One Approach to Aspect-Oriented Programming Implementation for the C programming language

Eugene Novikov
Institute for System Programming
Russian Academy of Sciences
Moscow, Russian Federation
Email: joker@ispras.ru

***Abstract*—The given paper introduces an approach for aspect-oriented programming implementation developing intended for the C programming language. Key features of C and a common C program build process are considered and it's shown how they influence on a supposed C AOP implementation. The last is described in details and after all its practical application is demonstrated. It's shown that the supposed C AOP implementation works well enough although it possesses some shortcomings. Some improvements required to overcome the given shortcomings are discussed.**

Keywords-aspect-oriented programming; join point; pointcut; advice; aspect; weaving; the C programming language; implementation

I. Introduction

Aspect-Oriented Programming (AOP) is a rather fresh programming paradigm that is intended to increase program modularity by means of cross-cutting concerns separation. Generally speaking cross-cutting concerns mean functionality or features that cannot be easily decomposed from so-called core concerns. The last in depend on a programming paradigm used is implemented as corresponding functions, classes and modules while cross-cutting concerns scatter through them and tangle a program source code. The typical example of cross-cutting concerns is logging. Also some more complex fields like errors handling, some sort of testing, security, and database operations can be treated as cross-cutting concerns. AOP provides programmers with opportunity to extract cross-cutting concerns into separate modules called aspects. To understand better let us consider other major AOP conceptions since they'll be widely used at the rest of the given paper.

The basic AOP conception is a **join point**. In general join points are those elements of the programming language semantics which the aspects coordinate with [1]. The given paper takes a join point to be a program construction connected with its context. The typical example of a join point is a function/method call because of such a construction can be found almost in any programming language. But generally speaking join points depend on and even in some degree are determined by a programming language used. A **pointcut** is a set of join points satisfying a given condition. For instance, all memory allocating function (like *malloc*, *calloc* and so on) calls may be treated as a pointcut. Next AOP conception is an **advice**. An advice consists of a pointcut and a body. The last represents some actions to be executed in matching between a join point corresponding to a given pointcut and a program construction related with some context. Moreover an advice contains information on whether these actions should be executed **before**, instead of (**around**) or **after** a matched program point execution. Usually an advice body is written in a given programming language although some special AOP constructions (e.g. a matched entity name) may be also available. An **aspect** already mentioned above is a separate module that consists of a number of advices implementing some part of cross-cutting concerns. More exactly an aspect also can contain some other constructions, e.g. named pointcuts that is pointcuts associated with identifiers for following usage. At last, the process of aspects with main program integration is referred to as **weaving**. Weaving can be done at any stage of a program processing (at compile time, at post compile time, up to run time) that is exhibited by different approaches.

An AOP implementation depends on a programming language used as was said. Generally an AOP implementation represents a programming language superset required to write aspects and some tool(s) to weave aspects with programs. Let us consider the most advance and popular AOP implementation AspectJ [2] intended for the Java programming language. Note that even though the goal of this paper is the C programming language, nevertheless AspectJ is well suited because of C and Java programming languages have many similar constructions and almost all AOP implementations are more or less based on AspectJ ideas.

```
// An aspect consisting of a named pointcut
// and an advice.
aspect Logging {
  // A named pointcut that matches a join
  // points set of method calls.
  pointcut move():
    call(void FigureElement.setXY(int,int)) ||
    call(void Point.setX(int))              ||
    call(void Point.setY(int));
  // An advice performing some actions before
  // execution of a matched by the given named
  // pointcut program point.
  before(): move() {
    System.out.println("about to move");
  }
}
```

Figure 1. Example of an AspectJ aspect for a graphical system logging

In using an AspectJ extension for Java a logging functionality for a graphical system can be extracted into an aspect showed in Fig. 1 [3]. In whole this means that before execution of each called method from the specified ones the given log message will be printed to a screen. The AspectJ weaver deals with Java program bytecode and after its work such the object code is obtained. This weaver is implemented as a part of a special compiler. The given example shows that an AOP implementation really strongly depends on a programming language and a program build process. Indeed there are more than 20 different AOP implementations just for the Java programming language. So the main goal, to separate cross-cutting concerns from the core ones for a given programming language, can be reached in the different ways.

The rest of the paper is structured as follows. Section II considers features of the C programming language and a typical build process of programs written in the given language. On the basis of these features and demands of the Linux driver verification project (it's considered there) requirements to an AOP implementation for the C programming language are collected. Section III describes related work and shows how different approaches meet the requirements pointed out in Section II. A suggested approach of an AOP implementation for the C programming language is introduced in Section IV. Section V estimates an application of the suggested approach. Section VI summarizes the work done and considers future work directions.

II. Requirements to an AOP implementation for the C programming language

Let us consider a typical workflow in building of a common C program and estimate how AOP conceptions may be related with different C constructions. It's worth while noticing that during the given consideration we won't restrict a C AOP implementation to represent just AOP constructions similar to the AspectJ ones as it's done by the most of AOP implementations. On the contrary we will try to describe an AOP implementation specific for the C programming language. Generally speaking it's assumed that such the implementation won't have any fundamental limitations for C cross-cutting concerns separation.

Fig. 2 illustrates 3 stages of a common C program build process, *preprocessing*, *compilation* and *linking*. Note that rectangles having dash line borders represent third-party components used by a program considered.

A. Preprocessing

At the first stage a preprocessor in depend on passed preprocessor options includes necessary header files (both program's *h1.h*, *h2.h*, *h3.h*, … and libraries' *lib1.h*, *lib2.h*, *lib3.h*, …) into a program source code files *a1.c*, *a2.c*, *a3.c*, … and expand macros there. More exactly it's the two main actions performed by a preprocessor but the rest ones aren't touched in the given paper. Both header files including and macro expansion may be related with AOP conceptions in the following way. Each file included to a given program source code file and even that source code file itself can be treated as a corresponding join point. Therefore the including process can be modified by adding needed instructions before, after or instead of a given file. For instance, this helps to add some auxiliary preprocessor directives, function prototypes and so on. Macro expansion also can be altered in the similar way. So instead of (or before, or after) a substituted code we may put our own code that may deal with macro arguments as well as perform some required actions. Preprocessing is the essential C feature because of there is just few programming languages through all that supports it.

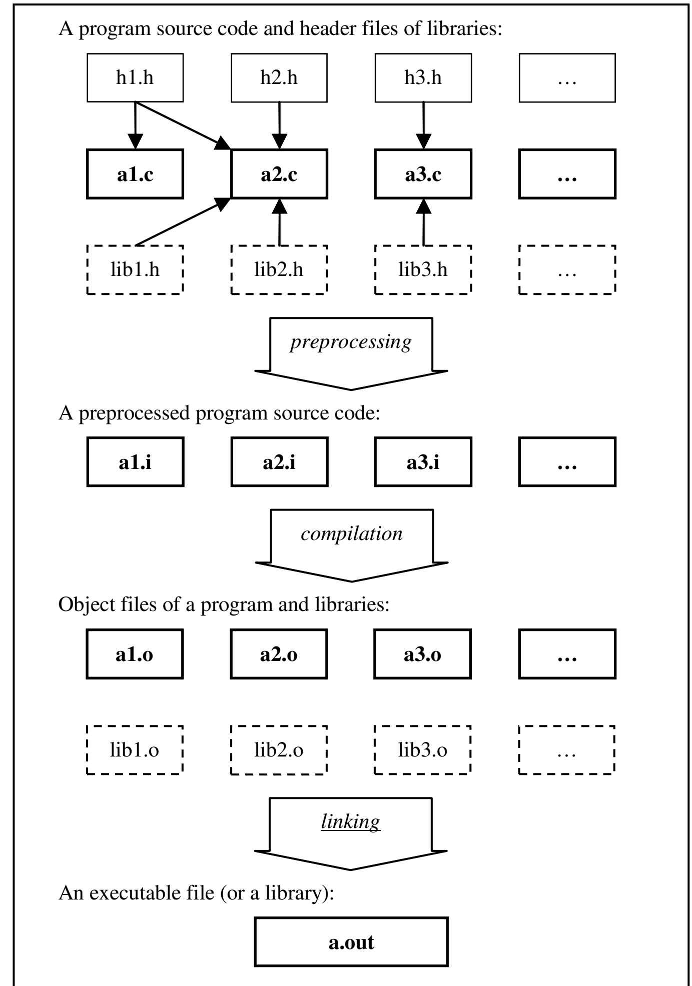

Figure 2. Common C program build process

B. Compilation

Then at the second stage a compiler parses a preprocessed program source code files *a1.i*, *a2.i*, *a3.i*, … and produces corresponding object files *a1.o*, *a2.o*, *a3.o*, … Compilation may be affected by some build options. Traditionally AOP conceptions are developed for constructions of the given stage. For instance, AOP conceptions are related with function/method definitions and calls, type/class declarations and variable and field manipulations. Here is indeed the large area for AOP to be involved. As for the given work it's suggested that there should be implemented at least support for such join points as a function definition and call, a type declaration, a local and global variable, field and function

parameter set and get. The most of current AOP implementations support just the given or even a smaller set of join points (see Section III). Moreover C is a programming language having pointers and a lot of operations with them. The most popular operations like a pointer dereference and vice versa a taking of a variable/field/parameter address and a taking of an array element should be supported as corresponding join points. Also it's required that for each mentioned join point some actions written as advice bodies can be performed before, instead of and after a corresponding program construction execution. Of course it isn't a complete list of different program join points, e.g. loop and specific condition statements as well as a lot of different expressions weren't described. But indeed they also may be taken into account sooner or later.

Both the first and the second stages weaving should produce a correct source code or/and a corresponding compiler internal representation. For example, advice bodies should be substituted and be compliable as well as a given program source code. Also for these stages a considered construction scope (either some file or some function) plays a significant role because of some action like a function call may be performed either in one file or in another one, in one function or in another one. It is important to notice that among advice body instructions there may be some specific AOP instructions. In the given paper they are referred to as **body patterns**. For instance, there may be such body patterns as:

- a matched construction name and type (for a function call and definition, for a variable declaration, etc.);
- matched construction argument names and types (for a function call and definition);
- a matched construction size (for entities having an array type or strings);
- and even a matched construction itself (e.g. to have ability to call a matched function from inside a corresponding advice body).

C. Linking

Linking performed at the third stage by a linker with corresponding build options assembles given program object files *a1.o*, *a2.o*, *a3.o*, … together with libraries object files *lib1.o*, *lib2.o*, *lib3.o*, …. After all an executable file or a library *a1.out* is obtained. It's worth while mentioning that C program object files to be linked shouldn't contain the same defined symbols such as function definitions having the same names. So if some shared functions and global variables are required to separate cross-cutting concerns they should be contained just in one object file. For instance, this may help to use different counters or flags, i.e. to save a shared context or state, and to efficiently execute the same code by means of special auxiliary functions (one can see an example in Section V). Interaction of AOP directly with object files and a running program is beyond of the given paper.

So the common C program build process, the most of key C constructions and their influence on an AOP implementation were considered. But the goal of the given paper isn't to introduce some AOP implementation for the C programming language but is to suppose the one that can be used for real programs. To the author's knowledge unfortunately the most of C AOP implementations are used just for artificial simple examples and isn't widely used in practice (there is some discussion about the given issue in Section III). So the AOP implementation concerned at this paper was strongly affected by the Linux driver verification (LDV) project [4][5]. The goal of that project is to provide an industrial quality toolset that allows to use different static code analysis tools to verify whether drivers satisfy correctness rules or not. The appropriate way to formalize these correctness rules in the manner being independent on a static verifier used and than to instrument a driver source code to be checked is to use AOP. Therefore this constrains some extra circumstances on a C AOP implementation:

- Support of the C programming language with all GNU extensions as an input language (it's a standard language for drivers writing) as well as all support of standard and GNU build options.
- Offering of a well set of AOP constructions corresponding to the C programming language. This is required since correctness rules refer to different C constructions used in different contexts. But nevertheless aspects development should be rather easy.
- An output should be also a correct program in C equivalent to the original one except it may be extended with corresponding cross-cutting concerns. This is required by the following application of static code analysis tools.
- An AOP implementation should be quite easy maintained and extended with new features. This comes because of new correctness rules are constantly appearing, so an extra AOP constructions support is required.

Note that nevertheless the most of these requirements are suitable for any program written in the C programming language (may be with allowance that this is done on the Linux platform). So a supposed C AOP implementation appears to be used both in the LDV project and in developing of a rather random C program. Moreover the requirement for an output to be a C program is useful for an AOP implementation debugging, because of by means of this output one can easily observe how a given AOP implementation behaves.

III. RELATED WORK

AOP for the C programming language that is the goal of the given paper is considerably less developed in comparison with the one for Java. At present the most interesting C AOP implementation is ACC (AspeCt-oriented C) [6]. Fig. 3 shows that its superset for C likes the one for Java made in AspectJ [7]. That aspect means that after function *foo2* is called its result will be printed to a screen. ACC weaving differs from the one of AspectJ. For a given preprocessed C file ACC produces a corresponding C file extended with cross-cutting concerns. Despite of ACC supports a rather large set of AOP

constructions it cannot deal with preprocessor ones since it takes already preprocessed source code. Also it is intended just for one file processing and there isn't ability to specify some shared variables and auxiliary functions. ACC has its own closed C parser that fails to process some GNU extensions. Maintenance of ACC by its developers isn't active and due to its core component is closed it isn't so easy to deal with it.

```
  // An advice printing a message after a
  // given function call is performed.
  after (int res): call(int foo2(int)) &&
result(res) {
    printf(" after call foo2, return %d\n", res);
  }
```

Figure 3. Example of an ACC aspect

InterAspect is a more recent AOP implementation intended for the C programming language [8]. It was developed almost at that time when the given work was done. This tool is interesting because of it's based on GCC plugins [9], so it is most likely to support all GNU extensions. Unfortunately the InterAspect tool after all produces an object code (in fact this is done by GCC itself) like AspectJ so it cannot be directly used for static verification. At present the given tool supports rather limited number of AOP constructions and preprocessor constructions aren't supported as well as state variables and auxiliary functions. Instead of a C superset it provides a special C AOP library allowing to write aspects like an usual C program. But as one can see in Fig. 4 it seems to be even a more complex task to write such an aspect. In fact there only a joint point for *malloc* function call is defined. The tool was actively developed recently. However its development was stopped at the end of 2010. Nevertheless its progress should be tracked and correlated with the suggested approach.

```
  static void instrument_malloc_calls() {
    /* Construct a pointcut that matches calls
to: void *malloc(unsigned int). */
    struct aop_pointcut *pc =
aop_match_function_call();
    aop_filter_call_pc_by_name(pc, "malloc");
    aop_filter_call_pc_by_param_type(pc, 0,
aop_t_all_unsigned());
    aop_filter_call_pc_by_return_type(pc,
aop_t_all_pointer());
    /* Visit every statement in the pointcut. */
    aop_join_on(pc, malloc_callback, NULL);
  }
```

Figure 4. Example of a part of an InterAspect aspect

Another good approach is SLIC (Specification Language for Interface Checking (of C)) [10]. To the author's knowledge it's the only C AOP implementation that is widely used in practice. However it has just one field of application, it's used during a process of static verification of Microsoft Windows operation system drivers. SLIC allows to use state variables and has a simple syntax for aspect writing. A SLIC specification is indeed some kind of an aspect. The example of a SLIC specification is demonstrated in Fig. 5. This artificial specification states that it is an error to have more than four zeroes in a queue. A SLIC preprocessor weaves driver source code with a specification and after all produces equivalent C program to be checked by means of a static verifier. A shortcoming of the given approach is that there just few join points are implemented (in fact just a function call and definition). Also the given project is completely closed.

```
  state { int zero_cnt = 0; }
  put.entry {
    if ($1 == 0) {
      if (zero_cnt == 4)
        abort "Queue has 4 zeroes!";
      else
        zero_cnt = zero_cnt + 1;
    }
  }
  get.exit {
    if ($return == 0)
      zero_cnt = zero_cnt - 1;
  }
```

Figure 5. Example of a SLIC specification

A lot of other AOP implementations for C like C4, Aspicere2, Xweaver project, WeaveC and so on posses a less number of useful features than the ones described above, so they aren't considered in this paper. Also AOP tools dealing with C++ even though they may be adapted in some way for the C programming language aren't introduced because of usually they produce output in C++ while C is required by static code analysis tools.

IV. Overview of suggested C AOP implementation architecture

A suggested approach tends to implement all the requirements described in Section II in the most complete way. So after thorough investigation it was decided to base it on the LLVM compiler infrastructure [11]. In turn this infrastructure is built on top of GCC, it has so-called LLVM GCC Front End binding GCC with LLVM tools. So the LLVM compiler infrastructure inherits a GCC parsing of both C constructions and GNU extensions and supports all GCC build options almost as InterAspect described above. The suggested C AOP implementation is built on top of a GCC parser itself. Because of GCC includes preprocessing the given C AOP implementation can deal with both preprocessing and compilation join points. Next the LLVM tools include its own linker and a C backend tool. The first allows to link several object files of the whole program together, so some set of source code files can be woven instead of an alone file. The C backend tool is used to produce a C source code file to be verified by a static code analysis tool. To write aspect files it was decided to use a superset of C like AspectJ, ACC and SLIC do. Section V contains an example of such an aspect that is used in practice. Below the overall architecture of the suggested C AOP implementation is considered in more details. It's shown how program source code files, libraries' header files and aspect file are used and modified to weave cross-cutting concerns with a program.

Different constructions matching and weaving are performed through 4 stages by means of LDV GCC Front End invocation on each stage. Then linking and a C source code file generation are done. First of all it's necessary to mention that

there are usually 2 aspect files. The first is intended for weaving with all program source code files. The second aspect file is required to define auxiliary function definitions and global variable declarations shared between all other source code files. The second aspect file is applied just to one program source code file of those forming a final executable file or a library. To make the further description more general * is used instead of corresponding names. For instance a first aspect file is denoted as **.aspect*, and a second as **.aspect.common*.

A. Aspect preprocessing

At the first stage comments of both C and C++ styles are eliminated from both aspect files. So **.aspect.nc* and **.aspect.common.nc* (where *nc* means "*no comment*") are obtained. Then at every stage such the modified aspect files are parsed by means of a special parser (that is later referred to as *aspect parser*) implemented as a patch for LLVM GCC Front End. In an aspect file parsing lexical, syntax and semantic correctness is checked. Advice bodies are looked through just to determine body patterns. In case of some error an exact place and an error type are reported. If a given aspect file is correct it's translated into own internal representation used during matching and weaving later.

At the first stage required modifications are done for a program source code file processed, **.c*. Either before or after or instead of it some additional source code is inserted. This is done to process further these modifications as soon as possible, i.e. even by means of a preprocessor because of they may contain some preprocessor directives. By analogy with a preprocessor a file obtained after this stage is called **.c.p* (*p* means "*preprocessed*") and the given stage is named *aspect preprocessing*. At the moment there isn't weaving for included files but it can be implemented in the similar way.

B. Macro weaving

At the second stage during the standard preprocessing of a **.c.p* file performed by LLVM GCC Front End using corresponding build options (e.g. to find all included files) macro matching and weaving are performed. So this stage is referred to as *macro weaving*. When a corresponding to a given pointcut macro directive is matched a macro body is extended in a way required by an advice. After all there is a **.c.p.mw* (*mw* means "*macro woven*") file that is the both aspect preprocessed and preprocessed one.

C. Advice weaving

The third and the fourth stages correspond to the compilation phase. Here is important to notice that we don't restrict an advice body source code with C constructions usage and we don't parse it by ourselves. Instead, advice bodies are substituted to a given source code file as unique auxiliary function bodies on advice pointcut matching. And then the LLVM GCC Front End powerful parser processes them. So at the third stage auxiliary functions required to implement advice body actions are created in depend on join points matching and advice requirements. Also to perform parsing of type declaration extensions as well as to allow using of given extensions in auxiliary functions type declarations weaving is done at the third stage. At this stage the LLVM GCC Front End C parser deals with a preprocessed file **.c.p.mw* and produces step by step its intermediate representation in the form of the GCC internal representation, called later as a *parsing tree*. Also parsed entities (in fact, type declarations and function bodies) are looked through to find matches with pointcuts defined in a given aspect file. It's kept where matched entities are placed (to insert either auxiliary function prototypes or to extend corresponding type declarations later), what exact types and names are matched to replace body patterns used in corresponding advice bodies. After all required type declaration extensions as well as auxiliary function definitions with substituted body patterns and their prototypes are directly inserted into corresponding places of an initial source code file **.c.p.mw* and a **.c.p.mw.aw* (*aw* means "*advice woven*") file is obtained. The stage is called *advice weaving*.

D. Compilation

After that at the fourth final stage the inserted source code is checked for correctness and translated into a parsing tree as well as an initial source code. Also at the third stage matching and weaving are performed in parsing. Here function definitions and function body expressions are modified directly at the level of the parsing tree and some relations with auxiliary functions are established if it's necessary. After the parsing is completed LLVM GCC Front End behaves in its standard mode and obtains an object file as well as a compiler does.

All four stages described above are summarized in Table 1. The table shows how input data is modified and used and what output is obtained in depend on a given stage.

TABLE I. DATAFLOW OF MATCHING AND WEAVING STAGES

Stage	*.aspect	*.c	Build options
Aspect preprocessing	Comments elimination (**.aspect.nc*) and parsing	Include join point weaving (**.c.p*)	Aren't used
Macro weaving	Parsing	Macro weaving and preprocessing (**.c.p.mw*)	Preprocessor options are used
Advice weaving	Parsing	Auxiliary functions and declarations direct including (**.c.p.mw.aw*)	Compiler options are used
Compilation	Parsing	Function definitions and bodies weaving, compilation	Compiler options are used

E. Linking and C source code file generation

Further required object files are linked together by means of the LLVM linker tool. As it was already mentioned for a resultant file just one object file woven with both aspect files is taken. For an assembled object file the LLVM C backend tool produces a C source code file that can be processed by a static verifier. Although the last action is optional. For example, instead of this there is ability to produce an executable file for a given program that is intended for some architecture supported by the LLVM compiler infrastructure.

V. APPLICATION OF SUGGESTED C AOP IMPLEMENTATION

The suggested AOP implementation for the C programming language is already included into a LDV project toolset. It's

used to formalize few correctness rules and in driver source code instrumentation intended for a further verification by means of static code analysis tools.

Fig. 6 shows an example of aspect files used in verification of the "Locking a mutex twice or unlocking without prior locking" correctness rule. Note that these aspect files are simplified in comparison with the actually used ones since some extra lock functions aren't presented. Syntax is most likely to be rather intuitively clear. It's worth while noticing that there are 2 join points as for macro *mutex_lock* and for function *mutex_lock*. This is required because of Linux kernel can define either a macro or a function in depend on its configuration. Function *mutex_unlock* is always declared as extern, so it doesn't require instrumentation since it can be explicitly defined. Global variable *ldv_mutex* is an example of shared state variables while *ldv_mutex_lock* is an auxiliary shared function. Function *ldv_check_final_state* is executed at the end of checking to ensure that nothing is locked then.

```
model0032a-blast.aspect
  before: file ("$this") {
  #include <linux/kernel.h>
  #include <linux/mutex.h>
  extern void ldv_mutex_lock(struct mutex *lock);
  }
  around: define(mutex_lock(lock)) {
  ldv_mutex_lock(lock)
  }
  before: call(extern void mutex_lock(struct
mutex *)) {
  ldv_mutex_lock($arg1);
  }

model0032a-blast.aspect.common
  after: file ("$this") {
  #include <linux/kernel.h>
  #include <linux/mutex.h>
  #include "engine-blast.h"
  int ldv_mutex = 1;
  void ldv_mutex_lock(struct mutex *lock) {
    ldv_assert(ldv_mutex == 1);
    ldv_mutex = 2;
  }
  void mutex_unlock(struct mutex *lock) {
    ldv_assert(ldv_mutex == 2);
    ldv_mutex = 1;
  }
  void ldv_check_final_state(void) {
    ldv_assert(ldv_mutex == 1);
  }
  }
```

Figure 6. Example of aspect files of the supposed C AOP implementation

To estimate quality of the proposed C AOP implementation 2 experiments were performed. The first one used a specially prepared Linux kernel configuration and corresponding kernel function implementations (like *mutex_unlock* showed in Fig. 6) while the second one used aspect files like presented in Fig. 6 and following instrumentation. Later the first experiment is called *plain* and the second one is called *aspect*. During experiments all drivers of Linux kernel 2.6.31.6 [12] that can be represented as kernel modules (there are 2160 such the drivers) were examined against the correctness rule about mutex lock/unlock described above with help of BLAST static code analysis tool [13]. The most interesting results demonstrating verdict changes between plain and aspect approaches are shown in Table II (a first verdict belongs to the plain experiment, and the second one belongs to the aspect one). *Safe* verdict means that a given driver satisfied the given correctness rule, *unsafe* is the reverse one, *unknown* verdict means that a static verifier used failed to check a given driver (e.g. because of time or memory shortage or due to some parsing error).

TABLE II. COMPARISON OF THE SUGGESTED C AOP IMPLEMENTATION WITH ANOTHER APPROACH

Safe → Unsafe	**Safe → Unknown**	**Unsafe → Unknown**	**Unknown → Safe**	**Unknown → Unsafe**
4	95	18	82	3

As one can see from Table II the supposed C AOP implementation behaves rather well because of the number of "bad" transitions (i.e. from safe/unsafe to unknown) almost equals to the number of "good" transitions. There are 95 + 18 = 113 "bad" transitions and 82 + 3 = 85 "good" ones. Their difference is just 28, that is less then 1.3% of the total number of kernel modules.

In fact it requires more memory for a generated file verification to be performed in the aspect experiment in comparison with the plain one. So, 62 modules were not checked because of memory shortage. Also in the aspect experiment some produced by LLVM C backend C constructions are rather complex for the static verifier used (31 modules were not checked due to the given reason). Although the plain experiment showed that even more drivers confuse a BLAST C parser because of complex constructions coming from initial driver source code as is. There are 68 such modules. The rest transitions from/to unknown verdict are concerned with either some bugs in the supposed C AOP implementation (20 modules for the aspect experiment) or time/memory shortage in the plain experiment (17 modules). Unfortunately, all additionally found unsafes (7 modules for which safe or unknown verdict was exchanged with unsafe one) are false positives because of either generated C file shortcomings (like generation of big unsigned integer numbers instead of negative ones that is demonstrated later) or incomplete correctness rule implementation and some static verifier lacks.

But nevertheless the most significant shortcoming of the supposed C AOP implementation consists in a generated code itself. Fig. 7 illustrates an example of how a driver source code is modified after the given implementation invocation. As it was already mentioned sometimes this prevent a static verifier from check performing due to complex constructions generated. As Fig. 7 shows there is a lot of variables having prefix *blast_must*. This is a special workaround made as a corresponding LLVM C backend patch. It is required to designate so-called *must-aliases*, that is the aliases that alias only one known memory location (all artificial temporary variables are must-aliases). The suggested approach application leads to more memory requirement for a testing to be executed. Such the generated source code scares users trying to see on it, for example, in analyzing unsafes or in debugging the given C AOP implementation. In fact the LLVM compiler

infrastructure used is responsible for this shortcoming. First of all it deals with a GCC internal representation called GIMPLE that already rather differs from a source code pure representation. Next it is intended for machine independent source code generation. So one can see large positive numbers instead of small negative ones in Fig. 7.

```
drivers/pci/hotplug/fakephp.c (preprocessed)
   if (strict_strtoul(buf, 0, &val) < 0)
     return -22;
   if (val)
     pci_rescan_bus(slot->dev->bus);

fakephp.ko.linked.cbe.c
  blast_must_tmp__85 = *(&llvm_cbe_buf_addr);
  blast_must_tmp__86 =
strict_strtoul(blast_must_tmp__85, 0u,
(&llvm_cbe_val));
  if ((((signed int )blast_must_tmp__86) <
((signed int )0u)))
    goto llvm_cbe_bb;
  else
    goto llvm_cbe_bb1;
  llvm_cbe_bb:
    *(&llvm_cbe_tmp__73) =
18446744073709551594ull;
    goto llvm_cbe_bb5;
  llvm_cbe_bb1:
    blast_must_tmp__87 = *(&llvm_cbe_val);
    blast_must_tmp__88 = *(&llvm_cbe_slot);
    blast_must_tmp__89 = *((&blast_must_tmp__88-
>field1));
  if ((blast_must_tmp__87 != 0ull))
    goto llvm_cbe_bb2;
  else
    goto llvm_cbe_bb3;
  llvm_cbe_bb2:
    blast_must_tmp__90 = *((&blast_must_tmp__89-
>field1));
    blast_must_tmp__91 =
pci_rescan_bus(blast_must_tmp__90);
  llvm_cbe_bb3:
```

Figure 7. Comarison of a driver source code with the generated one

Another big shortcoming is connected with the fact that LLVM GCC Front End is based on the rather old GCC compiler (of 4.2.1 version, nowadays 4.5.2 is a stable release) while the modern Linux kernel drivers already posses such new constructions that aren't processed with it. So different workarounds are required to overcome this.

After all let us imagine how different approaches introduced in Section III could meet aspect files presented in Fig. 6, driver source code instrumentation, following static analysis and obtained verification results examination. First of all none of them supports the join point concerned with the preprocessor construction `define(mutex_lock(lock))`. Then, step by step, ACC fails to parse driver source code because of unsupported fresh GNU extensions to the C programming language and that tool cannot be adjusted because of it uses a closed parser. InterAspect deals with GIMPLE representation of source code and, if we had some C backend tool for GCC, InterAspect would produce instrumented source code too dissimilar to the original one almost as well as LLVM C backend. Both ACC and InterAspect doesn't support state variables and functions like `ldv_mutex` and `ldv_mutex_lock` correspondingly. Most likely that we could verify the given model by means of SLIC, except the preprocessor issue, but in fact this is one of the simplest model from the LDV project. Other models require more complex join points and advice bodies, so what can we do if SLIC supports just function calls and definitions and it is the closed project.

VI. CONCLUSION

This paper describes an approach of how to implement aspect-oriented programming in the way specific for the C programming language. It considers features and shortcomings of current implementations. After all a new implementation that tends to cover all major features of the C programming language as well as to take into account those features that come from the C programs build process is considered. It's shown how the given C AOP implementation behaves to reach the required intention.

For the supposed C AOP implementation its real application for the Linux driver verification process is demonstrated. An example of real aspect files implementing a correctness rule associated with the mutex lock/unlock problem is given. Also the supposed approach is compared with another one that doesn't use AOP. It's shown that the given C AOP implementation is rather good except a generated source code is too complex for further analysis and it's quite unlike the original one. Mental comparison with another AOP approaches, such as ACC, InterAspect and SLIC, is done. Finally it becomes clear that the given approaches can not meet all requirements imposed on the suggested C AOP implementation by a number of reasons.

The current development of the supposed approach of the AOP implementation for the C programming language tends to overcome the restrictions specified above. To keep all advantages of the supposed approach as well as to eliminate the given shortcomings it was decided to develop our own C backend tool intended directly for GCC itself. It's assumed that it'll be built on top of stable GCC "from svn" that is it'll parse all modern constructions and GNU language extensions. Also the given C backend tool should work at the low-level GCC internal representation even before GIMPLE. Thus far a produced source code will most likely to be very similar to the original one. We believe that this will allow to combine abilities of both the supposed C AOP implementation and powerful GCC compiler to process C source code and to use AOP.

One can obtain the current AOP implementation for the C programming language from a LDV development site [14]. There it can be found as a part of *rule-instrumentor*. It's planed that an updated C AOP implementation will also be there soon.

REFERENCES

[1] Definitions of key AOP concepts. http://www.aosd.net/wiki/index.php?title=Main_Page

[2] AspectJ: an aspect-oriented extension to the Java programming language. http://www.eclipse.org/aspectj/

[3] An AspectJ example.

http://eclipse.org/aspectj/doc/released/progguide/starting-aspectj.html

[4] A. Khoroshilov, V. Mutilin, V. Shcherbina, O. Strikov, S. Vinogradov, and V. Zakharov, “How to cook an automated system for Linux driver verication,” 2nd Spring Young Researchers' Colloquium on Software Engineering, vol. 2, pp. 10-14, 2008.

[5] A. Khoroshilov, V. Mutilin, A. Petrenko, and V. Zakharov, “Establishing Linux driver verification process,” Perspectives of Systems Informatics, vol. 5947 of Lecture Notes in Computer Science, pp. 165-176, 2010.

[6] M. Gong, C. Zhang, and H.-A. Jacobsen, “AspeCt-oriented C,” Technology Showcase, CASCON 2007, Markahm, Ontario, 2007.

[7] W. Gong and H.-A. Jacobsen, “AspeCt-oriented C Language Spefication Version 0.8,” University of Toronto, 2008.

[8] J. Seyster, K. Dixit, X. Huang, R. Grosu1, K. Havelund, S. A. Smolka, S. D. Stoller, and E. Zadok, “Aspect-Oriented Instrumentation with GCC,” Procedings of the First International Conference on Runtime Verification, pp. 405-420, 2010.

[9] GCC plugins. http://gcc.gnu.org/wiki/plugins

[10] T. Ball and S.K. Rajamani, “SLIC: a Specification Language for Interface Checking (of C),” Technical Report MSR-TR-2001-21, Microsoft Research, 2002.

[11] The LLVM Compiler Infrastructure. http://llvm.org/

[12] Linux kernel 2.6.31.6. http://www.kernel.org/

[13] D. Beyer, T.A. Henzinger, R. Jhala, and R. Majumdar, “The software model checker Blast: Applications to software engineering,” Int. J. Softw. Tools Technol. Transf. 9(5), pp. 505-525, 2007.

[14] The LDV project. http://forge.ispras.ru/projects/ldv