ISSN 0361-7688, Programming and Computer Software, 2013, Vol. 39, No. 4, pp. 194–206. © Pleiades Publishing, Ltd., 2013.
Original Russian Text © E.M. Novikov, 2013, published in Programmirovanie, 2013, Vol. 39, No. 4.

An Approach to Implementation of Aspect-Oriented Programming for C

E. M. Novikov

Institute for System Programming, ul. Solzhenitsyna 25, Moscow, 109004 Russia

E-mail: novikov@ispras.ru

Received December 12, 2012

Abstract—This paper describes an approach to implementation of aspect-oriented programming (AOP) frameworks for C, outlines traditional AOP facilities for different programming languages, and shows how specific features of C and a build process of C programs affect AOP implementations. Next, we consider additional requirements imposed by a practical application of AOP implementations for C programs. Existing solutions are described and possibility of their use is analyzed. The paper describes a new AOP tool for C that implements the proposed approach and demonstrates its capabilities.

DOI: 10.1134/S0361768813040051

1. INTRODUCTION

Aspect-oriented programming (AOP) has emerged as a response to the question of how to "properly" decompose a large program into modules. One of the most interesting and simultaneously powerful ideas on decomposition criteria was the idea of D.L. Parnas to arrange each design decision into a separate module, especially concerning complex solutions that, most likely, will be revised in the future [1]. Parnas has shown that traditional facilities used to support modularity are often sufficient, but, sometimes, there is a clear lack of them, and additional ones are needed. In Russia, almost at the same time, in the 1970s, A.L. Fuksman had been investigating this problem. He justified the need to introduce special constructs into the practice of programming for "a concentrated description of dispersed operations" [2][1].

Programming languages currently in use, as a rule, do not support such constructs. For example, those module types that are available in procedural or object-oriented languages help to decompose a program system by grouping functionally related components and/or data. However, there are also other methods of decomposition, which may depend on architectural solutions intended, for example, for technological tasks rather than on a structure of a basic functionality implementation. An "additional" functionality corresponding to these problems was called **crosscutting concerns**.

[1] Note that the decomposition problem arises not only at the program design stage. During its lifecycle, a development project needs to be equipped with some program modules that are used for debugging, monitoring, optimization, security analysis, etc. On the one hand, these "modules" are not an integral part of a program; on the other hand, a general solution of the decomposition problem should consider these modules as well.

The description of crosscutting concerns has to be distributed through program code, which tangles it and leads to a code duplication. The probability of an error appearance increases because of incorrect modifications in parts of a program system associated with crosscutting concerns. Typical examples of crosscutting concerns are logging, tracing, configuration management, and error handling. More complex examples of crosscutting concerns can be found in ensuring safe access to systems, database handling, performing transactions, etc.

Aspect-oriented programming, a paradigm proposed in the 1990s, provides a special mechanism to supplement traditional facilities of modularity support related to basic functionality with ones related to crosscutting concerns. AOP extends the capabilities of the existing programming languages and smoothly complements them. AOP was originally developed for object-oriented programming languages (primarily, for Java); however, an AOP support can be provided also to procedural languages such as C.

This work was carried out within the project aimed at developing the LDV Linux driver verification toolkit [10–13]. This project treats AOP as a basic facility for automatic instrumentation of drivers' code before performing verification itself. Operating system drivers are usually written in C with the use of almost all facilities of the language. In view of this, we decided to develop a full-fledged AOP implementation that may become of value by itself rather than a highly specialized instrumentation tool. Therefore, the AOP features proposed in this study are useful for various C programs; but, some of these features address specific characteristics of code instrumentation of Linux kernel components (in particular, drivers). Accordingly, a part of this paper is devoted to AOP in general and to

specific features of AOP for C, while another part addresses details arising from needs of the LDV project.

Section 2 of this paper considers basic concepts of aspect-oriented programming and gives illustrating examples. Section 3 is devoted to traditional methods of aspect description. Section 4 discusses an impact of C specific features on implementation of AOP frameworks. Section 5 considers requirements for AOP frameworks for C within the LDV project. Section 6 describes the existing AOP frameworks for C. A tool that implements the proposed approach is presented in Section 7. Section 8 assesses possibilities of a practical use of this tool and existing AOP implementations for C. Section 9 summarizes the discussion and considers directions of further development.

2. BASIC CONCEPTS OF AOP

One of the key concepts of aspect-oriented programming is a **join point**. There exist different definitions of this concept [3, 4, 8, 9, 14–16]. This paper defines the join point as a program construct that can be associated with a description of some part of crosscutting concerns of a program. Typical examples of the join points are functional calls and structure declarations.

A **pointcut** is a description of a set of join points logically grouped by some condition. For example, a pointcut can describe all calls of memory allocation functions (such as *malloc*, *calloc*, etc.).

One more concept of AOP is an **advice**. An advice specifies a set of actions that must be executed for join points described by a pointcut[2]. For example, for such a join point as a function call, an advice may include instructions for logging a message containing a value returned by the given function.

Pointcuts and advices allow AOP to extract a description of a part of crosscutting concerns into separate modules — the so-called **aspects**[3]. The rest of this section gives an example of an aspect and considers how pointcuts and advices can be specified.

Along with the possibility of describing crosscutting concerns as aspects, AOP involves tools for automatic binding of aspects with a target program. In this process, essentially for some representation of program code[4], potential join points are matched with pointcuts defined in an aspect. If this match is detected and there is an advice for the given pointcut, the corresponding join point should be framed by the code given in the advice. The process of binding aspects with the target program is called **aspect weaving**.

To implement AOP for some programming language, it is necessary to determine how to describe aspects and develop an aspect weaver. At present, there are a large number of AOP implementations for different programming languages: AspectJ [4] for Java, AspectC++ [8] for C++, Aspect.NET [9] for Microsoft.NET, ACC [14, 15], InterAspect [16], SLIC [18] for C, etc. As an example, we consider AspectJ, one of the most advanced and well-known AOP frameworks to date.

Aspects in AspectJ are being developed with the help of the same-named aspect-oriented extension of Java programming language [5]. Figure 1 shows the aspect *Logging*, which, for a graphical system, extracts crosscutting concerns of logging [6]. To do this, the aspect specifies the named pointcut[5] *move*. The join points described by *move* are calls of the method *setXY* of the class *FigureElement* and calls of the methods *setX* and *setY* of the class *Point*. In addition, the aspect *Logging* specifies an advice. This advice says that, before execution of join points described by the named pointcut *move* (i.e., before calls of appropriate methods), a message should be printed to the screen. Aspect weaving in AspectJ is performed at a program bytecode level using *ajc*, a special-purpose compiler of Java and its extension AspectJ [7]. The weaved bytecode can be interpreted by standard Java virtual machines.

In the development of an approach to implementation of AOP for C the experience of AspectJ was actively used. Also, we took into account experience of the existing AOP frameworks for C and C++. The next section presents well-established methods of aspect description that are applicable to various programming languages.

3. TRADITIONAL METHODS FOR ASPECT DESCRIPTION

The majority of the existing AOP implementations offer the extensions of those programming languages that are used for writing target programs. In AspectJ, AspectC++, and Aspect.NET, as well as, probably, in other AOP frameworks for object-oriented programming languages, aspects are largely similar to classes. Aspects may have their own fields and methods; they are supported by mechanisms similar to class inheritance and polymorphism. The AOP frameworks for C do not support such abstraction. Instead, they propose to describe aspects as separate files. As for other concepts, the AOP frameworks for different programming languages are more similar.

[2] More precisely, these join points should in some manner be framed or "decorated" with the corresponding code, which may contain both instructions and certain definitions or declarations

[3] Here, the term “aspect” has a highly specialized meaning used only in the literature on AOP.

[4] We note in advance that different types of aspect-related processing can be conducted with different types of program representation such as textual, intermediate, binary, etc.

[5] A named pointcut concept is formally considered in the next section

```
// Aspect consists of a named pointcut and an advice.
aspect Logging {
  // Named pointcut specifies matching to join points of a program
  // (calls of methods);
  pointcut move ():
    call (void FigureElement.setXY(int, int)) ||
    call (void Point.setX(int))               ||
    call (void Point.setY(int));
  // Advice prints a message on the screen before
  // an execution of a program join point
  // matched by the given named pointcut.
  before(): move() {
    System.out.println("about to move");
  }
}
```

Fig. 1. Example of an AspectJ aspect that extracts crosscutting concerns of logging for a graphical system.

Traditionally, the join points are divided into static and dynamic join points. The **static join points** are program declarations, such as structures, classes, functions, methods, variables, structure or class fields, etc. The **dynamic join points** correspond to events that can occur during a program execution. Examples of the dynamic join points are a function or method call, an assignment of a value to a variable or a field, an initialization of a structure or a class, etc.

Traditionally, a pointcut that describes a set of static join points is specified as a corresponding entity signature. A signature is allowed to be recorded by using the so-called **wildcards**. The wildcards make it possible to set a match to any type, a part of an entity name, or an arbitrary list of parameters. In the AspectJ example shown in Fig. 1, the methods *setX* and *setY* of the class *Point* can be described by using the signature "* *Point.set**(..)".

As a rule, a pointcut describing a set of dynamic join points is specified by adding a keyword to a pointcut that describes a set of static join points. This keyword reflects an essence of the relevant event. The existing AOP frameworks support the following pointcuts describing dynamic events[6]:

- **call**—a function call;
- **execution**—a function execution occurring after control is transferred from a function that calls the given function to that function;
- **set**—setting a value to a variable or a field;
- **get**—using a variable or a field.

It is worth noting that AOP traditionally supports an implicit description of join points based on their context, for example:

- **infile**—all join points of some file;
- **infunc**—all join points of some function;
- **cflow**—all join points that are found in a context of execution of join points of another pointcut.

The considered pointcuts for describing join points are called **primitive pointcuts**. Combinations of pointcuts that can be obtained by using operators "and" (intersection), "or" (union), "not" (exclusion), and brackets (grouping) are called **composite pointcuts**. The composite pointcuts play a special role in refining a description of a set of join points, for example, by combining explicit and implicit methods for specifying join points. A **named pointcut** is a primitive or composite pointcut that is bound with a certain name by which the given pointcut can be referred to. In the example shown in Fig. 1, *move* is a named pointcut combining three primitive pointcuts.

In AOP, an advice traditionally consists of a declaration and a body. The **advice declaration** contains a pointcut and also, when the pointcut describes a set of dynamic join points, one of the keywords *before*, *after*, or *around*. The pointcut determines a condition when the advice should be used. The keywords indicate how this should be done: *before* – before, *after* – after, and *around* – in place of[7]. The **advice body** contains code that should frame the corresponding join points. As already noted, the advice of the example shown in Fig. 1 says that before the execution of the join points

[6] From now on, in this section, we will not consider join points that are of a purely object-oriented nature and have no procedural counterparts, because the aim of this paper is to develop an approach to the implementation of AOP for C.

[7] Instead of the literal meaning of "around", most AOP frameworks use *around* advices primarily to cancel an execution of the corresponding dynamic join point. In accordance with this, we proposed a more relevant name, although we will use the word *around*.

described by the named pointcut *move*, the message should be printed to the screen.

Normally, an advice body is written with the help of instructions of the same programming language in which the target program is written. Along with this, the AOP frameworks allow special instructions to be used in advice bodies. For example, by using the instruction *proceed*, one can write a function call that is specified as matching by an advice pointcut. With special instructions in advice bodies, one can use the so-called **reflection information**, which is information on the corresponding join point and its context. Traditionally, the following reflection information is supported for the join points:

1. For function calls and executions:
   - a name of a called function;
   - a type of a returned value;
   - types and a number of arguments.
2. For function calls (additionally):
   - actual parameters;
   - a returned value;
   - a file and a function in a context of which the call occurs.
3. For assignments and uses of variables and fields:
   - a name of a variable or a field;
   - a value being assigned or a current value of a variable or a field;
   - a file and a function in a context of which the assignment or the use occurs.

An important problem that traditionally arises in the aspect description is the order of application of several advices when their pointcuts correspond to the same join point. Different AOP implementations offer slightly different solutions. For example, AspectJ offers the following algorithm of application of several advices for the same join point. Among advices of the same type (*before*, *after*, *around*), the earlier an advice occurs in the aspect, the earlier it is applied. First of all the *around* advices are applied in the following way:

(1) If a body of a currently applied advice contains no special instruction *proceed*, its application is terminated.

(2) Otherwise, a part of this advice is applied until the instruction *proceed*; instead of this instruction, a next *around* advice (if available) is applied or the algorithm

- applies all *before* advices;
- executes directly the join point itself;
- applies all *after* advices.

Then, the remaining part of the given advice is applied.

For more complex cases, when, for example, the program is weaved with several aspects, it is traditionally believed that the behavior of the aspect weaver is undefined.

Normally, advices for static join points are treated separately from advices for dynamic join points. Many AOP frameworks offer an ability to add fields into definitions of composite data types, such as structures and unions. Specification of advices for static join points is of particular importance for object-oriented languages because of a for them this allows to describe in aspects crosscutting concerns that are associated with encapsulation, inheritance, and polymorphism.

We have considered main traditional methods for the aspect description. These methods, to some degree, are supported by AOP frameworks for various programming languages. Section 6 shows that the existing AOP frameworks for C do not go beyond traditional methods for the aspect description. However, the C language and a build process of C programs have several specific features that should be taken into account in developing AOP framework.

4. INFLUENCE OF SPECIFICS OF C AND A BUILD PROCESS OF C PROGRAMS ON IMPLEMENTATION OF AOP FRAMEWORK

A key feature of C is that this language supports address arithmetic. For AOP, it is of special interest to consider variables or fields that have a pointer type, as well as various operations with them. All that is supported by AOP frameworks for variables and fields automatically applies to the pointers. However, for the dynamic join points, pointers need to be additionally supported by operations such as taking of a variable or a field address, a pointer dereference, an assignment to and a use of array elements.

Another feature of C is a mechanism of a build process of C programs. C is one of few programming languages for which compilation and linking of program code is preceded by preprocessing that is based on a configuration specified by build files and the user. Preprocessing is an integral feature of C (for example, Java has no such feature); therefore, this stage of source code processing should be considered in AOP frameworks for C. The main actions that are performed successively for all program files during this process are conditional compilation, inclusion of header files, and macro expansion.

Header files inclusion may be associated with AOP concepts as follows. Each header file included into some file and the file itself can be considered as static join points. As a signature that allows a set of such join points to be described, one can consider a path template allowing one to use a wildcard symbol denoting a sequence of characters of arbitrary length. Also, one can use a special notation for a currently preprocessed file. This makes it possible to develop aspect files for crosscutting concerns related to header files inclusion. This can be used, for example, to add some auxiliary preprocessor directives (thus affecting a program configuration), function prototypes, etc.

Macros are largely similar to functions. For example, a macro has a named parameterized definition,

while macro expansion is used to replace formal parameters in the definition by actual parameters and substitute the result into program code. Due to this analogy, macros can be reasonably considered as static join points. To describe a set of these join points, one can use a signature with support of wildcards for specifying both a part of a macro name and a list of parameters of arbitrary length. Again, by analogy with functions, one can consider for macros dynamic join points, such as macro "execution" and macro "call" (expansion). In a natural way, one can port a description of these join points through their context (*infile*) and a use of reflection information in advices, such as a macro name, actual parameters, and a context. With the help of advices for macros, one can add some auxiliary code in place of (or before, or after) the substituted code.

We have considered the characteristics of C and the build process of C programs that affect implementation of AOP frameworks for C. It is equally important to take into account characteristics of a practical application of AOP frameworks, which will be addressed in the next section.

5. SPECIFIC FEATURES OF PRACTICAL APPLICATION OF AOP FRAMEWORKS FOR C

Our approach to the implementation of the AOP frameworks for C was strongly affected by the Linux Driver Verification (LDV) project [10–13]. The aim of LDV is to provide a toolkit allowing one to use various static analysis tools in order to check if Linux drivers satisfy a certain set of correctness rules.

It turned out that an AOP framework for C is an appropriate framework to formalize correctness rules regardless of a static analysis tool and then to instrument the source code of drivers to be checked.

This use case required from AOP frameworks for C the following:

• Support C with all extensions of the GCC compiler as a source language (this is a standard language for writing drivers of the Linux operating system) and support standard and GCC options of C programs build.

• Offer a large set of AOP facilities allowing extracting crosscutting concerns of C programs into aspects. This is required due to a variety of correctness rules to be checked. In particular, formalization of correctness rules requires a model state and model functions that are essentially similar to fields and methods of aspects in AOP frameworks for object-oriented programming languages (an example can be seen in Section 8). Here, it is important that the use of the proposed facilities of AOP for developing aspects should be rather convenient and intuitively understandable.

• An aspect weaver must output a correct C program that is equivalent to the original program with the only exception that it is extended by a description of appropriate crosscutting concerns. This is required for subsequent use of static analysis tools.

• AOP frameworks should be sufficiently easy to maintain and to extend with new abilities. This requirement came from practice. For example, formalization of new correctness rules sometimes requires support of additional types of join points and reflection information.

It should be noted that the specific requirements formulated above can be considered in terms of extracting crosscutting concerns for any C program with a correction that it is done for Linux. For example, most programs operating under Linux require support of C with GCC extensions. To efficiently extract crosscutting concerns, one needs a large set of AOP facilities, and the use of AOP frameworks should be convenient. The requirement that an aspect weaver output should be a C program is also useful for debugging AOP frameworks.

6. EXISTING AOP FRAMEWORKS FOR C

AOP frameworks for C have been developed to a lesser extent in comparison with those for Java. At present, the most advanced AOP framework for C is ACC (AspeCt-oriented C) [14]. An extension of ACC for C [15] used for developing aspects is similar to the extension for Java made in AspectJ. Figure 2 presents an aspect example written in this extension of C. The aspect consists of one advice saying that, after all calls of the function *foo2*, a message that contains a return value of this function should be printed on the screen. An approach to aspect weaving in ACC differs from the one of AspectJ. For each input preprocessed file, the special compiler *acc* generates a C file extended by a description of relevant crosscutting concerns.

ACC supports a rather large set of AOP facilities for extracting crosscutting concerns. ACC does not support such traditional AOP facilities as dynamic join points for assignments and uses of fields, as well as the order of advices application to the same join point when *around* and *before/after* advices should be applied simultaneously.

As for specific facilities of AOP for C, ACC fails to affect the preprocessing process because an already preprocessed source code is taken as an input. Also, ACC offers no way to describe pointer operations in pointcuts.

When using ACC, it is necessary to set a model state and model functions manually because these features are not supported by the framework. ACC is released under GNU General Public License version 2, which allows its modification. But ACC uses a parser of C and its aspect-oriented extension, which was generated on the basis of a closed grammar. This parser cannot handle some GCC extensions. Therefore, ACC cannot be used, for example, for instrumentation of source code of latest versions of Linux drivers. For large and complex programs, ACC outputs C code,

```
// Advice prints a message and a return value
// of function foo2 after its call.
after (int res):
 call(int foo2(int)) && result(res) {
 printf("after call foo2, return %d\n", res);
}
```

Fig. 2. Example of an ACC aspect.

```
static void instrument_malloc_calls() {
  /* Creating a primitive pointcut matching calls
       of function 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());
  aop_join_on(pc, malloc_callback, NULL);
}
```

Fig. 3. Example of an InterAspect pointcut.

which is not equivalent to the original one and is incorrect in some cases. The framework is currently not supported by its developers and modification of its parser is difficult; therefore, further development of the given AOP framework is difficult.

InterAspect is one of the latest AOP frameworks for C [16]. This framework is interesting primarily because it is based on GCC plugins [17]; due to this, all GCC extensions are supported. To develop aspects, InterAspect provides a special AOP library for C rather than an extension of C. Figure 3 shows how this library can be used to define a primitive pointcut describing calls of the function *malloc*. This can be done by creating the corresponding pointcut (*aop_match_function_call*) and setting limitations on the names of called functions (*aop_filter_call_pc_by_name*), on the types of their arguments (*aop_filter_calls_pc_by_param_type*), and on the return value type (*aop_filter_call_pc_by_return_type*), which is a rather labor-consuming task.

InterAspect supports a significant number of traditional AOP facilities. The most notable shortcoming of the framework is lack of support for *around* advices. Also, InterAspect fails to specify join points by their context, does not support composite pointcuts, provides insufficient reflection information on join points, and fails to set advices for static join points. Like ACC, InterAspect does not support AOP facilities related to specific features of C and a build process of C programs. InterAspect operates at a low-level internal representation of GCC, extending it by describing crosscutting concerns. A tool output, object or binary code, is generated by GCC, which does not allow InterAspect to be used directly for static analysis, as well as creates inconveniences in debugging the framework. InterAspect is being developed passively. The tool is released under GNU General Public License version 3, which makes it possible to extend its functionality.

One of the most promising (in terms of instrumentation) approaches was implemented in SLIC (specification language for interface checking) and its preprocessor [18]. SLIC is a C like language for developing specifications. In fact, a SLIC specification is nothing else than an aspect. In bodies of SLIC advices, one can use conditional expressions, simple assignments, references to function parameters and their return value, as well as special instructions for static analysis tools. SLIC supports a setting of a model state. The SLIC specification shown in Fig. 4 demonstrates an ability to check errors in programs using AOP. In this example, we have a wrong situation, where a queue has more than four zeros. The specification determines the model state of the program, *zero_cnt*, which serves to count the number of zeros in the queue. The advice is specified for an enter to the function *put*, which checks the number of zeros in the

```
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;
}
```

Fig. 4. Example of a SLIC specification.

queue and increases *zero_cnt* if no error occurred. Similarly, an advice is specified for an exit from the function *get*.

SLIC is used during verification of Microsoft Windows drivers. The SLIC preprocessor instruments drivers' source code on the basis of a specification. The preprocessor outputs a C program that is equivalent to the original program, which is extended by additional checks. Then, the given program is checked using a static analysis tool.

Due to specifics of its application area, SLIC has never focused on being a full-fledged AOP framework for C. It supports only a few dynamic join points (in fact, only a function call and execution). SLIC fails to set join points by their context, does not support named pointcuts, and supports merely a single composite pointcut ("or"). The *around* advices are not supported, and a small amount of reflection information can be used in advice bodies. SLIC does not allow several advices to be specified for the same join point and does not support advices for static join points.

Like ACC and InterAspect, SLIC does not support AOP facilities related to specific features of C and a build process of C programs.

Source code of the SLIC preprocessor is closed unlike InterAspect and ACC, which particularly does not allow one to extend its application domain for extracting crosscutting concerns of C programs with GCC extensions. As already noted, SLIC allows setting a model state. Model functions are not supported. Since the tool is used rather actively to solve important industrial problems, we can assume that SLIC is supported at a sufficiently high level.

This paper does not address other AOP frameworks for C, such as C4, Aspicere2, Xweaver, and WeaveC, because they offer fewer useful features in comparison with the ones described above. Also, this review omits AOP frameworks for C++. These tools can be adapted for extracting crosscutting concerns of C programs; however, for developing and weaving aspects, they widely use concepts and facilities of object-oriented programming.

Thus, the existing AOP frameworks for C do not offer all considered facilities of AOP. Most of the considered frameworks have fundamental limitations for adding an extra support of some features. This conclusion has led us to the need to develop our own AOP framework for C.

7. C INSTRUMENTATION FRAMEWORK

The proposed approach to implementation of aspect-oriented programming frameworks for C was implemented in C Instrumentation Framework (CIF). To develop aspects, CIF proposes to use a C extension similar to AspectJ, AspectC++, ACC, SLIC, and many other AOP extensions for various programming languages. Aspects are stored as files separately from program source code. A CIF aspect file example can be found in the next section.

CIF supports the majority of the AOP facilities for C described in the proposed approach. All traditional facilities of AOP, as well as the possibility to set pointcuts for including header files, "executing", and expanding macros, are supported. To date, like in other AOP frameworks for C, CIF does not support pointcuts for operations with pointers.

In developing CIF, much attention was paid to its practical application, particularly for instrumentation of Linux drivers' code before its static analysis. Due to the proposed architecture, the framework accepts source code of programs in C with all GCC extensions as input, as well as supports the standard and GCC build options for C programs. After aspect weaving, CIF can output both the C code (which, for example, can be transferred to a static analysis tool) and those representations of the program code that are supported by GCC.

The Table 1 compares supported features of CIF and the existing AOP frameworks for C described in Section 6. As one can see, CIF supports almost all required facilities.

Let us consider the architecture of CIF in more detail. Originally, the architecture was based on the LLVM infrastructure [20]. It turned out that this approach has a number of shortcomings, especially from the point of view of practical application. Therefore, this work proposed the new architecture.

At the input, CIF receives an aspect file, an unpreprocessed C file, and a set of options of preprocessing and compilation[8]. The tool further operates like the standard preprocessor and the C compiler, except that CIF additionally performs aspect weaving of source code. Aspect weaving is conducted automatically in five stages. At each stage, CIF calls with appropriate options a modified version of GCC 4.6.1 that includes

[8] An unpreprocessed C file and a set of preprocessing and compilation options can be obtained, for example, on the basis of program build files.

Table 1. Comparison of C Instrumentation Framework with existing AOP frameworks for C

Characteristics of an AOP framework		ACC	InterAspect	SLIC	CIF
Support of traditional AOP facilities	Pointcuts for static join points including support of wildcard symbols	+	+	±	+
	Pointcuts for dynamic join points	±	±	±	+
	Pointcuts for specifying join points by their context	+	–	–	+
	Composite and named pointcuts	+	±	±	+
	Before, *after*, and *around* advices for dynamic join points	+	±	±	+
	Arbitrary correct C code in advice bodies	+	+	±	+
	Special instructions in an advice body including reflection information	+	±	±	+
	Order of application of advices	±	±	–	+
	Advices for static join points	+	–	–	+
Support of AOP facilities for C	Pointcuts for inclusion of header files	–	–	–	+
	Pointcuts for "executions" and expansions of macros	–	–	–	+
	Pointcuts for operations with pointers	–	–	–	–
Possibilities of a practical application	C with GCC extensions including GCC build options	±	+	±	+
	Model state and model functions	–	+	±	+
	Output of an aspect weaver in C	±	–	±	+
	Extensibility of framework possibilities	±	+	–	+
	Support of a framework	–	±	+	+

the preprocessor and the C compiler [21]. The modified version of GCC has the following:

• an aspect parser that checks lexical, syntactic, and semantic correctness of aspect files and converts them into an internal representation of CIF (advice bodies are considered only to find specific instructions in them);

• a component intercepting program elements (join points of a program) processed by the preprocessor and the compiler of GCC and converting it into an internal representation of CIF;

• a component that matches join points and pointcuts of advices given in the aspect;

• a component that frames matched join points on the basis of advice requirements;

• a component that allows one to print C code on the basis of the high-level internal representation of GCC.

Let us consider aspect weaving at each of the five stages and use of input data in CIF. For convenience, we denote an input aspect file as *a.aspect* and an unpreprocessed C file as *a.c*.

The first stage is auxiliary. At this stage, the aspect file *a.aspect* is preprocessed by means of the GCC preprocessor. In this case, directives starting with the character "@", rather than the standard character "#", are interpreted. Options of aspect preprocessing can be set by the CIF option *aspect-preprocessing-opts*. We denote an output file as *a.aspect.i*. Preprocessing of the aspect files can be used, for example, to develop a library of aspect files, which is an important task for a wide practical application.

At the second stage, CIF inserts an additional text before or after *a.c* code. Since this is done before invoking the preprocessor, that text can contain different preprocessor directives. We denote the output file as *a.prepared*.

At the third stage, CIF runs the standard GCC preprocessor, which gets the file *a.prepared* and preprocessing options as inputs. As preprocessing proceeds, aspect weaving of "executions" and expansions of macros occurs in accordance with the advices of the aspect file *a.aspect.i*. Here, on the basis of information on the join point, special instructions are replaced in advice bodies. The preprocessed file obtained at the output is denoted as *a.macroweaved*.

At the fourth stage, CIF inputs the file *a.macroweaved* and compilation options to the standard GCC compiler. At this stage, the compiler parses the input file and converts it into its internal high-level

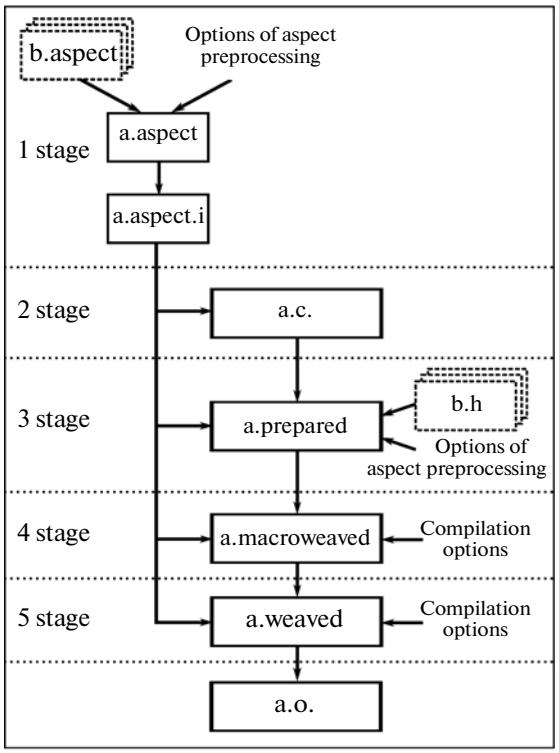

Fig. 5. C Instrumentation Framework workflow.

representation. Simultaneously, the special component of GCC intercepts processed definitions and calls of functions, assignments, and uses of variables and fields, as well as declarations of composite data types. If they are matched by the advices specified in the aspect file *a.aspect.i*, composite data types are complemented by fields at the level of textual representation of the file *a.macroweaved*, and for functions, variables, and fields, auxiliary functions are created to be appended at the end of the file *a.macroweaved*. Bodies of auxiliary functions are generated on the basis of bodies of respective advices. Special instructions in advice bodies' code are replaced by their values on the basis of the obtained information on the join point. It makes sense to note that advice bodies can contain an arbitrary valid code in C with GCC extensions. This code is parsed and checked by the GCC compiler at the next stage of the CIF operation. We denote the file obtained at this stage as *a.weaved*.

At the fifth and final stage, CIF inputs the file *a.weaved* and compilation options to the standard GCC compiler. The further operation is largely similar to that of the fourth stage, except that declarations of composite data types are not intercepted, while, for functions, variables, and fields, the corresponding dynamic join points and auxiliary functions are linked at the level of the high-level internal representation of GCC. If CIF runs with the option – *back-end=src*, as the compiler completes parsing of program elements, their high-level representation is converted into C code, which is printed into an output file specified by means of a special option. It should be noted that all transformations maintain links with original source code of a program by adding line directives. If the value of the CIF option *–back-end* is one of *asm*, *obj*, or *bin*, the GCC compiler continues its work in a standard way and outputs one of the program code representations that are supported by the GCC compiler (assembler, object, or binary code respectively). The output file is denoted as *a.o*.

Figure 5 shows a CIF workflow, which clearly demonstrates the way how input data is used and converted. Here, the boxes with dotted borders indicate some external data.

8. PRACTICAL APPLICATION OF C INSTRUMENTATION FRAMEWORK

C Instrumentation Framework was included into the LDV toolkit [10–13]. LDV rules that describe correct use of the Linux kernel interface by drivers are manually formalized as aspects in the following way:

(1) One analyzes that part of the Linux kernel interface for drivers that is related to a given rule. In the majority of cases, one obtains macros and functions the correct use of which is specified by the rule.

(2) In an aspect, one specifies a model state and model functions where checks required by the rule and the corresponding changes in the model state are performed.

(3) With the help of advices, one specifies in the aspect the linkage of model functions to those places where respective interface of the Linux kernel is used.

(4) In different parts of the aspect, auxiliary elements are specified.

CIF is used for automatic instrumentation of drivers' source code aimed at its further checking by means of static analysis tools.

As an example, we consider one of LDV rules describing the correct registration of a Gadget class USB device. When this rule is violated, the entire system may crash [22]. The rule is as follows:

(1) Primarily it is necessary to register in an arbitrary order a class of devices and a range of device numbers, and, then, register a Gadget class device. No re-registration of resources is allowed.

(2) After using the device, it is necessary to deregister, first, the Gadget class device and, then, the class of devices and the range of device numbers in an arbitrary order.

In formalizing the rule, it should be taken into account that registration functions may not work because of some reason. In this case, one should also deregister registered resources. For example, if the Gadget class device cannot be registered, it is necessary to arbitrarily deregister the class of devices and the range of device numbers.

An analysis of the Linux kernel interface revealed the following macros and functions that are relevant to the rule:

(1) Macro "*class_create* (*owner*, *name*)" – creation and registration of the device class.

(2) Function "*void class_destroy*(*struct class* ∗)" – deregistration and destruction of the device class.

(3) Macro "*class_register*(*class*)" – registration of the device class.

(4) Function "*void class_unregister*(*struct class* ∗)" – deregistration of the device class.

(5) Functions "*int alloc_chrdev_region*(*dev_t* ∗, *unsigned*, *unsigned*, *const char* ∗)" and "*int register_chrdev_region*(*dev_t*, *unsigned*, *const char* ∗)" – registration of the device number range.

(6) Function "*void unregister_chrdev_region*(*dev_t*, *unsigned*)" – deregistration of the device number range.

(7) Function "*int usb_gadget_register_driver*(*struct usb_gadget_driver* ∗ *driver*)" – registration of the Gadget class device (up to Linux kernel version 2.6.37).

(8) Function "*int usb_gadget_probe_driver*(*struct usb_gadget_driver* ∗ *driver*, *int*(∗*bind*)(*struct usb_gadget* ∗))" – registration of the Gadget class device (starting with Linux kernel version 2.6.37).

(9) Function "*int usb_gadget_unregister_driver*(*struct usb_gadget_driver* ∗ *driver*)" – deregistration of the Gadget class device.

Figure 6 shows a part of the aspect file specifying the model state and model functions (comments and fragments of similar code are omitted). The model state and model functions are placed into a separate file with the help of the special advice *new* (line 32) to prevent from repeating for several C files which can be linked together. Here, this file is given by the environment variable *LDV_COMMON_MODEL*.

Line 33 includes an auxiliary header file that defines an interface of static analysis tools:

(*) Functions *ldv_under_ptr* (used in line 46) and *ldv_undef_int_nonpositive* (used in line 58) return an undefined pointer and an undefined non-positive integer respectively. These functions are necessary to simulate possible failures of resources registration.

(*) Macro *ldv_assert* (used in lines 49, 50, etc.) checks the condition received as a parameter. If a violation of this condition is detected, the static analysis tool will report a possible error.

Line 34 includes the auxiliary header file that determines model functions to handle errors transmitted by function return values of the pointer type. For example, line 47 uses the special macro *LDV_PTR_MAX*, which specifies the maximum possible value of the pointer that brings no information about an error.

Lines 35–38 specify possible values of the model variable *ldv_class*: *LDV_CLASS_UNREGISTER* corresponds to the state in which the device class is not registered and *LDV_CLASS_REGISTER* to the state where it is registered.

Lines 40–42 define and initialize variables that represent the model state. At the beginning of program execution, all resources are unregistered.

Lines 43–54 define the model function *ldv_create_class* corresponding to the Linux kernel interface macro *class_create*, which is responsible for the creation and registration of the class of devices. *ldv_create_class* simulates a possible failure in the registration of the device class (lines 45–47 and 53) and checks that the Gadget class device was not registered before the registration of the device class (line 49) and that the registration of the device class is not repeated (line 50). If these checks are passed, the model state is changed; namely, the variable *ldv_class* is set to *LDV_CLASS_REGISTER*.

The macro *class_register* corresponds to the model function *ldv_register_class* (lines 55–66). In general, this function is similar to *ldv_create_class*, except that a possible failure in the device class registration is modeled using integers rather than pointers (lines 57–59 and 65).

Lines 68–73 determine the model function *ldv_unregister_class* corresponding to the Linux kernel interface functions *class_destroy* and *class_unregister*. The function *ldv_unregister_class* checks if the Gadget class device is unregistered and if the device class is registered. Then, the device class is deregistered.

Lines 75–80 define a special model function that is executed when the driver operation is terminated. This function checks if all resources were deregistered before the final state.

Figure 7 shows a part of the aspect file that describes the binding of model functions (lines 12, 15, etc.) to macro "executions" (lines 11 and 14) and function calls (lines 17, 20, etc.). For this rule, no binding to a function execution is possible because implementations of these functions are unavailable for static analysis of driver' source code.

It should be noted that, by using a composite pointcut that specifies calls of functions *class_destroy* and *class_unregister* (line 17), one can call the same model function *ldv_unregister_class* (line 18) instead of them. A similar technique is used for calls of the functions *alloc_chrdev_region* and *register_chrdev_region*, as well as *usb_gadget_register_driver* and *usb_gadget_probe_driver*. The latter particularly makes it possible to use the given aspect file for instrumentation of Linux kernel drivers' source code of different versions (up to 2.6.37 and higher).

Figure 8 shows an auxiliary part of the aspect file. Line 01 includes a special aspect header file specifying the binding of model functions for processing errors transmitted through function return values of the pointer type, to executions of respective functions of the Linux kernel interface. Line 02 says that instrumented driver code should be preceded by model function prototypes (lines 03–09). This is needed because, before their usage, the functions must be

```
32 new: file(LDV_COMMON_MODEL) {
33 #include <verifier/rcv.h>
34 #include <kernel-model/err.h>
35 enum {
36   LDV_CLASS_UNREGISTERED,
37   LDV_CLASS_REGISTERED
38 };
39 // Similarly for the range of device numbers and a
   // Gadget class device.
40 static int ldv_class = LDV_CLASS_UNREGISTERED;
41 static int ldv_region = LDV_REGION_UNREGISTERED;
42 static int ldv_usb_gadget = LDV_USB_GADGET_UNREGISTERED;
43 void *ldv_create_class(void)
44 {
45   void *is_got;
46   is_got = ldv_undef_ptr();
47   if (is_got <= LDV_PTR_MAX)
48   {
49     ldv_assert(ldv_usb_gadget == LDV_USB_GADGET_UNREGISTERED);
50     ldv_assert(ldv_class == LDV_CLASS_UNREGISTERED);
51     ldv_class = LDV_CLASS_REGISTERED;
52   }
53   return is_got;
54 }
55 int ldv_register_class(void)
56 {
57   int is_reg;
58   is_reg = ldv_undef_int_nonpositive();
59   if (!is_reg)
60   {
61     ldv_assert(ldv_usb_gadget == LDV_USB_GADGET_UNREGISTERED);
62     ldv_assert(ldv_class == LDV_CLASS_UNREGISTERED);
63     ldv_class = LDV_CLASS_REGISTERED;
64   }
65   return is_reg;
66 }
67 // Similarly for ldv_register_region and
   // ldv_register_usb_gadget.
68 void ldv_unregister_class(void)
69 {
70    ldv_assert(ldv_usb_gadget == LDV_USB_GADGET_UNREGISTERED);
71    ldv_assert(ldv_class == LDV_CLASS_REGISTERED);
72     ldv_class = LDV_CLASS_UNREGISTERED;
73 }
74 // Similarly for ldv_unregister_region and
   // ldv_unregister_usb_gadget.
75 void ldv_check_final_state(void)
76 {
77   ldv_assert(ldv_class == LDV_CLASS_UNREGISTERED);
78   ldv_assert(ldv_region == LDV_REGION_UNREGISTERED);
79   ldv_assert(ldv_usb_gadget == LDV_USB_GADGET_UNREGISTERED);
80 }
81 }
```

Fig. 6. Model state and model functions for the LDV rule describing correct registration of the Gadget class USB device.

```
11 around: define(class_create(owner, name)) {
12   ldv_create_class()
13 }
14 around: define(class_register(class)) {
15   ldv_register_class()
16 }
17 around: call(void class_destroy(..)) ||
           call(void class_unregister(..)) {
18   ldv_unregister_class();
19 }
20 around: call(int alloc_chrdev_region(..)) ||
           call(int register_chrdev_region(..)) {
21   return ldv_register_region();
22 }
23 around: call(void unregister_chrdev_region(..)) {
24   ldv_unregister_region();
25 }
26 around: call(int usb_gadget_register_driver(..)) ||
           call(int usb_gadget_probe_driver(..)) {
27   return ldv_register_usb_gadget();
28 }
29 around: call(int usb_gadget_unregister_driver(..)) {
30   ldv_unregister_usb_gadget();
31 }
```

Fig. 7. Binding of model functions with "executions" of macros and calls of functions of the Linux kernel interface.

```
01 @include <kernel-model/err.aspect>
02 before: file("$this") {
03 void *ldv_create_class(void);
04 int ldv_register_class(void);
05 void ldv_unregister_class(void);
06 int ldv_register_region(void);
07 void ldv_unregister_region(void);
08 int ldv_register_usb_gadget(void);
09 void ldv_unregister_usb_gadget(void);
10 }
```

Fig. 8. Auxiliary include of an aspect header file and definition of model function prototypes.

defined or prototypes should be provided for them. Since model functions are defined in a separate file, they are provided with prototypes. In addition, different parts of the aspect file shown in Figs. 6–8 contain auxiliary model comments that are omitted for brevity.

Currently we have formalized 44 rules of the LDV project in form of aspects. Using these aspects, CIF has successfully instrumented source code of more than 95% of drivers (their total number is around 2000–4000 depending on the Linux kernel version from 2.6.31.6 to 3.7-rc4 [23]). The remaining drivers were not processed due to technical problems in the GCC component that prints C code on the basis of the internal high-level representation of the compiler. Since this component is being developed within this work, these problems will be potentially resolved.

The instrumented code was checked by means of the static verification tools BLAST [24] and CPAchecker [25]. As a result, we have detected 75 real bugs in Linux drivers that were reported to the kernel developers [26]. All these errors were fixed.

9. CONCLUSIONS

Aspect-oriented programming provides a special mechanism to extend modularity support facilities that can be found in the existing programming languages. AOP frameworks were originally developed for object-oriented programming languages (primarily,

for Java). This paper considers approaches to implementation of the AOP frameworks for C. To this end, we consider the traditional ways of aspect description for different programming languages, as well as the influence of specific features of C and the build process of C programs on AOP frameworks. Particular emphasis has been placed on the possibility of using of the AOP frameworks for C in practice (particularly for instrumentation of Linux drivers' source code before its static analysis). This paper describes capabilities and shortcomings of the existing solutions. Also, it presents the new AOP framework for C–C Instrumentation Framework. This framework excels the existing solutions in all indicators.

Further development of CIF will be aimed at improving support of the AOP facilities specific to C (particularly operations with pointers), as well as at expanding the CIF application domain.

Information on the current version of C Instrumentation Framework can be found at the project website [19].

ACKNOWLEDGMENTS

The work was supported by the Federal Target Program "Studies and developments on priority fields of scientific advancement in Russia for 2007–2013" (contract no. 07.514.11.4104).

REFERENCES

1. Parnas, D.L., On the criteria to be used in decomposing systems into modules, *Comm. ACM*, 1972, vol. 15, no. 12, pp. 1053–1058.
2. Fuksman, A.L., *Technological Aspects of Software Construction* Moscow: Statistika, 1979.
3. Definitions of aspect-oriented programming concepts. http://www.aosd.net/wiki/index.php?title-Glossary
4. Kiezales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W.G., An overview of AspectJ, *Proc. of the 15th European Conf. on Object-Oriented Programming (ECOOP'01)*, 2001, pp. 327–353.
5. AspectJ: an aspect-oriented extension to Java. http://www.eclipse.org/aspectj/doc/released/progguide/language.html
6. Introduction to AspectJ. http://eclipse.org/aspectj/doc/released/progguide/starting-aspectj.html
7. ajc: a compiler of Java and its extension AspectJ. http://www.eclipse.org/aspectj/doc/released/devguide/ajc-ref.html
8. Spinczyk, O., Lohmann, D., and Urban, M., AspectC++: an AOP extension for C++, *Software Dev. J.*, 2005, no. 5, pp. 68–76.
9. Safonov, V.O., Aspect.NET—An aspect-oriented programming tool for the development of reliable and safe programs, *Comput. Tools Education*, 2007, no. 5, pp. 3–13.
10. Khoroshilov, A., Mutilin, V., Sheherbina, V., Strikov, O., Vinogradov, S., and Zakharov, V., How to cook an automated system for Linux driver verification. *Proc. of the 2nd Spring Young Researchers' Colloquium on Software Engineering*, St. Petersburg, 2008, vol. 2, pp. 10–14.
11. Khoroshilov, A., Mutilin, V., Petrenko, A., and Zakharov, V., Establishing Linux driver verification process. *Proc. of the Int. Andrei Ershov Memorial Conf. "Perspectives of Systems Informatics", Novosibirsk, 2009, Lect. Notes Comput. Sci.*, 2010, vol. 5947, pp. 165–176.
12. Mutilin, V.S., Novikov, E.M., Strakh, A.V., Khoroshilov, A.V., and Shved, P.E., Linux Driver Verification architecture, *Tr. Inst. Syst. Prog.*, 2011, vol. 20, pp. 163–187.
13. Khoroshilov, A., Mutilin, V., Novikov, E., Shved, P., and Strakh, A. Towards an open framework for C verification tools benchmarking, *Proc. of the Int. Andrei Ershov Memorial Conf. "Perspectives of Systems Informatics",* 2011, pp. 82–91.
14. Gong, M., Zhang, C., and Jacobsen, H.-A., AspeCt-oriented C. Technology showcase, *Proc. of CASCON 2007, Markahm, Ontario*, 2007.
15. Gong, W. and Jacobsen, H.-A., AspeCt-oriented C language specification version 0.8, Univ. of Toronto, 2008.
16. Seyster, J., Dixit, K., Huang, X., Grosu, R., Havelund, K., Smolka, S.A., Stoller, S.D., and Zadok, E., Aspect-oriented instrumentation with GCC, *Proc. of the First Int. Conf. on Runtime Verification, Malta, 2010*, *Lect. Notes Comput. Sci.*, vol. 6418, pp. 405–420.
17. GCC plugins. http://gcc.gnu.org/wiki/plugins
18. Ball, T. and Rajamani S.K., SLIC: a specification language for interface checking (of C), *Tech. Rep. MSR-TR-2001-21*, Microsoft Research, 2002.
19. CIF: an implementation of aspect-oriented programming for C. http://forge.ispras.ru/projects/cif
20. Novikov, E., One approach to aspect-oriented programming implementation for the C programming language, *Proc. of the 5th Spring/Summer Young Researchers' Colloquium on Software Engineering*, Yekaterinburg, 2011, pp. 74–81.
21. GNU compiler collection 4.6.1. http://gcc.gnu.org/onlinedocs/gcc-4.6.1/gcc/
22. Fix of the error in the USB driver of the Gadget class. http://marc.info/?l=linuxusbm=129649764609408
23. Linux operating system kernel. http://kernel.org
24. Beyer, D., Henzinger, T.A., Jhala, R., and Majumdar, R., The software model checker Blast: Applications to software engineering, *Int. J. Software Tools Technol. Transfer*, 2007, vol. 9, no. 5, pp. 505–525.
25. Beyer, D. and Keremoglu, M.E., CPAchecker: a tool for configurable software verification, *Proc. of CAV 2011* Gopalakrishnan, G. and Qfideer, S. (eds.), Heidelberg: Springer, 2011; *Lect. Notes Comput. Sci.*, vol. 6806, pp. 184–190.
26. List of Linux operating system driver errors detected within the LDV project. http://linuxtesting.org/results/ldv

Translated by V. Arutyunyan