Documentation:
AC++ Compiler Manual

pure-systems GmbH
and Olaf Spinczyk

Version 1.1, December 15, 2007

[1]os@aspectc.org
http://www.aspectc.org

[2]aspectc@pure-systems.com
http://www.pure-systems.com
Agnetenstr. 14
39106 Magdeburg
Germany

Contents

1 Introduction

The program ac++ is a compiler for the AspectC++ programming language. It is implemented is a preprocessor that transforms AspectC++ code into ordinary C++ code. During this transformation aspect code, which is defined by aspects, is woven statically into the component code. Aspects are a special AspectC++ language element, which can be used to implement crosscutting concerns in separate modules. Aspect definitions have to be implemented in special "aspect header files", which normally have the filename extension ".ah". After the code transformation the output of ac++ can be compiled to executable code with ordinary C++ compilers like GNU g++, or Microsoft VisualC++.

More details about the features of the AspectC++ can be found in the in the quick reference sheet and the publications about the language and its application. Everything is available on the AspectC++ homepage http://www.aspectc.org, which is also a source for updates of this manual.

The compiler's source code is freely available from the project's web site and covered by the GPL. For your convenience there are also binary versions of the open source implementation available. Besides the free GPL version, commercial licenses for ac++, the underlying parser and code manipulator library and the ac++ integration into the MS Visual Studio IDE as well as commercial support are available from pure-systems GmbH[1].

This document focuses on how the ac++ compiler works and how it is used. The following sections are structured as follows: Section 2 describes how to get and install the compiler. It is followed by section 3, which describes the two transformation modes of ac++ and the meaning of the command line arguments. Platform-specific notes are given in section 4. It describes the specifics of the ac++ ports and which non-standard features of the back-end C++ compiler are supported. Section 5 lists some known problems, common pitfalls, and unimplemented language features.

2 Download and Installation

Binaries of ac++ for various platforms are available for free download from the AspectC++ homepage (see section 4 for the list and state of the ac++ ports). The versioning scheme is shown in table 1 on the following page.

[1] http://www.pure-systems.com/

Scheme	Example	Kind of Release/Meaning
<version>.<release>	0.7	A regular release 0.7.
<version>.<release>.<fix-no>	0.7.3	Bug fix release number 3 of 0.7
<version>.<release>pre<no>	0.8pre1	Pre-Release number 1 for 0.8

Table 1: Versioning scheme

Besides the archive file with the compiler there is a README file and a CHANGELOG file available for each release. The README file explains the necessary steps for the installation, while the CHANGELOG documents the changes in the corresponding release as well as the history of changes.

The following subsections explain how the current version of the `ac++` compiler is unpacked, installed, and configured. This process depends on the development platform. Skip to the appropriate part from here.

2.1 Linux, Solaris, and MacOS X

The Linux, Solaris, and MacOS X installation procedures are very similar, because all of them belong to the UNIX system family. The `ac++` compiler and the example code is provided in a `gzip`-ed `tar` archive (tgz file). Note that on Solaris the GNU tar command `gtar` must be used instead of the Solaris `tar` to unpack the archive. On Linux and MacOS the archive can be unpacked with the following command in any directory:

```
tar xzvf <tar-file-name>
```

The command creates a directory aspectc++-<version>, which contains the `ac++` binary, the `ag++` front-end, the example code, and everything else that is needed to run the examples like a Makefile. To transform the examples (in the `examples` directory) simply execute `make` in the installation directory. Each example is then transformed from AspectC++ code into C++ code by weaving aspects and saved in `examples/<name>-out`. To run the example, enter the created directory, call `make` and start the executable.

The `Makefile`, which is used to compile the examples uses the command `ag++`, which is a wrapper for calling `ac++`, `g++`, and for the generation of the parser configuration file, which is needed for `ac++`. A separate manual for `ag++` is available from the AspectC++ web site.

2.2 Windows

The Windows port of ac++ supports the freely available Cygwin/GNU g++, MinGW g++, and MS Visual C++ compiler[2] as back-end compilers.

The installation of ac++ in an environment with GNU g++ and make is similar to the UNIX-like installation described in section 2.1. Additionally refer to section 4.2.2 on page 17, which provides some specific information about path names in the Cygwin environment.

A comfortable integration of ac++ into the Visual Studio .NET IDE is available by pure-systems GmbH[3]. The following procedure outlines the installation for windows command line compilers.

The ac++ compiler and the examples are provided in a ZIP archive. Unpack ac++ in a directory of your choice, for instance into C:\AC. The next step is to create a parser configuration file that describes predefined macros and standard include file paths of you back-end compiler. The files pumabc55.cfg and pumavc7.cfg can be taken as examples. An automatic generation of the config file as under UNIX systems is not available at the moment in the free ac++ version.

The examples directory contains various examples that show how to write aspects in AspectC++. You can use the examples.bat batch file to weave all the examples at once. After this step the transformed example files can compiled.

3 Invocation

3.1 Modes

The ac++ compiler support two major transformation modes:

3.1.1 Whole ProgramTransformation (WPT)

WPT mode was the first transformation mode of ac++. However, it is not obsolete, because it may be useful in many cases. In this mode ac++ transforms all files in a project directory tree (or set of directories) and saves the result in a different directory tree. For each translation unit and header file a new file is generated in the target tree with the same name. If further transformations of the source code have to be done, either with ac++ or other tools, this is the mode to choose. Even comments and whitespace remain untouched.

[2] http://www.borland.com/. Note that Borland C++ will probably be discontinued in the near future.

[3] an evaluation version can be downloaded from http://www.pure-systems.com/

The compiler performs a simple dependency check in WPT mode. A translation unit is recompiled if either the translation unit itself or any header file of the project has been changed. This is not very precise but make sure that after changing an aspect header file all translation units are recompiled.

3.1.2 Single Translation Unit (STU)

The new STU mode was introduced with `ac++` version 0.7pre1. Here `ac++` must be called once for each translation unit like a normal C++ compiler. This makes it easier to integrate `ac++` into Makefiles or IDEs. As `ac++` can't save manipulated header files in this mode, because the unchanged header files are needed for the next translation units, all #include directives in the translation unit that refer to header files of the project directory tree are expanded and, thus, saved together with the manipulated translation unit. The resulting files can be fed directly into a C++ compiler. They do not depend on any other files of the project anymore.

In the STU mode the user is responsible for checking the dependencies of changed files and for calling the right `ac++` to transform all translation units that depend on a changed file. The general dependency rule is that a translation unit depends on every header file that is directly or indirectly include and every aspect header that might affect the translation unit (normally all!) and the files they depend on. If you are using `g++` and `make`, checking of this rule can be automized:

```
g++ -E -I<some-path> -MM <trans-unit> -include "*.ah"
```

This call of the `g++` preprocessor prints a makefile dependency rule, which is suitable to determine when `ac++` must be run to rebuild a tranlsation unit.

3.2 Weaving in Library Code

A C++ library consists of header files that have to be included by the client code and an archive file that contains the object code. If the library is implemented in AspectC++ and the client code should not be compiled with `ac++` it is necessary to generate manipulated header files. In the WPT mode this is done anyway. In the STU a directory tree with all manipulated headers can be generated with the `-i` option (see 3.3.7 on page 11).

3.3 Options

Table 2 on the next page summarizes the platform-independent options supported by `ac++`. Platform specific options will be explained in section 4. All options can either be passed as command line arguments or by the configuration file[4], which is referenced by the environment variable PUMA_CONFIG (see section 2). '–' in any of the columns WPT or STU means that this option has no meaning in the corresponding translation mode.

The upper part of the table lists `ac++`-specific options, while the options in the lower part are widely-known from other compilers like `g++`.

3.3.1 `-p|--path <arg>`

This option defines the name of a project directory tree `<arg>`. The option can be used more than once if several directories belong to the project. At least one `-p` options is always need when `ac++` has to transform code, even in STU mode.

3.3.2 `-d|--dest <arg>`

With `-d` a target directory for saving is selected. It corresponds to the last `-p` option. For example, if two directories belong to a project they would be described in WPT mode with

```
-p dir1 -p dir2
```

and in WPT with two source/target pairs:

```
-p source1 -d target1 -p source2 -d target2
```

In STU mode `-d` makes only sense in combination with `-i` to generate header files for a library (see 3.3.7 on page 11).

3.3.3 `-e|--extension <arg>`

In WPT mode `ac++` searches in all project directories for translation units to transform. Translation units are identified by their filename extension. The default is "cc", which means that all files ending with ".cc" are handled. By using the option `-e cpp` or `-e cxx` you can select other frequently used filename extensions. The option can be used more than once, but only the last one is effective.

In WPT mode `ac++` generates a file called `ac_gen.<extension>`. This extension is also taken from the `-e` option, if one is provided.

[4] In the current ac++ version some of these options are not allowed in the config file, namely all between `-v` and `--no_problem...`. This problem will be fixed soon.

Option	WPT	STU	Description
-p|--path <arg>	X	X	Defines a project directory
-e|--extension <arg>	X	–	Filename extension of translation units
-v|--verbose <arg>	X	X	Level of verbosity (0-9)
-c|--compile <arg>	–	X	Name of the input file
-o|--output <arg>	–	X	Name of the output file
-g|--generate	–	X	Generate link-once code
-i|--include_files	–	X	Generate manipulated header files
-a|--aspect_header <arg>	X	X	Name of aspect header file or 0
-r|--repository <arg>	X	X	Name of the project repository
--config <arg>	X	X	Parser configuration file
-k|--keywords	X	X	Allow AspectC++ keywords in normal project files
--no_line	X	X	Disable generation of #line directives
--gen_size_type <arg>	X	X	use a specific string as size_t
--warn...	X	X	enable a weaver warning that is suppressed by default
--no_warn...	X	X	suppress a specific weaver warning
--problem...	X	X	enable back-end compiler problem workaround (see 4.2)
--no_problem...	X	X	disable back-end compiler problem workaround
-I <arg>	X	X	Include file search path
-D <name>[=<value>]	X	X	Macro definitions
-U <name>	X	X	Undefine a macro
--include <arg>	X	X	Forced include

Table 2: ac++ Compiler Option Summary

3.3.4 `-v|--verbose [<arg>]`

The compiler can print message on the standard output device, which describe what it is currently doing. These message can be printed with different levels of datails. You can select this level with the parameter `<arg>`. The range is from `0`, which means no output, to `9`, which means all details. The option `-v0` is the same as having no `-v` option at all. `-v` without `<arg>` is the same as `-v3`.

The `-v` option can be used more than once but only the last one is effective.

3.3.5 `-c|--compile <arg>`

The `-c` option is used to select an input file for `ac++` in the STU mode. Using it more than once is possible, but only one is effective. There are no restrictions on the filename extension. `ac++` expects that the file contains AspectC++ source code.

3.3.6 `-o|--output <arg>`

With the -o option one can select the name of the output file, i.e. the name of the target of the code transformation, in STU mode. If this option is not used, the default output filename is `ac.out`. Note that the output filename is *not* derived from the input file name as it is done by other compilers.

3.3.7 `-i|--include_files`

The `-i` option has to be used if the source code of the project should be compiled into a library and `ac++` should run in STU mode (see 3.2 on page 8). When a translation unit is transformed by using `-c` and `-o` in STU mode no manipulated header files are generated. All include files are expanded within the generated source code. This is fully sufficient if the translation units will then be compiled and linked directly. However, if a library should be provided the client needs a library file (an archive) *and* manipulated header files. These can be generated with `-i`. The generation results in a directory tree with the same structure as the input directory tree specified by `-p` exhibits. Use the `-d` option the select the target directory name(s).

Note that at the moment only and all files with the extension `.h` are considered to be include files. This is rather inflexible and will be improved in future releases.

3.3.8 -a|--aspect_header <arg>

By default ac++ searches all files with the filename extension .ah in the project directory tree(s) and allows all aspects defined in these files to affect the current translation unit. If one looks for a simple mechanism to deactivate aspects at compile-time, if .ah does not conform to your local conventions, or if not all aspects should affect all translation units (be careful! See 5.1 on page 18), the -a option might help.

The option may be used more than once and each of them selects one aspect header that has to be considered for the current translation unit in STU mode or all translation units in WPT mode. If no aspect header should be considered use -a0.

3.3.9 -r|--repository <arg>

The "project repository" is an XML-based description of global information about an AspectC++ development project that is compiled with ac++. It fulfills two purposes:

1. It is a vehicle to transport information from one compiler run to another
2. It might be used be integrated development environments to visualize the join points where aspects affect the component code.[5]

The -r option is used to define the name of the project repository file. However, this is an experimental feature. The file format is volatile. The uniqueness of join point IDs is only guaranteed if the project is compiled with a project repository. If a file with the given name does not exist, ac++ will create a new repository file. If the file exists, but is empty or does not contain valid data, ac++ terminates with an error message. A warning messages will be printed if the version of the weaver, which created the project repository, differs from the current ac++ version.

3.3.10 --config <arg>

Besides setting the environment variable PUMA_CONFIG this options can be used to set the path to the parser configuration file.

[5] In fact, the AspectC++ Development Tools for Eclipse (ACDT) already use the repository to visualize matched join points. See the ACDT homepage http://acdt.aspectc.org/ for information on the ACDT project.

3.3.11 `-k|--keywords`

By default the AspectC++ keywords `aspect`, `pointcut`, `advice`, and `slice` are only treated as keywords in aspect header files. If they are used in normal project files, `ac++` interprets them as normal identifiers. By this design decision aspects can be woven into legacy code even if the code uses the AspectC++ keywords as normal identifiers.

If the AspectC++ keywords should be interpreted as keywords in normal project files as well, the command line option `-k` or `--keywords` has to be used.

In files that do *not* belong to the project, e.g. standard library header files, the AspectC++ keywords are always regarded as normal identifiers, even if `-k` or `--keywords` is used.

3.3.12 `--no_line`

When `ac++` manipulates files, e.g. by inserting generated code, it also inserts `#line` directives. Inserting these directives can be disabled with the `--no_line` option. Normally, `#line` directives are only generated by C preprocessors. The directives are important for back-end compiler error messages and source code debuggers. Without the `#line` generation these numbers correspond to the lines in the generated code, while they correspond to the source code written by the programmer otherwise.

3.3.13 `--gen_size_type <arg>`

`ac++` generates a `new` operator, which has `size_t` in its argument type list. As the generated code shall not include the respective header file (to avoid portability problems), the weaver normally generates the name of the right type. However, in case of cross-compilation the type on the target platform might differ. Then it is possible to provide a string with this option, which is directly used in the constructor's argument list.

3.3.14 `--problem...`

An option like this is used to enabled a back-end compiler-specific code generation workaround. This is sometimes needed, because the C++ compilers differ in their degree of standard conformance. For details about the workarounds needed for each back-end refer to section 4.2.

Warning Name	Condition
deprecated	a deprecated syntax is being used
macro	macro-generated code would have to be transformed

Table 3: ac++ Warnings

3.3.15 --no_problem...

This option can be used to disable a back-end compiler-specific code generation workaround which is enabled by default.

3.3.16 --warn_...

With this option the weaver is instructed to print specific warnings that are otherwise suppressed. Table 3 lists the names of warnings currently supported by the weaver.

3.3.17 --no_warn_...

The warnings listed in table 3 can be suppressed with `--no_warn_`<*Name*>.

3.3.18 -I <arg>

The option `-I` adds the directory `<arg>` to the list of directories to be searched for header files. It can be used more than once and the search order is from the last directory to the first. The compiler `ac++` needs to know all directories, where header files for the current translation unit might be located.

In case of system headers there are often a lot of these directories. To make the setup of `ac++` more convenient we provide the `ag++ --gen_config` command. The command calls the `g++` compiler to get all these paths. A similar mechanism exists for the (commercial) Visual Studio Add-In. Users of Borland C++ or non-supported back-end compilers have to find out this list on their own.

3.3.19 -D <name>[=<value>]

With `-D` a preprocessir macro `<name>` will be defined. Without the optional value assignment the macro will get the value `1`. The option can be used more than once.

In most cases your source code expects some standard macros to be defined like `win32`, `linux`, or `i386`. And even if your code doesn't use them directly, they are often required to be set correctly by system header files. Thus, for the ac++ parser a correct set of these macros has to be defined. For g++ users we provide a command called ag++ that calls the compiler to get the list of these macros. A similar mechanism exists for the (commercial) Visual Studio Add-In. Users of Borland C++ or non-supported back-end compilers have to find out this list on their own.

3.3.20 -U <name>

This option can be used to undefine a previously defined macros.

3.3.21 -include <arg>

The -include option can be used to include a file <arg> into the compiled translation unit(s) even though there is no explicit #include directive given in the source code. If multiple -include options are given on the command line, the files are include in the same order (from left to right). If you use the option in STU mode make sure that the back-end compiler is not forced to include the same files again (read details in 5.1.2 on page 20).

3.4 Examples

- ac++
  Displays all options with a short description.

- ac++ -I examples/Trace -p examples/Trace -d examples/Trace-out
  Transforms the complete project from directory "examples/Trace" into the directory "examples/Trace-out". This is the whole program transformation (WPT) mode, which also performs a simple dependency check.

The following examples describe the compiler like interface (STU Mode). All dependency handling has to be done the user.

- ac++ -c main.cc -p.
  Transforms only the translation unit main.cc. The default name for the output file is ac.out.

- ac++ -c main.cc -o main.acc -p.
  Transforms the file main.cc into the new file main.acc.

- `ac++ -c main.cc -o main.acc -p. -a trace.ah`
  Transforms the file `main.cc` into the new file `main.acc` with the aspect located in `trace.ah`.
- `ac++ -i -v9 -p. -d includes`
  Creates the manipulated project header files and stores it into the directory `includes`. ATTENTION: This works only once, because of the `includes` directory is located in the project directory tree and the aspect header files exists twice now.

4 Platform Notes

4.1 Ports

The `ac++` compiler was originally developed on RedHat Linux systems.Today most of the development is still done under Linux (Debian and RedHat), but Windows has become a second development platform. This means that the Windows and Linux ports are the most tested. The Solaris and MacOS X ports were compiled, because they were demanded by users, but they are far less tested than our development platform ports.

4.1.1 Linux

The `ac++` binary was tested on...

- RedHat 7.3, 8.0, 9
- Debian 3.0
- Suse 8.2, 9, 9.1, 9.3

4.1.2 Windows

Windows systems have different filename conventions than UNIX systems. Although ac++ was originally developed on Linux and does not use or need the Cygwin environment, path names are allowed to contain '`\`' characters and drive names like '`C:`'. The UNIX filename delimiters '`/`' are also accepted.

4.1.3 Solaris

No specific information available, yet.

4.1.4 MacOS X

No specific information available, yet.

4.2 Back-End Compiler Support

The C++ compiler that should be used to compile the output of `ac++` (back-end compiler) plays a very important role for `ac++`, because compilers normally come with header files, which `ac++` must be able to parse. None of the back-end compilers listed here has totally standard conforming header files, which makes it very hard for `ac++` to parse all this code.

GNU `g++` (including Cygwin/GNU g++ under Windows) and MS VC++ are our best supported compilers. After the implementation of a lot of MS VC++ specific extensions, the freely available Borland C++ became less important in the development. Depending on the users' demands it might become an unsupported platform in future releases.

4.2.1 GNU g++

There are a lot of GNU g++ specific C++ extensions as well as several builtin functions and types. To enable all these extensions the option `--gnu` (or `--gnu-2.95` if g++ 2.9x header files should be parsed) has to be used. If a configuration file is generated with `ag++ --gen_config`, this option will be automatically inserted (either `--gnu` or `--gnu-2.95` depending on your compiler).

The `ac++` parser aims at being compatible with g++ and nearly all of the header files that come with g++ 3.x and 2.9x can be parsed. The workaround to install the old g++ 2.95 header and to modify your `puma.config` file so that `ac++` finds these old files while parsing your code is no longer needed starting from version 0.8pre2.

Compilers from the g++ family do not support explicit template specialization in a non-namespace scope. However, this feature is needed by ac++ in the code generation process. A workaround for this problem is automatically enabled when you use the `--gnu` or `--gnu-2.95` option. To explictly enable or disable the workaround use `--problem_spec_scope` or `--no_problem_spec_scope`.

4.2.2 Cygwin/GNU g++

The `ac++` compiler can also be used with the Cygwin/GNU g++ compiler under Windows. Note, that `ac++` itself is not a Cygwin application and, thus, does not support Cygwin-specific path names like `/home/olaf`, which is relative to the cygwin installation directory. If your generate your parser configuration file automatically with `ag++ --gen_config` the contained include paths will automatically be converted from Cygwin paths names to Windows path names using the `cygpath` command. However, be careful when you set the `PUMA_CONFIG` environment variable or when you pass any other path name to `ac++`.

4.2.3 MS VC++

The `ac++` parser aims at being compatible with Microsoft Visual C++ 7. This compiler comes with a large number of non-standard language extensions. To enable support for these extensions in the `ac++` parser the command line option `--vc` must be provided either on the command line or in your configuration file.

It is not recommended to use `ac++` with Visual C++ 6 as this compiler has some problems with the generated code, even though the generated code is standard complient.

We recently found a bug in Visual C++ 6 and 7, which is related to local classes defined in header files. As `ac++` sometimes generates such classes a workaround has to be enabled until Microsoft fixes the problem. The workaround can be enabled with the command line option `--problem_local_class` and disabled with `--no_problem_local_class`. In the current `ac++` version the workaround is enabled by default if the executable was compiled for the Windows platform.

4.2.4 Borland C++

The Borland C++ compiler is not well supported by `ac++`. Probably the support will be completely discontinued in the near future.

5 Problems & Workarounds

5.1 Common Pifalls

5.1.1 Include Cycles

In versions prior to 1.0pre1 include cycles could occur in many situations and workarounds could not always be found. In version 1.0pre1 include cycles can only occur in the case of aspect code with introductions. Advice for code join points cannot produce cycles.

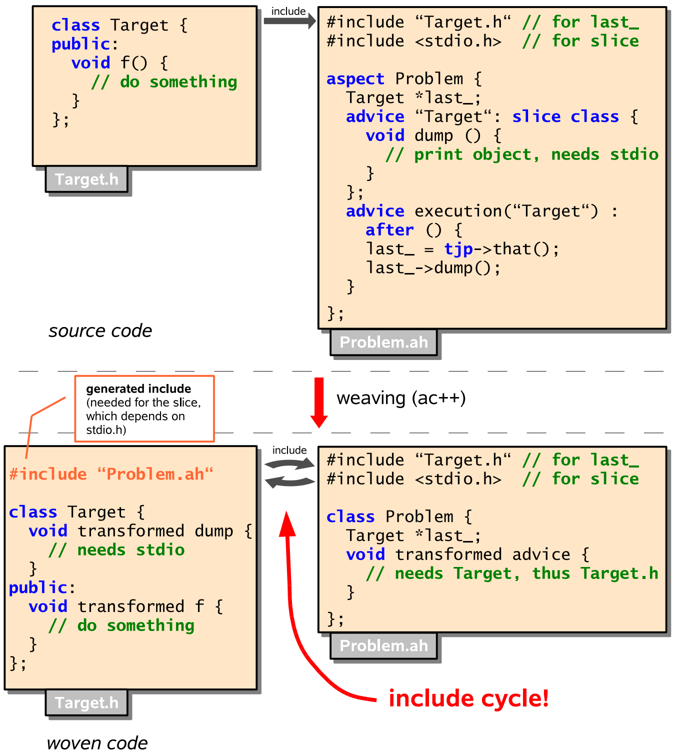

Figure 1: Include cycle problem

The reason for the remaining possible cycles is that `ac++` generates `#include <aspect-header>` in every file that contains the definition of a target class of an introduction. Without this generation pattern definitions from the aspect header would not be accessable by introduced code. However, if the aspect header directly or indirectly includes the target file, there is a cycle, which might cause parse errors.

Figure 1 illustrates the include cycle problem by giving an example. Here an aspect `Problem` uses the type `Target` and therefore includes `Target.h`. At the same time the aspect introduces a slice into the class `Target`. As the slice might depend on definitions or `#include`s in `Problem.ah`, the weaver generates the `#include "Problem.ah"` in `Target.h`. This causes the include cycle. Include guards (which should always be used!) avoid duplicate definitions, but do not solve the problem. It might still be the case that the parser complains about undefined types.

To avoid these cycles introductions can always be separated from the aspect by means of slices. Slice declarations and slice member definitions can be located in arbitrary aspect header files. The aspect weaver will only include these aspect headers in the target classes' header/implementation files and thereby avoid the cycle. For a slice reference within an advice declaration even a forward

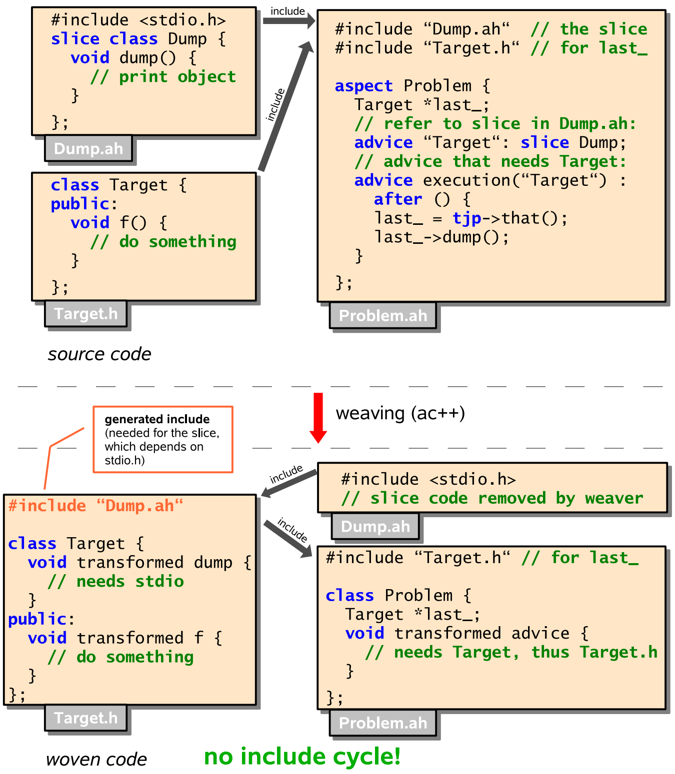

Figure 2: Include cycle avoidance

declaration of the slice is sufficient.

Figure 2 shows how the include cycle from the example in Figure 1 can be avoided. Here the function `dump` is implemented in a separate slice class `Dump` that is stored in an aspect header file `Dump.ah`. The implementation of `Dump` could rely on defintions and `#include`s in `Dump.ah` (`stdio.h` in this example), but not on defintions in `Problem.ah`. Therefore, the aspect weaver generates `#include "Dump.ah"` and not `#include "Problem.ah"` in `Target.h`.

Note that in the case of **non-inline** introductions the #include directive is generated in the file that contains the “link-once element” of the target class, which is never a header file. You can, for example, exploit this feature to produce cyclic class relationships. The included file will be the aspect header file that contains the definition of the non-inline slice member.

5.1.2 Duplicate Forced Includes in STU Mode

In the Single Translation Unit (STU) mode `ac++` handles forced includes (see `-include` option in section 3.3.21 on page 15) in the following way:

internal includes: If the included file is part of your project, the file content will be expanded in the compiled translation unit.

external includes: If the included file is *not* part of the project, ac++ generates an #include directive with the absolute path name of the file.

In both case the back-end compiler should not be forced to include the same file again. For example, g++ users should not use the -include option with ac++ and with g++, because otherwise symbols might be defined twice.

5.1.3 TODO in this manual:

- Generating header files into the project tree
- Different macros for translation units while parsing in WPT mode
- -a 0 option
- -a: different instances of the same header files

5.2 Unimplemented Features

5.2.1 Multi-Threading Support

C++ has no integrated thread modell like Java. Therefore, the woven AspectC++ code cannot rely on any available thread synchronization mechanism. As a result the implementation of the cflow pointcut functions is currently *not thread-safe*.

We are urgently investigating how thread synchronization and thread local storage can be integrated into AspectC++.

5.2.2 Parse Errors

If ac++ stops processing because of parse errors this might be due to an incompatibility or missing feature in the underlying C++ parser.

In the case that the error is found in your own code, i.e. code you are able to modify, you could use the following workaround:

```
#ifdef __puma
// ... simplified version of the code for ac++
#else
// ... original code
#endif
```

Even if your own AspectC++ code contains only harmless C++ code you might experience parsing problems due to header files from libraries which your application code includes, especially in the case of template libraries. In this situation it might help to copy the file with the parse error into a different directory. Then you have to change the code in this file to avoid the error message by simplifying it. The final step is to extend the `puma.config` file by a "-I <path>" entry for the directory where you placed the copy. As the result `ac++` will now parse the simplified version while the original file is untouched and used while the C++ compiler runs.

5.2.3 Templates

Currently ac++ is able to parse a lot of the (really highly complicated) C++ templates, but weaving is restricted to non-templated code only. That means you can not weave in templates or even affect calls to template functions or members of template classes. However, template instances can be matched by match expressions in the pointcut language and calls to members of class templates or template functions can be affected by advice.

5.2.4 Unimplemented Language Elements

`set` and **`get`** are the most demanded unimplemented language features. The discussion about a useful semantics of these pointcut functions in AspectC++ is still in progress.

5.2.5 Support for Plain C Code

Currently `ac++` generates C++ code, which cannot be compiled by a C compiler. As for many hardware platforms in the embedded domain no C++ compiler is available we are actively looking for a solution.

5.2.6 Miscellaneous

- `ac++` currently expects that header files have the extension ".h". Future version will solve this problem by adding a further command line option.

5.2.7 Constructor/Destructor Generation

If advice for construction/destruction joinpoints is given and no constructor/destructor is defined explicitly, ac++ will generate it. However, currently ac++ assumes that the copy constructor has one argument of type "`const <Classname>&`". This leads to problems if the implicitly declared copy constructor has an argument of type "`<Classname>&`". Therefore, you should not define construction/destruction advice for classes with this copy constructor signature.

6 Code Transformation Patterns

This appendix documents some internals of the `ac++` weaver implementation.

6.1 Inclusion of Aspect Header Files

The weaver has to guarantee that aspect header files are only compiled in a translation unit if they are affecting the shadows of code join point that are located within the translation unit. If an aspect header has to be included because of this reason, the same check have to be performed again, because the aspect header might contain code join points that have to be affected by other aspects.

In order to implement this behavior a forward declaration of the advice invocation function is generated and a macro `__ac_need_`*<mangled_ah_filename>* is defined in each file that contains a join point shadow, which is affected by an aspect that is defined in an aspect header whose mangled files name is *<mangled_ah_filename>*. Multiple inclusions shall be avoided. Therefore, another macro `__ac_have_`*<mangled_ah_filename>* is set wherever an aspect header is included by generated code. The following code is an example that shows the code which is generated at the end of each translation unit for each known aspect header of the project:

```
#ifdef __ac_need_<mangled_ah_1>
#ifndef __ac_have_<mangled_ah_1>
#define __ac_have_<mangled_ah_1>
#include "ah_1"
#endif
// other aspect headers that are needed if ah_1 is needed
#ifndef __ac_have_<mangled_ah_4>
#define __ac_have_<mangled_ah_4>
#include "ah_4"
#endif
#endif // __ac_need_<mangled_ah_1>
```

This code transformation pattern might result in multiple `#include` directives for the same aspect header files. This is correct, as there might be cyclic dependencies between the aspect headers.