due to the structural differences between PUMA and Clang.
- The `Introducer` is missing introspection code but otherwise complete. It is, however, a fragile piece of code where problems are abundant.
- The `CodeWeaver` only has basic support for execution advice, other advice types are currently missing. This includes call advice which are the most complicated advice type in terms of code transformation.
- The `IncludeExpander` was not ported to Clang and still uses PUMA's preprocessor which allows it to work but leaves a dependency on PUMA.

6 Evaluation

Most of the AspectC++ weaver is now implemented using Clang. This chapter focuses on how the new implementation compares to the predecessor based on Puma. The major aspects that are evaluated are conformance to the C++ standard and overall performance. C++ conformance was one of the foremost issues with Puma, with many open bug reports about it poorly handling certain C++ constructs. Clang claims to implement the C++03 and C++98 standards completely [16] so it should perform better. Weaving speed is critical for development and debugging of software using AspectC++, it would be great if Clang offers an improvement here as it is heavily optimized for fast compile times [1].

6.1 C++ Conformance

In contrast to Puma, which has little use outside of AspectC++, Clang is widely used as a C++ compiler which gives it a high coverage of uncommon C++ features. To get a first estimation on the C++ conformance all open bugs in the *parser* category in AspectC++'s bug database [1] were tested. Table 6.1 lists them. The *Puma* and *Clang* columns show whether the provided test case parses without error using a Puma- or Clang-based AspectC++ weaver. The *Type* column categorizes the bug reports into one of the following groups.

C++ conformance This is the most common category which contains bug reports where Puma rejects valid C++ input. Clang excels at this, resolving all known bugs.

GNU extension Lists bugs concerning GCC extensions not fully supported by Puma. Clang supports a larger number of those extensions than Puma does, the only one that is missing in the list of bug reports are the <? and >? operators, which have been deprecated by GCC and support was dropped in GCC 4.0. It is unlikely that they will ever be implemented in Clang.

VC extension Lists extensions from Microsoft's Visual C++ compiler. Both Clang and Puma support a limited subset of those extensions which are important for correctly parsing header files on Microsoft Windows. It is not clear whether some of the extensions are actually bugs in Microsoft's compiler that have been fixed since the bug report was filed.

[1]http://www.aspectc.org/bugzilla/

Internal Puma bug The test cases concern implementation details in Puma which do not directly apply to Clang.

Invalid or incomplete test case The given test case was not sufficient to generate a reproduction of the bug reported or contained invalid C++.

C only Bug only affects C code and is irrelevant for AspectC++.

Not a parser bug Bugs concerning parts of Puma or AspectC++ unrelated to C++ parsing.

Of the tested bug reports, AspectClang could successfully compile 36, compared to the old AspectC++ weaver failing all but seven. The seven remaining bugs were probably fixed but the report was not closed. This means more than 29 bug reports can be closed immediately when Clang becomes the primary C++ front end for AspectC++, most of which preventing using AspectC++ with popular C++ code bases. This does not take into account new bugs that were introduced by the switch to Clang but it is likely that the amount of inappropriately rejected inputs will be much lower with the new implementation.

6.1.1 Compatibility with Other Code Bases: SystemC

SystemC is a C++ class library to simulate and synthesize electronic systems. Hardware description and aspect oriented programming play well together and so do SystemC and AspectC++ [19] making it a natural target for evaluation.

A first attempt at trying to compile SystemC 2.2.0 showed one of the differences that parsing with Clang brings. It is much stricter and will reject more invalid code than Puma did. In this case SystemC contains multiple constructs that are invalid according to the C++ standard, for example declaring an instance reference variable as mutable. Clang rejects this code requiring changes while Puma has no problem with it. Clang is also stricter than many other compilers. The widely used GCC C++ compiler accepted this particular construct before version 4.6 which was released in 2011 [20]; SystemC 2.2.0 dates from 2007. It is likely that other old C++ code will need patches to properly compile with Clang too.

Luckily SystemC 2.3.0 fixed the build with Clang so existing code bases may compile after upgrading them to version 2.3.0. There is a minor bug in the `ag++` driver which prevents using the `-pthreads` command line flag with it but that can be worked around by removing it from the compiler flags line.

All SystemC 2.3.0 tests compile cleanly and pass when built with AspectClang. They don't make any use of AspectC++ features though.

Table 6.1: Status of AspectC++ bugs with PUMA and Clang as of August 15, 2013

ID	Summary	Type	Puma	Clang
40	advice aspects cannot be used with methods/functions with...	not a parser bug		
61	wrong handling of protection and casting	C++ conformance	no	yes
73	Function bodies of templates are completely skipped.	not a parser bug		
100	VisualC++: Explicit constructor call in a member access e...	VC extension	no	no
102	Visual C++: Implicit int support	VC extension	no	no
145	argument default values for pointers and references	C++ conformance	no	yes
186	Scanner problem with old DOS files	VC extension	no	no
187	__stdcall *	VC extension	no	yes
188	GNU <? and >? operators	old GNU extension	no	no
189	friend declarations	VC extension/C++11	no	yes
220	Recursive Inheritance	C++ conformance	no	yes
236	Parser fails with non trivial template metaprogram	invalid test case		
258	destructor of POD types	C++ conformance	yes	yes
264	Do not handle "taking address of temporary" as error	invalid test case		
275	resolution of cast expressions	internal PUMA bug		
276	implicit conversion in init declarators	internal PUMA bug		
277	Puma: original parsing example missing	not a parser bug		
287	__wchar_t is no keyword	VC extension	no	yes
289	Overload resolution not g++ compatible	C++ conformance	no	yes
298	Visual C++ __pragma keyword "disable" causes parser error	VC extension	no	yes
299	Parser error with Microsoft Visual C++ 8.0 keywords	incomplete test case		
302	Problems processing examples with AspectC++ 1.0pre3 and c...	incomplete test case		
305	non-local template members not instantiated	C++ conformance	no	yes
309	Error when try to compile the examples	not a parser bug		
325	No error message for invalid class member initializer	VC extension	no	no
338	problem with two-dimensional static member array	C++ conformance	no	yes
347	Early name binding in templates not implemented	C++ conformance	no	yes
348	explicit specialization of function template not linked w...	internal PUMA bug		
356	Backslash path delimiter in #include	VC extension	no	yes
365	Inconsistent syntax of slice introduction advices	not a parser bug		
369	the parser cannot handle the <boost/thread.hpp> header	incomplete test case		
374	GNU __attribute is not supported	GNU extension	yes	yes
375	GNU __attribute__ ((__mode__ (__<SIZE>__))) not...	GNU extension	no	yes
388	c99 type of aggregate initialization not supported	GNU extension	no	yes
406	Implicit call to overloaded operator of template class no...	C++ conformance	yes	yes
424	Overload resolution for pointers to static member functio...	C++ conformance	yes	yes
428	preprocessor does not support C++ and/or/not etc. tokens	C++ conformance	no	yes
433	Weave failing on operator overload	C++ conformance	yes	yes
435	Multiple definitions error with templates.	incomplete test case		
437	Function address as template argument: & needed	C++ conformance	yes	yes
438	GNU extension ?: not supported	GNU extension	no	yes
453	reserved identifier violation	not a parser bug		
454	Pointer attributes are not initialised in constructor ini...	not a parser bug		
455	Copy assignment operators without check against self-assi...	not a parser bug		
466	Template specialization not found if defined before prima...	C++ conformance	yes	yes
468	preprocessor turns #define into whitespace and not into a...	Preprocessor	no	yes
473	Deduction of template parameters types between function a...	C++ conformance	no	yes
479	Forward-declared template functions are not instantiated	internal PUMA bug		
480	template specialization by sizeof() differs to g++ due to...	GNU extension	no	yes
482	invalid operand to array subscript '[]'	C++ conformance	no	yes
487	__builtin_types_compatible_p not correctly implemented	C only		
488	semantic analysis of typeof() arg not correct in the C pa...	C only		
492	GNU built-in type traits incomplete	GNU extension	no	yes
496	dimension of static array in struct not stored correctly	internal PUMA bug		
502	Complicated overload resolution scenario	C++ conformance	no	yes
505	Template function call with explicit template args not co...	C++ conformance	no	yes
506	Conversion operator in template class seems to be ignored	C++ conformance	no	yes
507	Cyclic dependency between tempalte default arg and templa...	C++ conformance	no	yes
508	Overload resolution problem with argument that is a bitfi...	C++ conformance	no	yes
510	Arithmetic '<' confuses template-id parsing	C++ conformance	no	yes
511	Dependent type not correctly detected in template definition	C++ conformance	no	yes
514	make examples fails	not a parser bug		
515	template template confusion in specialization	C++ conformance	no	yes
			Sum 7	Sum 36

6.1.2 Compatibility with Other Code Bases: Qt

Qt is a large C++ library, containing, among other things, a tool kit for graphical user interfaces. It also has many users in the AspectC++ community making it an important test case for AspectClang. To evaluate the changes in C++ conformance, a recent Qt version (4.8.5) was downloaded and built. After that, the included example projects were built with both a PUMA-based and a Clang-based AspectC++ weaver. This exercises essentially all headers in Qt which would also be used with any other application using Qt.

AspectC++ has clearly been thoroughly tested against Qt 4.8 and all tests passed with a PUMA-based weaver. The tests actually found a new problem with the Clang-based. The Clang-based phase 1 is stricter and does allow AspectC++ keywords in fewer places. A PUMA-based phase 1 did not show this behavior. Rejected cases include `slice` used as an argument name. The same problem exists for other AspectC++ keywords.

```
static void append_slice(PieSlice *slice);
```

AspectClang refuses to compile this example while the old `ac++` accepts it. Three Qt examples were affected by this.

- chapter5-listproperties
- chapter6-plugins
- pbuffers

For the newer Qt 5.1 things look quite different. Version 5 has started making use of new C++11 features when it identifies a compiler that's sufficiently new. Since PUMA doesn't qualify for this Qt tries a fall back mechanism, but this also fails. Without patching PUMA Qt 5 is completely unusable with AspectC++. With Clang things look much better, the only failures are due to the phase 1 problem listed above, making use of AspectC++ keywords in unexpected places. Two tests are affected by this.

- pbuffers
- boxes

Qt 5 (and to some degree Qt 4.8) also provide new features using C++11 [21]. It's still possible to build it without using a compiler with any support for C++11 but it is likely that users will pick up the new features eventually. AspectC++ is not yet ready for the new C++ revision but having it built upon a parser with good support for the new standard will make the transition easier.

6.1.3 Compatibility with Other Code Bases: LyX

LyX is a visual LaTeX editor. It is interesting because it makes use of both Qt 4 and boost, another widely used C++ class library known to be exceptionally challenging for C++ compilers. Puma cannot even preprocess all boost headers properly and is extremely far from parsing it. Clang claims to compile boost in its entirety [13].

LyX only uses a subset of boost but a Puma based weaver fails to compile any file from the project. With Clang underneath it gets further along but eventually stops. There were multiple issues identified in the **LyX** code base.

- Since Puma cannot preprocess boost and AspectClang still runs the preprocessor once to expand includes late in the pipeline. Since the expander is not needed for plain C++ code without any aspects it can be disabled to avoid this issue during the build.

- Phase 1 fails to parse an explicit instantiation of a boost template in the **LyX** code base.

  ```
  #include <boost/crc.hpp>
  template struct
    boost::detail::crc_table_t<32, 0x04C11DB7, true>;
  ```

  This bug is also present in a Puma `ac++`. A comment above the instantiation describes it as working around a bug in older compilers. It can be removed without impacting the overall build.

- It found multiple bugs in the `ModelBuilder` part for Clang, sending it into an infinite loop while trying to analyze boost headers. Those issues have been fixed in AspectClang.

With the minor tweaks listed above, AspectClang can build a working **LyX**.

Boost headers have been by far the most challenging pieces of code that were tested with `ac++`. It has uncovered problems in many parts of the AspectC++ weaver. The issues were not fundamental and could be easily fixed or worked around, opening up boost to the AspectC++ community. This is something that was never possible with Puma and will certainly be useful for many users of AspectC++.

6.2 Performance

Performance was evaluated after phase 1 was implemented and a more thorough analysis was done after phase 2. This shows the impact of the individual phases on overall weave times. All measurements were performed on a `x86_64` system with *Debian 7.0*. If not stated otherwise all given times show wall times and represent the average of three consecutive runs. Since the AspectC++ weaver and in particular AspectClang is still in development revision numbers for the AspectClang branch are added for better reproducibility.

6.2.1 Phase 1

With just phase 1 changed[2] to use Clang as a preprocessor a first performance analysis was performed. In figure 6.2.1 the timings for weaving individual source files in PUMA are shown, which itself is a heavy user of AspectC++. Weaving PUMA triggers many conditions not covered by AspectC++'s regression test suite.

Building PUMA with Clang phase 1 and PUMA phase 2 takes about 1.22× the time than with a pure PUMA `ac++`[3] on weaving PUMA. The most plausible explanation for this is the retokenization overhead when switching between the different parsers. The current implementation runs multiple lexical analysis steps on the input code, some of which are unnecessary.

1. At the beginning of the weave processes the main file is read, tokenized and a preamble is inserted for use by both phases. This could be done on a textual basis without turning it into tokens. When entering phase 1 the tokens are discarded and only the text is used by Clang.

2. Now Clang tokenizes and preprocesses the code for phase 1. The tokens are analyzed and transformations are applied.

3. After phase 1 the modified text is turned into tokens once again for consumption by phase 2. In the old phase 2 all manipulation was based on tokens so this wasn't necessary. When phase 2 is ported it the Clang preprocessor will take over this tokenization step.

Given the speed of Clang's preprocessor over PUMA's it is likely that when everything is switched over to Clang this performance regression will be reverted. There is a lot of optimization potential and only one tokenization step per phase is needed instead of two. It may also be possible to cache preprocessor state between the phases to gain a speedup in this area [22].

[2]SVN revision 258
[3]SVN revision 217

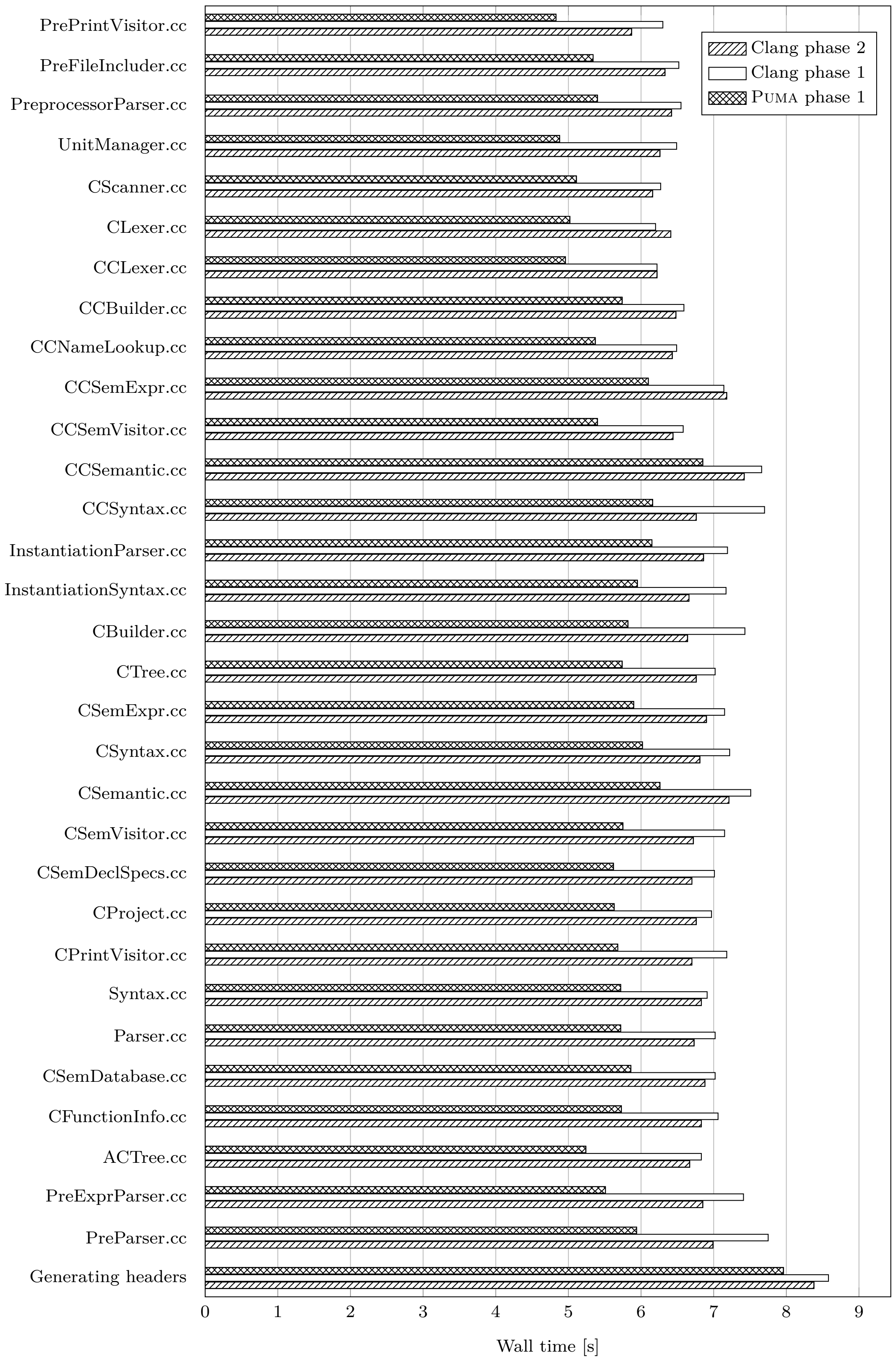

Figure 6.1: Weaver timings on PUMA with the original `ac++`, Clang phase 1 and Clang phase 2

Filename	Wall Time P1P2	σ P1P2	Wall Time C1P2	σ C1P2	Wall Time C2P2	σ C1C2
PrePrintVisitor.cc	4.83s	0.23	6.30s	0.03	5.87s	0.02
PreFileIncluder.cc	5.34s	0.08	6.52s	0.02	6.33s	0.06
PreprocessorParser.cc	5.40s	0.21	6.55s	0.03	6.42s	0.02
UnitManager.cc	4.88s	0.10	6.49s	0.04	6.26s	0.04
CScanner.cc	5.11s	0.12	6.27s	0.04	6.16s	0.01
CLexer.cc	5.02s	0.07	6.20s	0.02	6.41s	0.39
CCLexer.cc	4.96s	0.04	6.22s	0.01	6.22s	0.06
CCBuilder.cc	5.74s	0.03	6.59s	0.02	6.48s	0.02
CCNameLookup.cc	5.37s	0.10	6.49s	0.02	6.43s	0.05
CCSemExpr.cc	6.10s	0.09	7.14s	0.02	7.18s	0.41
CCSemVisitor.cc	5.40s	0.07	6.58s	0.02	6.44s	0.03
CCSemantic.cc	6.85s	0.01	7.66s	0.03	7.42s	0.04
CCSyntax.cc	6.16s	0.03	7.70s	0.85	6.76s	0.04
InstantiationParser.cc	6.15s	0.04	7.19s	0.29	6.86s	0.02
InstantiationSyntax.cc	5.95s	0.05	7.17s	0.56	6.66s	0.00
CBuilder.cc	5.82s	0.08	7.43s	1.31	6.64s	0.02
CTree.cc	5.74s	0.04	7.02s	0.34	6.76s	0.04
CSemExpr.cc	5.90s	0.04	7.15s	0.19	6.90s	0.04
CSyntax.cc	6.02s	0.06	7.22s	0.44	6.81s	0.03
CSemantic.cc	6.26s	0.03	7.51s	0.48	7.21s	0.01
CSemVisitor.cc	5.75s	0.16	7.15s	0.43	6.72s	0.02
CSemDeclSpecs.cc	5.62s	0.04	7.01s	0.41	6.70s	0.02
CProject.cc	5.63s	0.13	6.97s	0.31	6.76s	0.02
CPrintVisitor.cc	5.68s	0.05	7.18s	0.47	6.70s	0.00
Syntax.cc	5.72s	0.07	6.91s	0.31	6.83s	0.23
Parser.cc	5.72s	0.02	7.02s	0.34	6.73s	0.03
CSemDatabase.cc	5.86s	0.08	7.02s	0.24	6.88s	0.01
CFunctionInfo.cc	5.73s	0.04	7.06s	0.26	6.83s	0.03
ACTree.cc	5.24s	0.03	6.83s	0.19	6.67s	0.05
PreExprParser.cc	5.51s	0.02	7.41s	0.68	6.85s	0.02
PreParser.cc	5.94s	0.04	7.75s	1.05	6.99s	0.02
Generating headers	7.96s	0.23	8.58s	0.20	8.38s	0.07

Table 6.2: Weaver timings. PUMA phase 1&2 (P1P2), Clang phase 1 PUMA phase 2 (C1P2), Clang phase 1&2

6.2.2 Phase 2

Even though phase 2 is not yet complete[4] it can weave all of PUMA. Call advices are missing from the output so the result is non-functional. Profiling shows that a majority of the weaving time is spent preprocessing and parsing C++ code, so the actual transformation time is negligible.

As the data in table 6.2 shows, `ac++` with both phases converted to Clang takes about 1.17× as much time as the original AspectC++ weaver. This is a small improvement over the 1.22× of the hybrid Clang phase 1 PUMA phase 2 version. The tokenization steps are now as following.

1. The first tokenization to insert a preamble is still in place.
2. Now Clang tokenizes and preprocesses the code.
3. For phase 2 Clang tokenizes the code a second time.
4. For writing the transformed source code to disk PUMA's `Project` is used. It uses tokenized data so the code must be sent through the scanner one more time.

The first and the last step is unnecessary and could be removed as an optimization. Clang also performs more thorough semantic analysis checking for errors and possible mistakes than PUMA does. This could contribute to another minor slowdown. During the weaving of PUMA many warnings are shown for dubious constructs in PUMA's code base.

Profiling shows that a lot of time is still spent in PUMA. There are some obvious hotspots.

- Clang parsing for phase 2. This takes about 30% of the overall runtime.
- `IncludeExpander`, which runs the preprocessor again. This takes about 15% of the overall runtime.
- Switching to PUMA, this reads all input files and turns them into tokens. This takes about 15% of the overall runtime.

[4]SVN revision 340

6.3 Discussion

The evaluation showed examples where AspectClang shines but also some if its dark corners. Some questions were left open that are discussed in this section.

What was not tested?

Emitted code for aspects
: The Clang-based AspectC++ weaver is not yet in a state where it can correctly weave complex examples making use of different types of advice. Basic examples using only execution advice work but not much else.

C++11 support
: While it is not unlikely that many C++11 constructs will work out of the box with a Clang-based weaver the interactions with advice code are complicated and beyond the scope of this thesis.

How well does AspectClang support C++?

The test results were very promising. All of the reported PUMA bugs regarding C++ conformance are no longer observable with a Clang-based weaver. The new weaver also works well with code bases that were supported by PUMA and significantly expands upon its capabilities. New code bases like Qt 5 are supported without any further work needed and even intricate code bases like boost can be supported with only minor tweaks to the weaver.

This is a significant improvement over PUMA which aging code base has often problems with newer code like Qt 5 or even Clang's source code.

What impact has Clang on the time spent weaving AspectC++?

With the testing performed on the PUMA code base AspectClang was slower on average, taking about 1.17× the time the old weaver did. There is still potential for optimization though so it might be possible to remove this performance regression or even get an improvement over PUMA. Clang's parser has proved to be very fast but only accounts for about 60% of the time spent in the weaver. The rest is still taken up by the remaining PUMA parts and the weaver itself, evaluating pointcuts and doing code transformation.

7 Conclusions and Future Work

In this thesis the structure of the AspectC++ weaver was thoroughly analysed and a design was proposed on how to move the existing Puma-based weaver to the Clang C++ front end. Many similarities between the two C++ parser implementations were found but also significant differences that made extensive changes to the weaver itself necessary. All design challenges could be overcome, bringing a complete picture of what a Clang-based AspectC++ weaver could look like.

The proposed design was later implemented, which resulted in the AspectClang weaver. The existing AspectC++ was adapted in an incremental fashion, so a mostly working weaver was available at all times and Puma- and Clang-based components could coexist in the same binary. This allowed testing parts of the weaver before everything is completed.

Not everything in the weaver was ported to Clang in the scope of this thesis, but all major components are working and went through significant testing.

- The aspect discovery phase is complete. It successfully parses all AspectC++ code that was tried with it and detects aspects using Clang's preprocessor. The output with aspect code removed can be fed into a Puma-based transformation phase and passes all AspectC++ regression tests and can weave a working Puma, which itself is a heavy user of AspectC++.

- The model building component that translates syntax trees into a simplified model to be consumed be the code transformations had to be significantly changed to fit into Clang's model of processing a syntax tree. Nevertheless it works flawlessly for many inputs producing a model that is very close to the one emitted by the old weaver.

- The slice introduction mechanism also had to undergo exhaustive reworking to inject code into Clang's parser while it is processing source code. This turned out to be the most error-prone piece of AspectC++ and a cornucopia of subtle bugs. This was tested by running it on the Puma code base and, after many hours of debugging, it seems to work correctly on this use case.

- The actual code transformations only include support for basic execution advice. The more complex call advice are not implemented yet due to time constraints. Since all the foundations have been laid out there should not be any major problems finishing it though.

- There are still some parts of Puma used by the AspectClang weaver. Some are not yet fully replaced such as the final include expansion step. Others require moving existing code from the Puma project into the AspectC++ source tree to remove the dependency on the Puma framework.

Even though the AspectClang weaver does not yet cover all of the features the Puma-based weaver had, many tests could be performed evaluating its performance characteristics and conformance with C++ standards.

- C++ conformance is better across the board. All known Puma bugs regarding rejection of valid C++ code were tested and not a single one of them persists with Clang. Code bases like SystemC that worked with Puma still work with the new parser and of other tested code bases like Qt or some subsets of boost work without much additional tweaking needed. This is a significant improvement over the existing weaver.
- Performance regressed slightly over the Puma-based weaver. Profiling showed some areas where low-hanging optimization opportunities remain; other inefficiencies require in-depth modifications of the design.
- C++11 support was not yet evaluated in depth. Some of the tested code bases, like Qt, make use of C++11 features and those seem to work fine. The weaver itself was not extended to support new C++11 constructs.

In conclusion, AspectClang provides a large improvement over Puma, both in the amount of maintenance required and in C++ conformance. There are still rough edges that have to be fixed before AspectClang can fully replace the existing weaver.

7.1 Future Work

After the implementation of AspectClang is finished, new opportunities that were opened up by the Clang parser can be pursued. Clang provides a wealth of libraries that AspectC++ can take advantage of. This includes things like the GCC-compatible Clang driver which can be used to simplify AspectC++'s `ag++` driver and a rich diagnostic system which the weaver could make use of to improve its error reporting.

Novel ways to handle on the fly injection of code could be explored, including handing aspects more like templates to remove the fragility of manipulating parser state when it does not expect it. With Clang it would also be possible to have the results directly turned into machine code using the LLVM code generator, speeding up the process. This has to remain an option though, as woven AspectC++ should stay compatible with other compilers for various embedded platforms and Microsoft Windows.

Then there is the new C++11 standard, which, according to C++ creator Bjarne Stroustrup, feels “like a whole new language” [23]. Its impact on AspectC++ is yet to be determined. Clang provides the parser foundations that can now be used in the weaver.

Bibliography

[1] *Clang - Features and Goals.* http://clang.llvm.org/features.html, Last checked: August 19, 2013

[2] Spinczyk, Olaf ; Gal, Andreas ; Schröder-Preikschat, Wolfgang: AspectC++: An Aspect-Oriented Extension to C++. In: *In Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002*, 2002, p. 53–60

[3] Kiczales, Gregor ; Hilsdale, Erik ; Hugunin, Jim ; Kersten, Mik ; Palm, Jeffrey ; Griswold, William G.: An Overview of AspectJ, Springer-Verlag, 2001, p. 327–353

[4] Spinczyk, Olaf ; Lohmann, Daniel ; Urban, Matthias: Advances in AOP with AspectC++. In: *Proceedings of the 2005 conference on New Trends in Software Methodologies, Tools and Techniques: Proceedings of the fourth SoMeT W05.* Amsterdam, The Netherlands : IOS Press, 2005. – ISBN 1–58603–556–8, 33–53

[5] Urban, Matthias ; Spinczyk, Olaf: *AspectC++ Language Reference*, April 2011 . http://www.aspectc.org/fileadmin/documentation/ac-languageref.pdf

[6] Urban, Matthias ; Lohmann, Daniel ; Spinczyk, Olaf: The aspect-oriented design of the PUMA C/C++ parser framework. In: *Proceedings of the 9th International Conference on Aspect-Oriented Software Development.* New York, NY, USA : ACM, 2010 (AOSD '10). – ISBN 978–1–60558–958–9, 217–221

[7] Urban, Matthias ; Lohmann, Daniel ; Spinczyk, Olaf: Puma: An Aspect-Oriented Code Analysis and Manipulation Framework for C and C++. Version: 2011. http://dx.doi.org/10.1007/978-3-642-22031-9_5. In: Katz, Shmuel (Hrsg.) ; Mezini, Mira (Hrsg.) ; Schwanninger, Christine (Hrsg.) ; Joosen, Wouter (Hrsg.): *Transactions on Aspect-Oriented Software Development VIII* Bd. 6580. Springer Berlin Heidelberg, 2011. – ISBN 978–3–642–22030–2, 141-162

[8] *PUMA TODO.* https://svn.aspectc.org/repos/Puma/trunk/TODO, Last checked: August 27, 2013

[9] ISO: Information technology – Programming languages – C++ / International Organization for Standardization. Version: 2011. http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=50372. 2011 (ISO/IEC 14882:2011)

[10] Smith, Richard: *Turing machine implementation in constexpr.* http://llvm.org/svn/llvm-project/cfe/trunk/test/SemaCXX/constexpr-turing.cpp, Last checked: August 27, 2013

[11] Gregor, Doug ; Lattner, Chris ; Kremenek, Ted: State of Clang, Presented at the third general meeting of LLVM Developers and Users, 2009

[12] *CUDA LLVM Compiler.* https://developer.nvidia.com/cuda-llvm-compiler, Last checked: August 27, 2013

[13] *Clang++ Builds Boost! - LLVM Project Blog.* http://blog.llvm.org/2010/05/clang-builds-boost.html, Last checked: August 22, 2013

[14] *freebsd-current: Clang now the default on x86.* http://lists.freebsd.org/pipermail/freebsd-current/2012-November/037610.html, Last checked: August 27, 2013

[15] *LLVM Users.* http://llvm.org/Users.html, Last checked: August 27, 2013

[16] *C++98, C++11, and C++14 Support in Clang.* http://clang.llvm.org/cxx_status.html, Last checked: August 19, 2013

[17] *Clang vs GCC (GNU Compiler Collection).* http://clang.llvm.org/comparison.html#gcc, Last checked: August 27, 2013

[18] *How To Release LLVM To The Public.* http://llvm.org/docs/HowToReleaseLLVM.html, Last checked: August 20, 2013

[19] Engel, Michael ; Spinczyk, Olaf: Aspects in hardware: what do they look like? In: *Proceedings of the 2008 AOSD workshop on Aspects, components, and patterns for infrastructure software.* New York, NY, USA : ACM, 2008 (ACP4IS '08). – ISBN 978–1–60558–142–2, 5:1–5:6

[20] *GCC Bugzilla – Bug 33558 'mutable' incorrectly accepted on reference members.* http://gcc.gnu.org/bugzilla/show_bug.cgi?id=33558, Last checked: August 21, 2013

[21] *C++11 in Qt5.* http://woboq.com/blog/cpp11-in-qt5.html, Last checked: August 23, 2013

[22] *Pretokenized Headers (PTH).* http://clang.llvm.org/docs/PTHInternals.html, Last checked: August 20, 2013

[23] Stroustrup, Bjarne: *The C++ Programming Language, 4th Edition.* 4th. Addison-Wesley Professional, 2013. – ISBN 978–0321563842

[24] *"clang" CFE Internals Manual.* http://clang.llvm.org/docs/InternalsManual.html, Last checked: August 20, 2013

List of Figures

List of Tables

Listings