CIF Documentation

Evgeny Novikov, Ilya Shchepetkov

Aug 08, 2022

CONTENTS

CIF (C Instrumentation Framework) is a tool that implements aspect-oriented programming for the C programming language. You can learn more about CIF at the project site.

CHAPTER

ONE

CONTENTS

1.1 Deployment

You can download archives with CIF binaries prepared in advance either from the official project site or from artifacts attached to GitHub Actions. Also, you can build CIF yourself from scratch according to instructions below. Section *Building debug version of Aspectator* describes various variants of development builds.

1.1.1 Build dependencies

To build CIF you need to install the following packages:

- make
- gcc
- g++
- flex
- bison

1.1.2 Build and install

First you need to download archives with the source code of prerequisites needed by GCC (*gmp*, *mpfr*, *mpc* and *isl*):

```
$ cd aspectator
$ ./contrib/download_prerequisites
```

Then return back to the root of the repository and execute *make*:

```
$ cd ..
$ make
```

You can use option *-jN* for *make* to significantly speed up building, e.g.:

```
$ make -j16
```

In addition, you can speed up building further by disabling bootstrap:

```
$ ASPECTATOR_CONFIGURE_OPTS="--disable-bootstrap" make -j16
```

After successful build you can install CIF, e.g.:

```
$ sudo make install
```

You can specify the alternative directory where CIF will be installed, e.g.:

```
$ DESTDIR=/home/user/cif make install
```

1.1.3 Automatic testing

You can run the following command for automatic testing of CIF:

```
$ make test
```

It requires Python 3 and pytest to be installed.

1.1.4 Uninstall

You can uninstall CIF by running the following command:

```
$ sudo make uninstall
```

If CIF was installed into an alternative directory with the *DESTDIR* option then you need to use it again:

```
$ DESTDIR=/home/user/cif make uninstall
```

1.1.5 Cleanup

You should run the following command to remove build directories:

```
$ make clean
```

1.2 Tutorial

This section describes several typical cases of using CIF as well as current most vital limitations. CIF has a bunch of command-line options that you can investigate by running it with *-h* or *--help*. In the given tutorial we will consider only the following ones:

- *--in* – a path to a C source file to be processed.
- *--aspect* – a path to an aspect file. Below there will be several examples of aspect files.
- *--out* – a path where a result will be placed.
- *--back-end* – a kind of a back-end to be used, e.g. 'src' or 'bin'.

If you are going to try provided examples, we recommend to change your current directory to `docs/samples` within the source tree root. Hereinafter all file paths and commands will be relative to that directory.

For all use cases below we will consider as an input the C source file presented in Listing 1.1. You can find it here: `calculate-max-rectangle-square.c`.

Listing 1.1: Input C source file

```c
/* This is a very simple program that finds out a rectangle with a maximum square from a␣
↪provided list of rectangle
   heights and widths. It is intended only for demonstration of CIF capabilities. Please,
↪ do not use it anywhere since
   it contains several issues.
   The program expects the following input:
       height1 width1 height2 width2 ... heightN widthN
   where heighti and widthi should be integers representing respectively height and␣
↪width of ith rectangle. */
#include <stdio.h>
#include <stdlib.h>

#define MAX(a, b) (a > b ? a : b)

struct rectangle
{
    unsigned int height;
    unsigned int width;
    unsigned int square;
};

unsigned int calculate_rectangle_square(struct rectangle *r)
{
    r->square = r->height * r->width;
    return r->square;
}

int main(int argc, const char *argv[])
{
    unsigned int rectangles_num;
    struct rectangle *rectangles;
    unsigned int cur_max_rectangle_square = 0;

    rectangles_num = (unsigned int)(argc / 2);
    rectangles = calloc(rectangles_num, sizeof(*rectangles));

    for (int i = 0; i < rectangles_num; i++) {
        struct rectangle *cur_rectangle = rectangles + i;
        cur_rectangle->height = atoi(argv[2 * i + 1]);
        cur_rectangle->width = atoi(argv[2 * i + 2]);
        calculate_rectangle_square(cur_rectangle);
        cur_max_rectangle_square = MAX(cur_max_rectangle_square, cur_rectangle->square);
    }

    printf("Maximum rectangle square is %u\n", cur_max_rectangle_square);

    return 0;
}
```

Normal compilation and running of this program will result in the following output:

```
$ gcc calculate-max-rectangle-square.c -o calculate-max-rectangle-square
$ ./calculate-max-rectangle-square 2 5 7 3 4 4
Maximum rectangle square is 21
```

1.2.1 Weaving function calls

This is the most typical use case. Listing 1.2 provides an example of an appropriate aspect file located in `weave-func-calculate-rectangle-square.aspect`.

Listing 1.2: Aspect file intended for weaving calls of function **calculate_rectangle_square()**

```
/* Introduce extra checking and debugging for calls of function calculate_rectangle_
↪square(). */
around: call(unsigned int calculate_rectangle_square(struct rectangle *r))
{
    unsigned int tmp, res;

    // Check for possible overflow.
    tmp  = r->height * r->width;
    if (r->height != 0 && tmp / r->height != r->width)
        printf("After multiplication of %u and %u there will be overflow, so you can get␣
↪invalid result\n", r->height, r->width);

    // Invoke woven in function itself.
    res = $proceed;

    // Debug its result.
    printf("Calculated rectangle square is %u (%u * %u)\n", res, r->height, r->width);

    return res;
}
```

To weave in the target C source file with the given aspect you can run the following command:

```
$ ../../inst/bin/cif --in calculate-max-rectangle-square.c --aspect weave-func-calculate-
↪rectangle-square.aspect --out calculate-max-rectangle-square --back-end bin
```

Then you will get the following output when running the generated program binary:

```
$ ./calculate-max-rectangle-square 2 5 7 3 4 4
Calculated rectangle square is 10 (2 * 5)
Calculated rectangle square is 21 (7 * 3)
Calculated rectangle square is 16 (4 * 4)
Maximum rectangle square is 21
```

This demonstrates debugging and logging facilities of CIF. Probably by some reason you would not like to add an appropriate code directly to program's source files. So you can use aspect files instead in a similar way.

The same aspect also enables extra checking. Therefore, you can get the following warning if you will intentionally violate implicit assumptions regarding possible multiplication overflows:

```
$ ./calculate-max-rectangle-square 4317358 6927345
After multiplication of 4317358 and 6927345 there will be overflow, so you can get␣
↪invalid result
Calculated rectangle square is 1971072462 (4317358 * 6927345)
Maximum rectangle square is 1971072462
```

(the valid result should be 29907828354510).

1.2.2 Weaving macros

Sometimes it may be necessary to change macros. CIF is capable to do that. `weave-macro-max.aspect` in Listing 1.3 contains an example that add extra debugging for macro *MAX*.

Listing 1.3: Aspect file intended for weaving macro **MAX**

```
/* Notify when current maximum value is changed. It is assumed that "a" holds that value.
↪ */
around: define(MAX(a, b))
{
({unsigned int __max; if (a > b) {__max = a;} else {printf("Update maximum value from %u␣
↪to %u\n", a, b); __max = b;} __max;})
}
```

On executing following commands you will get the output as follows:

```
$ ../../inst/bin/cif --in calculate-max-rectangle-square.c --aspect weave-macro-max.
↪aspect --out calculate-max-rectangle-square --back-end bin
$ ./calculate-max-rectangle-square 2 5 7 3 4 4
Update maximum value from 0 to 10
Update maximum value from 10 to 21
Maximum rectangle square is 21
```

1.2.3 Weaving variables

Listing 1.4 shows how to weave in variable assignments. The corresponding aspect file is `weave-var-rectangles-num.aspect`.

Listing 1.4: Aspect file intended for weaving assignments of variable **rectangles_num**

```
/* Do not consider last rectangle. */
after: set(unsigned int rectangles_num)
{
    $res--;
}
```

This aspect makes pretty artificial change (it excludes the last rectangle from calculations), but you can have some more important things to do, e.g. you can dump all values assigned to a given variable.

To test this aspect you can run the following commands:

```
$ ../../inst/bin/cif --in calculate-max-rectangle-square.c --aspect weave-var-rectangles-
↪num.aspect --out calculate-max-rectangle-square --back-end bin
$ ./calculate-max-rectangle-square 2 5 7 3 4 4 8 9
Maximum rectangle square is 21
```

1.2.4 Weaving compound types

CIF suggests means to modify compound types such as structures, unions and enumerations. For instance, you can find an example of an appropriate aspect in Listing 1.5 (`weave-struct-rectangle.aspect` in `docs/samples`).

Listing 1.5: Aspect file intended for weaving structure **rectangle**

```
/* It does not work at the moment. */
before: introduce(struct rectangle)
{
    unsigned int perimeter;
}

before: call(unsigned int calculate_rectangle_square(struct rectangle *r))
{
    /* Calculate, store and print rectangle perimeter in addition to square. */
    r->perimeter = r->height + r->width;
    printf("Calculated rectangle perimeter is %u (%u * %u)\n", r->perimeter, r->height,␣
↪r->width);
}
```

This aspect adds extra field *perimeter* to the definition of structure *rectangle*. Besides, through weaving of function *calculate_rectangle_square()* it calculates, stores and prints out perimeters for all rectangles.

To test this aspect you can run the following commands:

```
$ ../../inst/bin/cif --in calculate-max-rectangle-square.c --aspect weave-struct-
↪rectangle.aspect --out calculate-max-rectangle-square --back-end bin
$ ./calculate-max-rectangle-square 2 5 7 3 4 4 8 9
Calculated rectangle perimeter is 7 (2 * 5)
Calculated rectangle perimeter is 10 (7 * 3)
Calculated rectangle perimeter is 8 (4 * 4)
Calculated rectangle perimeter is 17 (8 * 9)
Maximum rectangle square is 72
```

1.2.5 Querying source code

CIF can execute different queries to target source files. For instance, you can use aspect `query-func-calls.aspect` shown in Listing 1.6 to find out all function calls.

Listing 1.6: Aspect file demonstrating source code queries

```
/* Query all function calls and print some information on them. */
query: call($ $(..))
{
```

(continues on next page)

(continued from previous page)

```
    $fprintf<"func-calls.txt", "Function %s() is called at line %d\n", $func_name, $call_
↪line>
}
```

This aspect file will not affect your program. You will only get an extra file at the weaving stage. For instance:

```
$ ../../inst/bin/cif --in calculate-max-rectangle-square.c --aspect query-func-calls.
↪aspect --out calculate-max-rectangle-square --stage instrumentation
$ cat func-calls.txt
Function calloc() is called at line 32
Function atoi() is called at line 36
Function atoi() is called at line 37
Function calculate_rectangle_square() is called at line 38
Function printf() is called at line 42
```

(we asked CIF to stop after stage *instrumentation* since we would not like to get the program binary in this case).

1.2.6 Invalid aspects

You can wonder how to track various issues with aspects. First of all, CIF will fail and report appropriate errors if you will provide syntactically invalid aspects. Sometimes, aspects can have valid syntax, but they might not work as expected. Listing 1.7 presents the content of `weave-invalid-func-decl.aspect`. Therein we deliberately specified invalid declaration for function *calculate_rectangle_square()*.

Listing 1.7: Aspect file using invalid target function declaration

```
/* Declaration of function calculate_rectangle_square() is not valid intentionally. */
around: call(struct rectangle *calculate_rectangle_square(unsigned int, unsigned int))
{
}
```

You will not get any warnings if you will run CIF as usual:

```
$ ../../inst/bin/cif --in calculate-max-rectangle-square.c --aspect weave-invalid-func-
↪decl.aspect --out calculate-max-rectangle-square --back-end bin
```

But if you will set environment variable `LDV_PRINT_SIGNATURE_OF_MATCHED_BY_NAME` the situation will change:

```
$ export LDV_PRINT_SIGNATURE_OF_MATCHED_BY_NAME=1
$ ../../inst/bin/cif --in calculate-max-rectangle-square.c --aspect weave-invalid-func-
↪decl.aspect --out calculate-max-rectangle-square --back-end bin

These functions were matched by name but have different signatures:
  source function declaration: unsigned int calculate_rectangle_square (struct rectangle␣
↪*r)
  aspect function declaration: struct rectangle *calculate_rectangle_square (unsigned␣
↪int, unsigned int)
```

1.2.7 Getting woven source files

It may be useful for debugging and necessary for some applications to get woven source files rather than binaries as an output. For instance, you can slightly change the command for the first use case (note the change of *bin* to *src* for command-line option *--back-end*):

```
$ ../../inst/bin/cif --in calculate-max-rectangle-square.c --aspect weave-func-calculate-
↪rectangle-square.aspect --out calculate-max-rectangle-square --back-end src
```

and investigate outputted file `calculate-max-rectangle-square` that will be a C source file.

1.2.8 Further study

CIF has much more capabilities in addition to the ones that we presented in this tutorial. You can read *Aspect-Oriented C* that describes the aspect-oriented extension of the C programming language to study all possible ways of using CIF. Besides, you can find a lot of examples of aspects in projects Klever (in particular, here and here) and Clade (here).

1.2.9 Known issues

CIF is not used very widely, so there is a lot of different issues with it. You can find the known issues in the official issue tracker. The most vital ones are as follows:

- CIF does not have a command-line interface that is compatible with a compiler (#6829). Thus, you can not easily incorporate it into your program's build process.
- CIF does not support multiple advices for the same join point (#358).
- CIF does not support well the entire C programming language with all GCC compiler extensions. Other compiler extensions are supported to the extent that it is done by GCC itself (you can find some related command-line options here).
- CIF is not particularly optimized. It is noticeable if it is called to handle hundreds or thousands of files.

1.3 Aspect-Oriented C

1.3.1 Introduction

This section presents an aspect-oriented extension of the C programming language (hereinafter AOC (Aspect-Oriented C)). This extension allows you to extract cross-cutting concerns of programs into separate modules, so-called *aspects*, consisting of a set of *advices* primarily.

You can implement cross-cutting concerns within *advice bodies* using any correct C code suitable for function bodies. Also, you can use GCC compiler extensions and a set of *special directives*. *Advices* include *pointucts* to specify *join points* of the program for which it is necessary to execute this code. For instance, AOC deals with definitions and substitutions of macros as well as definitions and declarations of functions, variables, and composite types as join points. In order to simplify the development of aspects, macros and declarations of functions, variables, and types used to describe join points generally coincide in syntax, constraints, and semantics with the corresponding constructions of the C programming language with GCC compiler extensions (see sections *Macros* and *Declarations of functions, variables, and types* for details). You can see an example of an aspect in Listing 1.8.

Listing 1.8: Example of an aspect with two advices

```
before: call(void lock(void))
{
    if (locks_counter)
        abort();
    locks_counter++;
}

before: call(void unlock(void))
{
    if (!locks_counter)
        abort();
    locks_counter--;
}
```

Before parsing aspects, *aspect preprocessing* is carried out. Aspect preprocessing behaves exactly in the same way as preprocessing performed by the GCC compiler except for symbol @ is treated instead of #. Listing 1.9 exemplifies using such preprocessor directives as macros and conditionals in the aspect. The corresponding preprocessed aspect is shown in Listing 1.10.

Listing 1.9: Example of using preprocessor macros and conditionals in an aspect

```
@define LOG_FILE "work/info.txt"
@define GET get_property

@if defined DEBUG
@define LOG(action, prop) $fprintf<LOG_FILE, "%s property %s\n", action, prop>
@else
@define LOG(action, prop)
@endif

query: call(int GET(const char *))
{
    LOG("get", $arg_sign1);
}
```

Listing 1.10: Preprocessed aspect

```
# 10 "aspect-macros-and-conditionals.aspect"
query: call(int get_property(const char *))
{
  $fprintf<"work/info.txt", "%s property %s\n", "get", $arg_sign1>;
}
```

In addition to using preprocessor macros and conditionals in aspects, you can also include aspects like C source files include headers. Listing 1.11 demonstrates an example of using **@include** in the aspect. The included aspect is shown above in Listing 1.9. The corresponding preprocessed aspect is shown in Listing 1.12.

Listing 1.11: Example of including an aspect

```
@define PRODUCTION
@include "aspect-macros-and-conditionals.aspect"
```

Listing 1.12: Preprocessed aspect

```
# 10 "aspect-macros-and-conditionals.aspect"
query: call(int get_property(const char *))
{
  ;
}
```

Similarly to the C programming language, you can use *comments* in aspects. Unlike C, not all comments are eliminated at aspect preprocessing. This is the case for comments used in *advice bodies*. For instance, in this way you can implement so-called model comments explaining particular actions and checks performed by requirement specifications.

In addition to the possibility to describe cross-cutting concerns in the form of aspects, AOC assumes means for automatic linkage of aspects with source files of the target program. This process is referred to as *aspect weaving*. In effect, for some representation of program source files, it searches for join points corresponding to advice pointcuts specified in the aspect. In case matches are found, join points are framed with the code specified in *advice bodies* (you can see section *Advices* for more insights). Eventually you can get either woven in program source files or their compiled versions.

Following subsections present a formal grammar of AOC. We use the following notation. Nonterminals are bold and they may be links to appropriate definitions, e.g. *pointcut*, while terminals are enclosed in double quotes, e.g. "call"[1]. :== following a nonterminal represents a definition of this nonterminal. Various variants of a nonterminal definition are either placed on separate lines or separated by |. In nonterminal definitions optional nonterminals are enclosed in square brackets, e.g. [*pointer*].

Note: Keep in mind that the actual implementation may be slightly inconsistent with the given description. Some things may be missed while it can bring extra functionality. You can find known issues in the official issue tracker. Please, do not hesitate to report other ones.

1.3.2 Tokens

Syntax

```
aoc-token  ::=  c-or-aoc-keyword
                aoc-identifier
                aoc-integer-constant
                aoc-string-literal
                c-or-aoc-punctuator
                file-name
                advice-body
                location-control-directive
                comment
```

[1] Double quotes themselves are framed by single quotes like '"'.

Constraints

Compared to `token` defined in 6.4 of [ISO-9899-2011], *aoc-token* has the following amendments:

- Modified set of keywords *c-or-aoc-keyword* is used instead of `keyword` (*Keywords*).
- *aoc-identifier* replaces `identifier` (*Identifiers*).
- AOC supports only integer constants *aoc-integer-constant* rather than `constant` (*Integer constants*).
- `string-literal` is replaced by *aoc-string-literal* (*String literals*).
- *aoc-punctuator* is used instead of `punctuator` (*Punctuators*).

In addition, *aoc-token* supports:

- *file-name* (*File names*).
- *advice-body* (*Advice bodies*).
- *location-control-directive* (*Location control directives*).
- `comment` (*Comments*).

We do not describe `preprocessing-token` presented in 6.4 of [ISO-9899-2011] according to the remark on aspect preprocessing given in *Introduction*.

1.3.3 Keywords

Syntax

```
c-or-aoc-keyword   ::=   c-keyword
                         aoc-keyword
c-keyword          ::=   "auto" | "char" | "const" | "double"
                         "enum" | "extern" | "float" | "inline"
                         "int" | "long" | "register" | "restrict"
                         "short" | "signed" | "static" | "struct"
                         "typedef" | "union" | "unsigned" | "void"
                         "volatile" | "_Bool" | "_Complex" | "_Imaginary"
aoc-keyword        ::=   "after" | "around" | "before" | "call"
                         "declare_func" | "define" | "execution" | "expand"
                         "file" | "get" | "get_global" | "get_local"
                         "infile" | "infunc" | "info" | "introduce"
                         "new" | "pointcut" | "set" | "set_global"
                         "set_local" | "query"
```

Constraints

In comparison with `keyword` presented in 6.4.1 of [ISO-9899-2011] in AOC *c-or-aoc-keyword* can be either a *c-keyword* keyword or an AOC *aoc-keyword* keyword. *c-keyword* does not support "break", "case", "continue", "default", "do", "else", "for", "goto", "if", "return", "switch" and "while", i.e. those keywords that can only be used in C statements and expressions. You still can use them in *advice bodies*, but they are not parsed at aspect weaving.

aoc-keyword is the definition of AOC keywords. It supports:

- "after", "around", "before", "info", "new" and "query" (*Advices*);
- "call", "define", "declare_func", "execution", "expand", "file", "get", "get_global", "get_local", "infile", "infunc",

"introduce", "pointcut", "set", "set_global" and "set_local" (*Pointcuts*).

Semantics

Basically the semantics of keywords *c-or-aoc-keyword* corresponds to the semantics of `keyword` described in 6.4.1 of [ISO-9899-2011]. An important difference is that a word can be *aoc-keyword* only outside of *comments*, *advice bodies*, *macros* and *declarations of functions, variables, and types*. Besides, only words used in *declarations of functions, variables, and types* can represent keywords of the C programming language.

1.3.4 Identifiers

Syntax

```
aoc-identifier           ::=   aoc-identifier-nondigit
                               aoc-identifier aoc-identifier-nondigit
                               aoc-identifier digit
aoc-identifier-nondigit  ::=   nondigit
                               "$"
```

Constraints

Nonterminals `digit` and `nondigit` are defined in 6.4.2 of [ISO-9899-2011]. Compared to `identifier`, which is presented in 6.4.2 of [ISO-9899-2011], AOC *aoc-identifier* supports modified set of non-digital characters *aoc-identifier-nondigit* instead of `identifier-nondigit`.

aoc-identifier-nondigit does not support universal character names `universal-character-name` and any other characters. Additionally, *aoc-identifier-nondigit* supports wildcard "$" (take into account that the **$** symbol is not included in the standard sets of non-digital characters `nondigit` and digits `digit`). We will consider other constraints related to "$" in following sections.

Semantics

In general the semantics of *aoc-identifier* corresponds to the semantics of `identifier` described in 6.4.2 of [ISO-9899-2011]. Each "$" wildcard in *aoc-identifier* corresponds to a sequence of characters (both `digit` and `nondigit`) of arbitrary length[2]. For instance, *aoc-identifier* **$_property$** will match such identifiers as **get_property**, **set_property** and **get_property_value**, but it will not match, say, **receive_message**. If several "$" wildcards are contiguous in the same identifier, they are treated as one "$". An identifier is not converted to a keyword if it uses at least one "$" wildcard. Following sections describe specific semantics of "$" wildcards for certain entities.

[2] Everywhere in this document an arbitrary length includes zero.

1.3.5 Integer constants

Syntax

```
aoc-integer-constant   ::=   decimal-constant
```

Constraints

Nonterminal `decimal-constant` is defined in 6.4.4.1 of [ISO-9899-2011]. Compared to `integer-constant` defined in 6.4.4.1 of [ISO-9899-2011], in AOC *aoc-integer-constant* does not support:

- `octal-constant`.
- `hexadecimal-constant`.
- `integer-suffix`.

Semantics

aoc-integer-constant dumbs down `integer-constant` presented in 6.4.4.1 of [ISO-9899-2011]. Appropriate integer constants are always stored in a variable with the *unsigned int* type (standard type conversion rules are applied in case of overflows).

1.3.6 String literals

Syntax

```
aoc-string-literal   ::=   '"' [s-char-sequence] '"'
```

Constraints

Nonterminal `s-char-sequence` is defined in 6.4.5 of [ISO-9899-2011]. Compared to `string-literal` specified in 6.4.5 of [ISO-9899-2011], *aoc-string-literal* does not support wide string literals L" s-char-sequenceopt ".

Semantics

aoc-string-literal is a simplification of `string-literal` presented in 6.4.5 of [ISO-9899-2011].

1.3.7 Punctuators

Syntax

```
c-or-aoc-punctuator   ::=   c-punctuator
                            aoc-punctuator
c-punctuator          ::=   "(" | ")" | "[" | "]" | "*" | "..." | "," | "$" | ".."
aoc-punctuator        ::=   "(" | ")" | ":" | "!" | "&&" | "||"
```

Constraints

In comparison with `punctuator`, which is presented in 6.4.6 of [ISO-9899-2011], in AOC *c-or-aoc-punctuator* can be either punctuator of the C programming language *c-punctuator*, or AOC punctuator *aoc-punctuator*. The definition of *c-punctuator* supports only "(", ")", "[", "]", "*", "..." and "," from the `punctuator` definition, i.e. those punctuators that can be used when writing *macros* and *declarations of functions, variables, and types*. Besides, *c-punctuator* supports following extra punctuators:

- "$" – a universal type specifier or a universal array size (*Declarations of functions, variables, and types*).
- ".." – a list of arbitrary parameters of a macro function or a function of arbitrary length (see *Macros* and *Declarations of functions, variables, and types* for more details).

The *aoc-punctuator* definition includes:

- ":" – it introduces a definition of a *named pointcut* or *advice*.
- "(", ")", "!", "&&", "||" – these punctuators are for the sake of development of *composite pointcuts*.
- "(", ")" – braces also separate *macros* and *declarations of functions, variables, and types* from descriptions of *pointcuts* and *advices*.

Semantics

The semantics of *c-or-aoc-punctuator* generally corresponds to the semantics of `punctuator` described in 6.4.6 of [ISO-9899-2011]. A vital difference is that a punctuator can be *aoc-punctuator* only outside of *comments*, *advice bodies*, *macros* and *declarations of functions, variables, and types*. Besides, only punctuators used in macros and declarations of functions, variables and composite types are considered as punctuators of the C programming language (*Macros* and *Declarations of functions, variables, and types*). The semantics of additional punctuators of *c-punctuator* is discussed in detail in sections *Macros* and *Declarations of functions, variables, and types*. Sections *Pointcuts* and *Advices* delves into the semantics of *aoc-punctuator*. We do not consider punctuators used in *special directives* here, because they have no meaning outside the context of special directives that are parsed in a special way.

1.3.8 File names

Syntax

```
file-name  ::=   '"' q-char-sequence '"'
```

Constraints

The `q-char-sequence` nonterminal is defined in 6.4.7 of [ISO-9899-2011].

Semantics

Basically the semantics of *file-name* corresponds to the semantics of header-name described in 6.4.7 of [ISO-9899-2011]. Some specific character sequences in file names are interpreted as follows:

- One or more **$$**[3]. Each **$$** corresponds to sequence of q-characters q-char-sequence of arbitrary length. If several **$$** are contiguous in the same file name, they are treated as one **$$**.
- Special directive **$this** that can be used only to indicate the file name and only in the form of "**$this**" (*Special directives*).
- Special directives with predefined values (see *Special directives* for more details).

Note: Generally speaking, one can use **$** characters in file names but this is not considered in AOC.

1.3.9 Advice bodies

Syntax

```
advice-body  ::=  "{" compound-statement-with-comments-and-special-directives "}"
```

Constraints

advice-body represents a C code enclosed in curly braces. It is similar to compound-statement of function-definition from 6.9.1 of [ISO-9899-2011]. In advice bodies one can use any correct C code with GCC compiler extensions that can be used in function bodies. In addition, advice bodies may contain *comments* and *special directives* which reflect information about joint points or have some special purpose. For example, special directive **$arg_numb** denotes the number of function parameters, **$fprintf** is intended for formatted output of data to a file, **$env** denotes a value of an environment variable.

Semantics

Advice bodies are not parsed except for *special directives* and *comments*. Special directives are substituted with corresponding values either during parsing of aspects (so-called special directives with predefined values) or at aspect weaving. Comments are ignored to correctly balance curly braces and determine ends of advice bodies. After parsing comments remain in advice bodies as is. This is necessary in order to keep, say, model comments.

1.3.10 Special directives

Syntax

```
special-directive                ::=  "$" aoc-identifier [aoc-integer-constant]
                                      "$" aoc-identifier [aoc-integer-constant] "<" special-directiv
special-directive-parameter-list ::=  special-directive-parameter
                                      special-directive-parameter-list "," special-directive-paramet
special-directive-parameter      ::=  special-directive
                                      aoc-integer-constant
```

[3] A pair of **$** characters is used to avoid collisions with *special directives*.

aoc-string-literal

Constraints

special-directive can be used only in *advice-body* and *file-name*. In order to avoid collisions with the C code used in advice bodies along with special directives, it is prohibited to use whitespace characters in special directives except for separating special directive parameters from each other. All special directives start with the **$** symbol which cannot be used in the C code.

`identifier` defines a type of special directive. The following types of special directives are supported: **$arg**, **$arg_numb**, **$arg_sign**, **$arg_size**, **$arg_type**, **$arg_val**, **$context_file**, **$context_func_file**, **$context_func_name**, **$env**, **$fprintf**, **$name**, **$proceed**, **$res**, **$ret_type**, **$storage_class**, **$signature** and **$this**. It is forbidden to use digits in `identifier` of *special-directive*. This is done to avoid collisions of identifiers with *aoc-integer-constant* that may be a part of special directives.

aoc-integer-constant of *special-directive* should be used only together with **$arg**, **$arg_sign**, **$arg_size**, **$arg_type** or **$arg_val**. These integer constants can only refer ordinal numbers of arguments of functions or macros from appropriate join points. Numbering begins with 1. You can not separate *aoc-integer-constant* from *aoc-identifier* as it was stated above.

special-directive-parameter-list should be used only along with **$env** and **$fprintf**. The only parameter allowed for **$env** is *aoc-string-literal*. This string literal should exactly match a name of one of environment variables. You can use any number of parameters for **$fprintf** but at least two parameters are mandatory. The first parameter should be either a string literal or a special directive with a predefined value which is also a string literal. This string literal should represent a file name (either relative or absolute path) that can be opened for writing[4]. The second parameter should be *aoc-string-literal*. This string literal represents simplified **format** defined in 7.21.6.1 of [ISO-9899-2011]. Only **%d** and **%s** specifiers are acceptable. They should match *aoc-integer-constant* and *aoc-string-literal* respectively among other parameters of special directives. Also, any of these parameters can be a special directive whose value is *aoc-integer-constant* or *aoc-string-literal*. Listing 1.10 contains an example of **$fprintf**.

Semantics

All special directives except **$fprintf** are replaced with some values: *integers*, *identifiers* without **$** wildcards or *string literals*.

Special directive **$fprintf** performs formatted data output to a specified file in the same way as standard C function *fprintf* described in 7.21.6.1 of [ISO-9899-2011].

Special directives **$env** and **$this** are the only special directives with predefined values. These values are determined at the stage of aspect parsing. Instead of **$env** a value of a corresponding environment variable is substituted. **$this** is identified with a name of a woven in C source file.

The remaining special directives are substituted at aspect weaving as follows:

- **$arg***i* – a name of ith formal parameter of a function or macro.
- **$arg_numb** – the number of parameters of a function or macro.
- **$arg_sign***i* – a signature of ith actual parameter of a function. An *argument signature* is an identifier based on a syntax tree of a corresponding argument. Argument signatures should be built in a way to distinguish arguments corresponding to different memory objects unambiguously though it is not always possible.
- **$arg_size***i* – an array size if ith actual parameter of a function is a pointer to a one-dimensional array or **-1** otherwise.

[4] This file is created if it does not exist.

- **$arg_type***i* – a type of i[th] formal parameter of a function. A corresponding type is provided by using *typedef*, so function pointers are also supported.
- **$arg_val***i* – a function name if i[th] actual parameter of a function is an address of some known function or **0** otherwise.
- **$context_file** – a path to a file containing a join point.
- **$context_func_file** – a path to a file that defines a function containing a join point.
- **$context_func_name** – a name of a function containing a join point.
- **$name** – a name of a macro, function, variable or composite type corresponding to a join point.
- **$proceed** – a join point itself, for example, an original function call.
- **$res** – a function return value (it is provided by a special variable).
- **$ret_type** – a type of function's return value or variable or a composite type (it is provided via *typedef*).
- **$storage_class** – a storage class of a function or global variable.

1.3.11 Location control directives

Syntax

```
location-control-directive  ::=  "#" aoc-integer-constant aoc-string-literal new-line
```

Constraints

The `new-line` nonterminal is defined in 5.2.1 of [ISO-9899-2011].

Location control directives (aka *line directives*) can be used outside of *advice bodies*. They should occupy exactly one line.

Semantics

The semantics of *`location-control-directive`* generally corresponds to the semantics of line control preprocessing directives described in 6.10.4 of [ISO-9899-2011]. In the *`location-control-directive`* definition *`aoc-integer-constant`* points out line numbers in files whose names are specified by *`aoc-string-literal`*.

`line directives` can arise at aspect preprocessing considered in *Introduction*. Users should unlikely use them.

1.3.12 Comments

Outside of `comment` the **//** symbols indicate the beginning of a one-line comment. The content of this comment is scanned only to detect the `new-line` character that ends it up and that is not included in the comment itself. Outside of `comment` the **/*** characters indicate the beginning of a multiline comment. The content of this comment is scanned only to detect the ***/** characters that end it.

On aspect preprocessing all comments always remain in the text of the resulting file with the aspect. This is done in order to keep, say, model comments. For a similar reason comments are kept within advice bodies at aspect parsing and aspect weaving.

1.3.13 Macros

Syntax

```
macro                          ::=   identifier
                                     identifier "(" [identifier-or-any-param-list] ")"
                                     identifier "(" [identifier] "..." ")"
                                     identifier "(" identifier-or-any-param-list "," [identifier] "..."
identifier-or-any-param-list   ::=   identifier
                                     ".."
                                     identifier-or-any-param-list "," identifier
```

Constraints

In comparison with preprocessor directives defined in 6.10 of [ISO-9899-2011], in AOC *macro* supports a GCC compiler extension that allows associating a name to "..." in the form of optional `identifier` before it. "..." designates a list of arbitrary macro parameters of arbitrary length. Also, *identifier-or-any-param-list* supports the ".." wildcard. It means a list of arbitrary macro parameters of arbitrary length.

Semantics

In general, the semantics of *macro* corresponds to the semantics of preprocessor directives described in 6.10 of [ISO-9899-2011]. Wildcard ".." matches a list of arbitrary macro parameters of arbitrary length at a joint point. For instance, **LOCK(x, ..)** will match both **LOCK(x)**, **LOCK(x, y)** and **LOCK(x, y, z)**, but it will not match **LOCK()** and **LOCK**. If there are several consecutive ".." separated by commas, they are treated as one "..".

1.3.14 Declarations of functions, variables, and types

Syntax

```
declaration                ::=   declaration-specifiers [declarator]
declaration-specifiers     ::=   storage-class-specifier [declaration-specifiers]
                                 type-specifier [declaration-specifiers]
                                 type-qualifier [declaration-specifiers]
                                 ".." [declaration-specifiers]
                                 "..."
storage-class-specifier    ::=   "typedef"
                                 "extern"
                                 "static"
                                 "auto"
                                 "register"
type-specifier             ::=   "void"
                                 "char"
                                 "short"
                                 "int"
                                 "long"
                                 "float"
                                 "double"
                                 "signed"
```

```
                                "unsigned"
                                "_Bool"
                                "_Complex"
                                struct-or-union-specifier
                                enum-specifier
                                typedef-name
                                "$"
struct-or-union-specifier   ::= struct-or-union identifier
struct-or-union             ::= "struct"
                                "union"
enum-specifier              ::= "enum" identifier
typedef-name                ::= identifier
type-qualifier              ::= "const"
                                "restrict"
                                "volatile"
function-specifier          ::= "inline"
declarator                  ::= [pointer] direct-declarator
direct-declarator           ::= identifier
                                "(" declarator ")"
                                direct-declarator "[" [integer-constant] "]"
                                direct-declarator "[" "$" "]"
                                direct-declarator "(" parameter-type-list ")"
pointer                     ::= "*" [type_qualifier_list]
                                "*" [type_qualifier_list] pointer
type_qualifier_list         ::= type-qualifier
                                type_qualifier_list type-qualifier
parameter-type-list         ::= parameter-list
parameter-list              ::= parameter-declaration
                                parameter-list "," parameter-declaration
parameter-declaration       ::= declaration-specifiers declarator
                                declaration-specifiers abstract-declaratoropt
abstract-declarator         ::= pointer
                                [pointer] direct-abstract-declarator
direct-abstract-declarator  ::= "(" abstract-declarator ")"
                                "[" direct-abstract-declarator "]" "[" [integer-constant] "]"
                                [direct-abstract-declarator] "[" "$" "]"
                                [direct-abstract-declarator] "(" [parameter-type-list] ")"
```

Constraints

In comparison with *declaration* that represents declarations of functions, variables, and types and that is defined in 6.7 of [ISO-9899-2011], AOC *declaration* have the following differences:

- It does not support `init-declarator-list`. Only *declarator* itself can be used instead.
- *struct-or-union-specifier* does not support specifying structure or union fields.
- *enum-specifier* does not support setting enumeration constants.
- The *direct-declarator* definition does not support:
  - Various forms of array assignment.
  - The outdated form of providing function parameters.
- *parameter-type-list* does not support "..." that designates a list of arbitrary function parameters of arbitrary length (it is supported at the level of *declaration-specifiers* which is discussed below).

- The *direct-abstract-declarator* definition does not support various forms of array assignment.
- *declaration-specifiers* additionally supports:
  - Wildcard ".." capturing a list of arbitrary function parameters of arbitrary length.
  - "..." that designates a list of arbitrary function parameters of arbitrary length. This works only for declarations from *parameter-list*.
- The *type-specifier* definition supports universal type specifier "$" in addition. One declaration can contain no more than one universal type specifier among all its specifiers. This restriction is important since exactly the same wildcard can be used in place of a declaration name. For a structure, union, or enumeration declaration a corresponding type specifier should be specified. This is necessary to distinguish declarations using two "$" symbols that match variables or functions. For example, **$ $** can correspond to variables such as *int var1*, *static long int var2* and *char var3[10]*, but it does not match *struct S*, *union U* and *enum E* types. For the latter you can use **struct $**, **union $** and **enum $** respectively.
- *direct-declarator* and *direct-abstract-declarator* supports universal array size "$".

Semantics

Declarations are distinguished in the following way. Absence of *declarator* in the *declaration* definition means that this declaration is a composite type declaration. If *declarator* is present then the declaration is either a function declaration (if there is *parameter-type-list*) or a variable.

Wildcard ".." in the definition of *declaration-specifiers* corresponds to a list of arbitrary function parameters of arbitrary length at a joint point. Several consecutive, separated by commas ".." are treated as one "..".

As a matter of fact "..." in *declaration-specifiers* exactly coincides with the same terminal in *parameter-type-list* (6.7.6 of [ISO-9899-2011]). The need to transfer it arose due to the ambiguity of the grammar otherwise.

Basically the semantics of *declaration* corresponds to the semantics of *declaration* described in 6.7 of [ISO-9899-2011].

Universal type specifier "$" in the definition of *type-specifier* means the following:

- If the universal type specifier is located before any other type specifier, then it denotes a list of arbitrary declaration specifiers of arbitrary length (the "$" symbol does not match arbitrary *typedef-name*). For instance, **$** matches **char**, **int**, **unsigned int**, **static inline int** and so on.
- If the universal type specifier is the only type specifier among declaration specifiers (according to the restriction specified earlier, it can be functions or variables only), then it denotes a type of variable or return value of a function, which is arbitrary up to the specified declaration specifiers. For instance, **$ int** matches **int**, **unsigned int** and **static inline int**, but it does not match, say, **char**.

Universal array size "$" in definitions of *direct-declarator* and *direct-abstract-declarator* corresponds to an arbitrary array size at a joint point. For example, **int array[$]** will match both **int array[3]** and **int array[5]**.

1.3.15 Pointcuts

Syntax

```
named-pointcut      ::=   "pointcut" identifier ":" pointcut
pointcut            ::=   identifier
                          primitive-pointcut
                          composite-pointcut
composite-pointcut  ::=   "!" pointcut
                          pointcut1 "||" pointcut2
                          pointcut1 "&&" pointcut2
                          "(" pointcut ")"
primitive-pointcut  ::=   "define" "(" macro ")"
                          "expand" "(" macro ")"
                          "declare_func" "(" declaration ")"
                          "execution" "(" declaration ")"
                          "call" "(" declaration ")"
                          "get" "(" declaration ")"
                          "get_global" "(" declaration ")"
                          "get_local" "(" declaration ")"
                          "infunc" "(" declaration ")"
                          "introduce" "(" declaration ")"
                          "set" "(" declaration ")"
                          "set_global" "(" declaration ")"
                          "set_local" "(" declaration ")"
                          "file" "(" file-name ")"
                          "infile" "(" file-name ")"
```

Constraints

It is forbidden to use "$" wildcards in `identifier` in the definition of *named-pointcut*. Preprocessed aspect files can not define several *named pointcuts* with the same `identifier`.

`identifier` can be only an identifier of a previously defined named pointcut in the definition of *pointcut*. It also can not use "$" wildcards.

Strictly speaking *pointcut1* and *pointcut2* represent different pointcuts in the definition of *composite-pointcut*.

The definition of *primitive-pointcut* has following constraints (you can find extra details about declarations in *Declarations of functions, variables, and types*):

- *declaration* for "declare_func", "execution" and "call" should be only a function declaration.
- *declaration* for "get", "get_global", "get_local", "set", "set_global" and "set_local" should be only a variable declaration.
- *declaration* for "introduce" should be only a declaration of a composite type.

Semantics

named-pointcut binds *pointcut* to `identifier` that one can use in other pointcuts to refer the given one.

composite-pointcut is a composition of pointcuts obtained using parentheses and operators "!", "&&" and "||". The precedence of operators "!", "&&" and "||" decreases left to right.

primitive-pointcut describes the following sets of joint points:

- "define" and "expand" – respectively a definition or substitution of *macro*.
- "declare_func", "execution" and "call" – correspondingly a declaration, definition, or call of a function having appropriate *declaration*.
- "get" and "set" – respectively a usage or assignment of a value to a variable with corresponding *declaration*.
- "get_global", "set_global", "get_local" and "set_local" – the same as the previous primitive pointcut, but global and local (including function parameters) variables are distinguished.
- "infunc" – join points in a context of a function with specified *declaration*.
- "introduce" – a definition of a structure, union, or enumeration with specified *declaration*.
- "file" – a file with *file-name*.
- "infile" – join points in a context of a file with *file-name*.

1.3.16 Advices

Syntax

```
advice              ::=   advice-declaration advice-body
advice-declaration  ::=   "before" ":" pointcut
                          "around" ":" pointcut
                          "after" ":" pointcut
                          "info" ":" pointcut
                          "new" ":" pointcut
                          "query" ":" pointcut
```

Note: "info" is a deprecated alias for "query". You can use any of them, but "query" is more preferable.

Note: It is not recommended to use "new".

Constraints

Each advice should consist of *advice-declaration* and *advice-body*. Any *pointcut* is allowed for *advice-declaration* with "before", "around", "after" and "query". Only *primitive-pointcut* corresponding to *file-name* is allowed for "new" *advice-declaration*.

In *advice-body* of "before", "around", "after", "new" and "query" one can use special directives "$env", "$fprintf" (if other special directives represent its parameters, then similar restrictions are imposed on them) and "$signature". Besides, in *advice-body* of "before", "around", "after" and "query" it is possible to use the following special directives when *pointcut* matches an appropriate joint point:

- For macro definitions – "$arg", "$arg_numb", "$context_file", "$name" and "$proceed".
- For macro substitutions – "$arg", "$arg_numb", "$arg_val" (a value of an actual macro parameter as is), "$context_file", "$name" and "$proceed".
- For function calls – "$arg", "$arg_numb", "$arg_sign", "$arg_size", "$arg_type", "$arg_val", "$context_file", "$context_func_file", "$context_func_name", "$name", "$proceed", "$res" (only for "after"), "$ret_type" and "$storage_class".
- For function declarations – "$arg_numb", "$arg_type", "$context_file", "$name", "$ret_type" and "$storage_class".
- For function definitions – "$arg", "$arg_numb", "$arg_type", "$context_file", "$name", "$proceed", "$res" (only for "after"), "$ret_type" and "$storage_class".
- For usages and assignments of values to local or global variables – "$context_file", "$context_func_file", "$context_func_name", "$name", "$proceed", "$res" (only for "after"), "$ret_type" (a matched variable type) and "$storage_class" (only for global variables).
- For declarations of composite types – "$context_file", "$name" and "$ret_type" (a matched composite type).

Semantics

pointcut included in *advice-declaration* determines a set of join points for which this advice should be applied, that assumes either executing the code from *advice-body* or framing join points with it.

"before", "after" and "around" advices are applied before, after or instead matched join points respectively. "around" advices can also wrap corresponding join points indicated by the "$proceed" special directive in *advice-body*.

"query" advices do not change the program code. These advices are used only for formatted output of information about joint points to a file by means of special directives "$fprintf".

The "new" advice creates a file that is specified in "pointcut". This feature allows, for example, to declare common variables and functions for several C source files.

In *advice-body* it is allowed to write arbitrary correct C code with GCC compiler extensions as well as a set of special directives (*Special directives*). You can use only special directives "$fprintf" in bodies of "query" advices (parameters of this special directive may be other valid special directives).

If parameter names are used in *parameter-type-list*, then you can use them to refer corresponding parameters in *advice-body*.

If several advices match the same join point, then only the one that occurs earlier in the aspect file is applied. For more complex cases, for example, when a program is woven in with several aspects at once, the behavior of the aspect weaver is uncertain.

1.3.17 Aspects

Syntax

```
text                          ::=   [advice-or-named-pointcut-list]
advice-or-named-pointcut-list ::=   advice-or-named-pointcut-list advice
                                    advice-or-named-pointcut-list named-pointcut
```

Constraints

Aspects should be placed in separate files. After performing aspect preprocessing (see *Introduction* for details), each aspect can either be empty or consist of one or more *advices* and *named pointcuts*. In addition, *line directives* and *comments* can be used.

Semantics

Aspects are additional modules that describe the cross-cutting concerns of programs.

1.4 Development

1.4.1 Building debug version of Aspectator

To build a debug version of Aspectator you can either run the following command:

```
$ make -j16 debug
```

or make the appropriate actions by hand. The first action is to create a separate directory for it, say:

```
$ mkdir build-debug
$ cd build-debug
```

Then you need to configure Aspectator:

```
$ MAKEINFO=missing ../aspectator/configure --enable-languages=c --disable-multilib --
↪disable-nls --enable-checking=release
```

and make its debug version:

```
$ make STAGE1_CXXFLAGS="-g -O0" all-stage1
```

You can use option *-jN* for *make* to essentially speed up building, but it can cause failures (just invoke the command several times to overcome this):

```
$ make -j16 STAGE1_CXXFLAGS="-g -O0" all-stage1
```

After making some changes to files starting with *ldv-* prefix it is strongly recommended to rebuild the debug version of Aspectator with *-Werror* flag to treat all warnings as errors:

```
$ make STAGE1_CXXFLAGS="-g -O0 -Werror" all-stage1
```

To debug Aspectator you can use *gdb* or *ddd*:

```
$ ddd gcc/cc1 &
```

To debug instrumentation you need to set the following environment variables:

```
set env LDV_STAGE=3
set env LDV_ASPECT_FILE=$ABS_PATH_TO_ASPECT_FILE
set env LDV_OUT=out.c
```

To debug C back-end you need to set the following environment variables:

```
set env LDV_STAGE=4
set env LDV_C_BACKEND_OUT=out.c
```

Note: These instructions were adapted from http://gcc.gnu.org/wiki/DebuggingGCC.

1.4.2 Profiling Aspectator

Sometimes developers need to track whether some memory issues (e.g. memory leaks, use after free, etc.) were introduced and to measure algorithms complexity. First of all you need to build a debug version of Aspectator (*Building debug version of Aspectator*) and install extra tools such as *valgrind*, *valkyrie* and *kcachegrind*.

Tracking memory issues of Aspectator

To track memory issues you need to run Aspectator under *valgrind* (do not specify *–suppressions* if you do not have them):

```
LDV_ASPECT_FILE=$PATH_TO_ASPECT_FILE \
LDV_STAGE=$STAGE \
LDV_OUT=$PATH_TO_OUT \
valgrind \
--tool=memcheck \
--leak-check=yes \
--suppressions=gcc.supp \
--num-callers=500 \
--xml=yes \
--xml-file=output.xml \
$PATH_TO_ASPECTATOR_BUILD_DEBUG/gcc/cc1 \
$PATH_TO_INPUT_FILE
```

After that you can either inspect *output.xml* manually or use *valkyrie*:

```
$ valkyrie -l output.xml
```

Tracking CPU time issues of Aspectator

To measure CPU time consumption you need to run Aspectator under *valgrind*:

```
LDV_ASPECT_FILE=$PATH_TO_ASPECT_FILE \
LDV_STAGE=$STAGE \
LDV_OUT=$PATH_TO_OUT \
valgrind \
--tool=callgrind \
$PATH_TO_ASPECTATOR_PROFILED_DEBUG/gcc/cc1 \
$PATH_TO_INPUT_FILE
```

After that you can either inspect files *callgrind.out.** manually or use some tool, e.g. *kcachegrind*:

```
$ kcachegrind -l callgrind.out.*
```

BIBLIOGRAPHY

[ISO-9899-2011] ISO/IEC 9899:2011 Information technology – Programming languages – C

INDEX

E

L