Software Engineering
2016; 4(3): 50-58
http://www.sciencepublishinggroup.com/j/se
doi: 10.11648/j.se.20160403.11
ISSN: 2376-8029 (Print); ISSN: 2376-8037 (Online)

An Approach About Monitoring Generic Properties on C Programs Using Aspect-Oriented Programming with ACC (AspeCtC)

Pikeroen Olivier

College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China

Email address:
olivier.pikeroen@gmail.com

To cite this article:
Pikeroen Olivier. An Approach About Monitoring Generic Properties on C Programs Using Aspect-Oriented Programming with ACC (AspeCtC). *Software Engineering*. Vol. 4, No. 3, 2016, pp. 50-58. doi: 10.11648/j.se.20160403.11

Received: April 30, 2016; **Accepted**: June 4, 2016; **Published**: June 8, 2016

Abstract: For systems which deal with serious or dangerous things (for example nuclear power plant), programs are constrained by legislation to be safe in any case, which requires verification process during the execution, in other words "Runtime Verification" (RV). The field of runtime verification has many different names: runtime monitoring, runtime checking, runtime result checking, runtime reflection, monitoring oriented programming, design by contract, runtime analysis, dynamic analysis, trace analysis, fault protection, etc. Aspect-oriented programming (AOP) is a useful paradigm for monitoring programs, even if it was not created for this purpose. Indeed, AOP tries to deal with crosscutting concerns (tangled and scattered codes) by "capturing" them within a new entity called aspect, and runtime verification properties are conceptually transversal to the code they verify, unavoidably resulting in such crosscutting codes. AOP is always used as an extension of an existing language. Hence it is necessary to design an aspect language extending the target language, and to use what is called a weaver, to realize a binding operation between the target program and the aspects. Thus, one can find many aspect-programming extensions (including an aspect language and an aspect weaver) for most programming languages. The first one to be developed was AspectJ, designed for Java, and which has been the best-known reference among aspect-oriented tools until now. We use an AspectJ-like tool, straight inspired from AspectJ but designed for C, called ACC, which is itself an improved version of AspectC. The purpose of this paper is to explore the possibility of implementing and monitoring generic (also called parameterized) verification properties, i.e. which could be used on any target code, with ACC through a basic example. As ACC, contrary to AspectJ, does not provide abstraction for aspects, which would have made generic monitoring an easy task, we tried to simulate abstraction by making use of macros in the aspect code, which boils down to monitor every parameterized property within one macro function. We will see that despite losses of expressiveness without complexification of the monitoring code, the method still allows monitoring generically any property which is already monitorable directly in ACC.

Keywords: Runtime Verification, Monitoring, Aspect-Oriented Programming, Generic Properties, AspectC, ACC

1. Introduction

1.1. Runtime Verification

Today, programs are everywhere, and need to satisfy many constraints, in order to suit the expectations of the people depending on its behavior. Traditionally, it exists two ways for verifying program executions behavior. The first one, called static analysis, applies before execution. There are three types of static analysis verification techniques: model checking, theorem proving, and static code analysis (testing). However, model checking is not adapted to high size systems, theorem proving is a (semi-)manual technique, and testing cannot deal with all the property behaviors. The second one, dynamic analysis, inspects single executions of the system under scrutiny [4]. Runtime verification is a dynamic analysis method, which only focuses on detecting faults, i.e. deviations between the current behavior and the expected behavior [8]. It considers a system to be checked, and a set of properties to be checked against the system execution. An execution is viewed

as a finite (or some say possibly infinite) sequence of program states, or a finite trace (or word, where each state can be viewed as a letter, or a symbol). This allows a formal representation of RV systems. We can define an alphabet as a set of symbols; a language over an alphabet A as a subset of A^*; a property over an alphabet A as a language over A.

A runtime verification process typically consists of the following three stages [4]. During the first stage, a monitor is generated, from a formal property. A monitor for a language on an alphabet is a device which as input takes events, that is to say a list of symbols (a trace) of the alphabet, produced by the running system, and emits a value (verdict) for the last symbol, based on the history of received events. Hence, when monitoring, we would rather see a language as a set of traces over an alphabet A, and a property, more precisely, as a mapping from the set of all traces over A to a set of verdicts. Then, monitoring would consist in defining the language which denotes the set of valid words given by a property (i.e. traces whose property's associated verdict indicates them valid), and checking whether the current word is an element of the language (i.e. finding the traces which emit a valid verdict according to the property). Or in other words, a monitor checks whether an execution meets a correctness property. The second stage is called instrumentation, and consists in preparing (instrumenting) the system to generate relevant events for the monitor. In the third stage, the monitor eventually analyses the execution, thanks to the generated events.

Runtime verification research works on two different aspects. The first one is about checking of traces, as algorithms to check property specifications over given traces. The second one is about generation of traces, as instrumentation.

1.2. Aspect-Oriented Programming

Instrumentation can be achieved using aspect-oriented programming, abbreviated AOP (as it happens in XspeC [6], an extension of AspectC with state machines, inspired by Rmor [5], a runtime verification framework to monitor C programs against state machines). Hence, we briefly introduce the AOP paradigm.

In programming, a concern is what interests stakeholders. When developing a software, programmers ideally wish they were able to separate concerns into modules, and develop them in isolation, one at a time. The problem they meet is that some concerns impact multiple components. These are called cross-cutting concerns. There are two types of cross-cutting codes: tangling, when each component contains the implementation to satisfy different concerns (one module, many concerns); scattering, when codes that realize a particular concern are spread across multiple components (one concern, many modules).

Cross-cutting codes can be found when coding verification (checking program behavior), as well as logging (tracking program behavior), policy enforcement (correcting behavior), security management (preventing attacks), profiling (exploring where a program spends its time), memory management, visualization of program executions, etc.

AOP tries to deal with both tangling and scattering issues. According to Gregor Kiczales, the inventor of the concept of aspect, *"AOP is about capturing cross-cutting concerns"*. Thus, an aspect is a software entity which captures a cross-cutting concern [7]. Given that AOP is always used as an extension of an existing language, it is necessary to use what is called a weaver, to realize a binding operation between the target program and the aspects. Weaving can be done during compilation or execution. To implement aspects, AOP uses join points which are points in the program where aspects can be inserted. Then, a crosscut (or pointcut) is a set of join points, and an advice is a code block implementing an aspect behavior.

These notions are the basis to understand AOP adapted to any language, and thus, for C.

In our case, instrumentation is encapsulated within aspects and integrated into the system through the weaving process, where events are represented by pointcuts.

2. About AspeCtC (ACC)

2.1. AOP Concepts Applied on AOP Tools

As we said previously, aspect-oriented programming (or AOP) is a paradigm aiming at modularizing cross-cutting concerns, with the concept of aspect. Therefore, it brings to programs avoidance of code redundancy, better reusability, maintainability, configurability, etc. Thus, it may apply in many domains such as tracing, synchronization, buffering, security, error handling, constraint checks, etc.

We saw the most basics concepts about AOP (i.e. join points, pointcuts, advices). Now, we should introduce further advanced ones, which are available on almost all current AOP tools. Advices generally support different types as code advice (*before*, *after*, *around*), introductions, or aspect order. The keywords *before*, *after* and *around* specify wherever the code advice should be executed with respect to the matched join point (respectively before, after, or around, i.e. some code before and some code after, optionally separated by the call to a function usually named *proceed*, which makes the matched join point execute). Introductions allow to extend the behavior of the program by adding into it new attributes or methods. Advices may also be used, when several aspects match the same join point, to specify the execution order of these aspects. Join points are declared by a join point description language, of which sentences are called pointcut expressions. Building blocks of pointcut expressions are called match expressions, which can be used with pointcut functions and logical operations. One can generally give pointcuts a name so that one can reuse them elsewhere. Two usual join points (or pointcuts) are call and execution join points, which match respectively a call and an execution of a function.

2.2. Introduction to AspectC and ACC

2.2.1. Overview

AspectC has been the first aspect language for C, written by Gregor Kiczales and Yvonne Coady, which was inspired

by AspectJ. The official website says indeed: *"The initial design of AspectC was taken directly from the non-object oriented subset of AspectJ"*. Unlike the latter, there is no explicit aspect declaration, as in AspectC, an aspect is a file or a list of files which are able to modularize a crosscutting concern, syntactically containing AspectC extensions and C code. Procedure calls (*call* pointcut) and executions (*execution* pointcut) can be used in advices. The dynamic *cflow* pointcut selects all join points which occur in the control flow of another join point. However, variables cannot be accessed in advices and there is no introduction advices either. Access to typed context is possible. Pointcuts *args* and *result* provide deeper matching by allowing access to context information (function arguments and return value), which can even be modified through advice code. AspectC seems unmaintained since 2003 without any official releases.

ACC (AspeCtC or AspeCt-oriented C), started to be developed in 2006, has been the most regularly released aspect language for C until 2008, and designed by the people behind TinyC^2 [13], under the direction of Hans-Arno Jacobsen. ACC transforms source code and offers its own internal compiler framework for parsing C. The compiler consists in translating code written in AspeCt-oriented C into ANSI C code. This code can be compiled by any ANSI C compliant compiler, e.g. GCC. ACC is actually an updated version of the original AspectC, but it brings a lot of improvements. For example, like in AspectJ, it adds a *JoinPoint* structure (provided in ACC library) to provide join point context or access to function arguments (through the pointer *this*). Arguments passed to *around* advices cannot be modified directly via the captured context. But instead, one should use the *JoinPoint* structure which contains pointers to the matched function's arguments. ACC provides a number of new pointcuts like *callp* (which matches function pointer calls), *infunc*, *infile* (which respectively match code inside a function, a file) and there are *get*/*set* join points which allow access to global variables (only). Introductions are now available, i.e. code can be introduced into structures and unions. Furthermore, ACC provides exception handling aspects, which use a special pointcut called *preturn*, which returns a user-provided error value *errorvalue* as return value. The last release of ACC was in 2010, and the project seems abandoned since then, with several remaining lacks. It is a closed system, in the sense that one cannot augment it with new pointcuts or access the internal structure of a C program in order to perform static analysis. Maintenance of ACC by its developers is not active and as its core component is closed it is not so easy to extend it [10] [11].

2.2.2. Language Design

The AspectC language grammar is quite equivalent to its "father" AspectJ. However, as they are respectively extended languages of C and Java, they unavoidably slightly differ from each other. We present here the main AOP concepts designed in ACC.

As we said above, contrary to AspectJ, in AspectC and its improved version ACC, aspects are not considered as classes (concept which does not exist in C), but as files, with the extension. *acc*. Thus, an aspect file itself represents an aspect.

Pointcuts in ACC are defined like in AspectJ, as shown in Figure 1. Match expressions allow the use of "wildcards" to provide smarter join point matching. The wildcard "$" matches any length of continuous strings, and "..." matches any length item list.

```
pointcut pointcutName(): pointcut expression;
```

Figure 1. *ACC: pointcut definition.*

Advice generic definition is given in Figure 2. The return type (*typeName*) of the advice must be specified only for *around* advices, otherwise it must not. And *typeName* must be the same type as the return type of the matched join point. The different advice types (*adviceType*) are *before*, *after* and *around*. The advice acts like a function, and thus may have arguments ("*parameter list*"), written like in a standard C function (the argument type followed by the argument name). The "*pointcuts*" represent any valid logical combination of pointcut expressions (using the logical operators "*&&*", "||", "*!*"). Finally, one may write any valid ACC code in the advice body (reminding that any valid C code is also valid ACC code). *around* advices should return a value, like a C function (with the keyword *return*), whose type is identical to the advice return type *typeName*. The value returned by the advice will be the value returned by the matched join point. Other kinds of advice (introductions, exception handling) have their own (different) definition grammar.

```
typeName adviceType(parameter list) : pointcuts {
    advice code
}
```

Figure 2. *ACC: advice definition.*

2.3. Conclusion

ACC is an upgrade version of AspectC, which is itself, as AspectC++, inspired from AspectJ. Thus, it provides lots of useful advanced pointcuts like *cflow* to advise in the control flow of a join point, but also created its own, like *callp* to advise the call to a pointer function. It also provides a join point structure (or API) with members to store the join point context information structure through the AspectJ-/OOP-like pointer *this*. However, it is unfortunate that the project has been abandoned without being totally achieved (some "todo"s remain in the source code), letting the compiler with several bugs and leaving us dissatisfied about few functionalities (e.g. bugs using the exception handling feature and the *JoinPoint* structure members, and lacks of expressiveness of match expressions' type-matching using wildcards).

3. Towards Generic Runtime Verification with ACC

3.1. The Reasons for Choosing ACC

At this point, we saw the concepts of both runtime verification and aspect-oriented programming. Runtime verification is a dynamic analysis approach to detect faults during runtime. Aspect-oriented programming is a paradigm which captures cross-cutting concerns, which unavoidably exist in many codes, particularly when programming verification. Thus, aspect-oriented tools are a godsend for runtime verification. However, AOP applications in the field of runtime verification are surprisingly rare for C language, maybe because no AOP tools for C has ever emerged as a reference, as they all suffer some lacks or have been unmaintained.

The aspect-oriented tool for C we chose is ACC for several reasons. First of all, one should know that the most famous (and used) tool for aspect-oriented programming is AspectJ. But the scope of our research is limited to C language only, unlike AspectJ which has been implemented for Java. Thus, one could focus on AspectC++ which is totally inspired of AspectJ and at first glance is able to weave in C. However, even though it is possible to weave in C code with AspectC++ [12], it is unfortunately not plainly possible. Indeed the documentation says: *"Currently ac++ generates C++ code, which cannot be compiled by a C compiler"*. Although it also adds: *"As for many hardware platforms in the embedded domain no C++ compiler is available we are actively looking for a solution."*, which could give us hope for a plain C code support in aspect weaving and compiling in the near future, and thus still makes AspectC++ potentially interesting for us, actually, the developers do not seem to make it a priority, as it has been several years that nothing came out about this support. Thus, the natural way was to make do with AspectC given that it is also an AspectJ-like aspect-oriented extending language, admittedly less documented, used, and released, but plainly designed for C. Because of certain bugs and lacks regularly discovered when working with ACC, we set about searching for other tools, and found the documentation of Aspicere which raised our interest (particularly because it provides generic pointcuts, as well as most of existing AOP features, and also because its design is based on the "coevolution" of source code and the build system [1], which would make the aspect-oriented tool, contrary to ACC, more adapted to the evolution of the build system). However, unfortunately, many attempts failed to install the latter because of absence of updates for many years. We also found several other tools for AOP (Arachne [3], C4, WeaveC, Xweaver), but which regarding the analysis made by Bram Adams [1], do not provide more aspect-oriented features than ACC. Furthermore, some of them have not been released for many years (Arachne), are untraceable (C4), or poorly documented (WeaveC). For these reasons, we chose working with ACC.

We should notice, however, that in the course of our research, a new compiler, called Movec [14] has been developed, as it happens, to fill the lacks of ACC about allowing generic monitoring. Movec is a monitor-oriented compiler and language inspired from JavaMOP (in Java), which aims at providing parametric runtime verification for C programs.

3.2. The Properties to Check Under Scrutiny Applied to a File Example

The example we chose here is inspired from the lectures of Klaus Havelund about monitoring in AspectJ. Consider a file, which one can open and close. We will check the following two properties at runtime:

- *Response*: A file should eventually be closed once opened.
- *Request*: A file cannot be closed unless it has been opened.

The two properties have something in common: they are of temporal dimension. Indeed, they implicitly imply the notions of past and future. The first one gives a specification about the future of an element (here the file, which shall be closed) under a present condition (when one opens it). This property is called a response property: whenever the method Q is called on an element e, eventually the method R will be called on e. The second property gives a specification about the past of an element (here the file, which should have been opened) under a present condition (when one tries to close it). This property is called a request property: whenever the method Q is called on an element e, in the past the method P must have been called on e.

We use the well-known standard functions of *stdio.h*: *fopen* and *fclose*. Consider the code given in Figure 3. The properties verification amounts to only advise the functions *fopen* and *fclose*. Closing *file2* should lead to a "request" error because it has not been opened before. And a "response" error should be found before the end of the execution because *file2* has not been closed ever. For later reuse, we give in Figure 4 the relevant pointcuts.

```
int main() {
    FILE *file1=NULL, *file2=NULL;

    file1=fopen("file1.txt", "r");
    fclose(file2); // error (request property)

    file2=fopen("file2.txt", "r");
    fclose(file1);
    return 0; // error (response property)
}
```

Figure 3. *Monitoring a file example: the source code.*

```
pointcut open(): call(FILE* fopen(...));
pointcut close(): call(int fclose(...));
pointcut prog exec(): execution($ main(...));
```

Figure 4. *Monitoring a file example: the needed pointcuts.*

3.3. Analysis of the Elements Needed in the Aspect Code

3.3.1. Storing the Objects Targeted by the Properties

First of all, we need to consider the elements we need. When an error occurs, e.g. a file which has not been closed before the end of the execution (response property), one might want to know which file it is, i.e. the file name. Because the

type *FILE* does not provide such information method to the programmer, we have to store the name for this file when it is opened (with *fopen*), because it is when the name is explicitly specified. This can be done easily with the ACC pointer *this*. What kind of element would be the best to store this value? Actually, we need to consider the case when many files are opened, so we need an array. This array should be declared global in the aspect file so that every advice could access it. And because one cannot directly access the name from the file, one should need to store the pair (file, name). But the file and its name are of different type, so one should use a structure. Therefore, we need an array of a structure-type elements. The advantage of a structure is that one can extend it easily with new members if needed.

3.3.2. Considering Dealing with Objects of Any Type

As we deal with arrays, we might want to have "array handling" methods (e.g. *addFile*, *findFile*, *removeFile*). But if we consider now that we would like to advise functions which affect other elements, we would need to rewrite the array handling functions. It would be a waste of time (and space) because the only thing changing would be the type of the parsed elements. One could consider the idea of using generic parsing array functions. Then one would think about using the only generic type of C language, i.e. *void**, the pointer to *void*. One can find on the web several implementations of such generic array handling methods.

3.3.3. Optimizing the Runtime Overhead

However, one generally researched purpose (or constraint) when applying runtime verification methods to a target program is to prevent any runtime overhead, or at least to minimize it at most. Well, linear array parsing is *O(n)* where *n* is the array length, which is not recommended for its speed when one work on many elements. As in our case we use aspects for monitoring, i.e. checking code at runtime, and thus may not know in advance the number of elements we will work on, one could think about a faster array parsing method. It exists such a one, of course, and it is called hashing. Unfortunately, the latter does not exist by default in C, therefore one should look for independently provided ones. On the other hand, they are quite numerous, even though significant efficiency differences exist between them. This is not the purpose of this paper to discuss about the best existing hash table implementation for C, but we may still give some names: TommyDS, khash, uthash, Concurrency Kit, googledensehash, etc. The one we chose is TommyDS, because it is well documented and provides clear and numerous tests with the corresponding source code, which show that it is one of the fastest existing hash table (according to its own tests yet). To use the *tommy_hashlin* (TommyDS' recommended hash table) methods, one must first of all download the tommyds package. One just need to copy the following files (being aware of their extension when compiling with ACC[1]) into the directory containing the source and aspect files: *tommychain.h*, *tommyhash.h*, *tommyhash.c*, *tommyhashlin.h*, *tommyhashlin.c*, *tommylist.h*, *tommylist.c*, *tommytypes.h*. Then, one must include the file *tommyhashlin.h* via an *#include* directive in the aspect files using the TommyDS methods. Furthermore, to store the elements in the hash table, one must first define them as a member of a structure which also contains a member of type *tommy_node*.

3.3.4. Dealing with Faults

Eventually, we need to provide some code to deal with occurring faults. If we have a glance at the request property: "a file cannot be closed unless it has been opened", we have to think about what should happen whenever the program tries to close a file which is not open. As the file is not open, the easiest way would be to totally skip the execution of the closing function (*fclose*). We can do that by advising the latter function with an *around* advice without the call of *proceed*. And the advice should return a value instead of the function, which would likely correspond to a default error value. Now, looking at the response property, we need to decide what should happen, before the end of the program execution, to the open files which have not been closed yet. One could think of merely displaying an error message and let the developer of the target code solve the problem, another might consider closing the files as necessary and hence close them through the aspect code. In both cases, one should provide a dedicated function to deal with all those files.

```
...
#include "tommyhashlin.h"

typedef struct FileUtil {
    FILE* file;
    char* name;
} FileUtil;

typedef struct TommyObject {
    tommy_node node;
    FileUtil* fileUtil;
} TommyObject;

void responseFile(const TommyObject* obj) {
    printf("close file : name : %s\n", obj->fileUtil->name);
    fclose(obj->fileUtil->file);
}

int compareFile(const void* arg, const void* obj) {
    return arg != ((const TommyObject*) obj)->fileUtil->file;
}

tommy_hashlin_files;

before(): prog_exec() {
    tommy_hashlin_init(&files);
}

after(FILE* file): open() && result(file) {
    ... // store the opened files in the hashtable
}

int around(FILE* file): close() && args(file) {
    ... // check the request property
}

after(): prog_exec() {
    ... // check the response property
}

after(): prog_exec() {
    tommy_hashlin_done(&files);
}
```

Figure 5. *Monitoring a file example: the aspect code structure.*

[1] Source files must be compiled by ACC with extension *.mcc* by default instead of *.c*. This can be changed in the parameters of the *acc* command.

3.3.5. The Final Code

The aspect file (Figure 5) contains the structure *FileUtil* to access the files information, and a structure *TommyObject* to access the hash table data. A function *responseFile* is provided to do both printing a message and close the file in order to deal with faults in the response property, as well as a function *compareFile* needed by TommyDS to search within the hash table. The latter, which has name *files*, stores objects of type *FileUtil*, and is created before the execution of the program and destroyed after it. There are three advices. The first one stores the opened files in the hash table at the time they are opened. The second one advises the closing of the files to check the request property, i.e. whether the matched file is currently open (is in the hash table, in which case it should be removed from it), or not, in which case an error will occur and the advice code should deal with it. The last advice matches the end of the program execution (but before that the hash table is deleted!), to check the response property, i.e. whether some files have still not been closed (the hash table is not empty), in which case the advice code should deal with it too.

The aspect code for the advices is presented in Figure 6. The "open" advice is not particularly interesting, as it only inserts elements into the hash table, so we do not give it. The "close" advice handles the request property, removes the file from the hash table (*tommy_hashlin_remove*) when the file is being closed but currently open, or skips the execution of *fclose* with the standard error return value *EOF* when the file is not currently open. The last advice (pointcut *prog_exec*) advises the end of the program execution for the response property, and closes the open files which have not been closed yet, using the function *responseFile*.

```
int around(FILE* file): close() && args(file) {
    TommyObject* obj=tommy hashlin remove(&files, compareFile, file,
tommy inthash u64((tommy uint64 t) file));
    if(obj) { // File is currently open
        return proceed();
    }
    else { // File is not currently open
        printf("Request property : Cannot close a not opened file.\n");
        return EOF;
    }
}

after(): prog exec() {
    printf("Response property : Matching answering method not found on %lu
elements -> provided behavior :\n", (uint64 t) tommy hashlin count(&files));
    tommy hashlin foreach(&files, (tommy foreach func*) responseFile);
}
```

Figure 6. *Monitoring a file example: the advices code.*

The output of the executed woven code will eventually be the one given in Figure 7.

```
Request property : Cannot close a not opened file.
Response property : Matching answering method not found on 1 elements -> provided
behavior :
close file : name = file2.txt
```

Figure 7. *Monitoring a file example: the woven and executed code output.*

3.4. Reusing Properties via Macros

3.4.1. Explanation

We saw how to monitor, with a file example, temporal properties, and used generic data structures to be able to monitor any kind of element. These data structures are helpful to prevent from implementing the same data handling methods multiple times for multiple kinds of elements. But now, we would like to be able to advise multiple kinds of join points using the same advice implementation. Unfortunately, unlike AspectJ and AspectC++, ACC does not provide generic (abstract) pointcuts, which could have make that work easy [9]. Two solutions may be considered. The first one would be to directly provide ACC with generic pointcuts. We did not go into this approach in depth. However, as C language has not been designed for providing abstraction, and ACC is totally implemented in C (and ACC language itself), and as we said is a closed system, this solution seems quite hard. Other AOP tools for C used the features of other languages, as the templates of Prolog (in Aspicere [1] [2]), to design their pointcut language, including generic pointcuts. The approach we chose consists in using macros to write advices.

The idea is to simulate generic join points thanks to macro functions. For example, to write a generic pointcut, one might write a macro, like in Figure 8, taking three arguments: one identifier, the second would be a list of parameters which would correspond to the arguments of the pointcuts *args* and/or *result* (in the pointcut expression) for manipulating the join point context, and the third a pointcut expression (using the previous arguments).

```
#define ABSTRACT POINTCUT(IDName, parameterList, PCExpression) \
pointcut concretePC_##IDName(parameterList): PCExpression;
```

Figure 8. *Monitoring using macros: an "abstract pointcut".*

Then, for example, we might want to have two pointcuts: one which matches the call of the function *fopen*, the other matching calls to the function *fclose*. Thus, we would write twice the macro *ABSTRACT_POINTCUT*, each with a different pointcut expression (see Figure 9).

```
ABSTRACT POINTCUT(open, FILE* file, call(FILE* fopen(...)) && result(file));
ABSTRACT_POINTCUT(close, FILE* file, call(int fclose(...)) && args(file));
```

Figure 9. *Monitoring using macros: creating two different pointuts with the same macro.*

The macro functions will be replaced during preprocessing by ACC's valid language pointcut definitions (see Figure 10).

At this point, using macros is obviously useless, but it may become interesting when writing advices, particularly in the case where the latter would verify the same property (the advice code would have the same behavior) on elements which can be of multiple kinds, handled by some generic methods.

```
pointcut concretePC_open(FILE* file): call(FILE* fopen(...) && result(file));
pointcut concretePC_close(FILE* file): call(int fclose(...) && args(file));
```

Figure 10. *Monitoring using macros: the "abstract pointcut" after preprocessing.*

```
REQUEST(open to close, call(FILE* fopen(...)) && result(file), call(int
fclose(...)) && args(file), FILE*, file, int, EOF);

RESPONSE(close when open, call(FILE* fopen(...)) && result(file), call(int
fclose(...)) && args(file), after(): execution ($ main(...)), FILE*, file,
fclose);

RESPONSE(free when malloc, call(void* malloc(size t)) && result(object) &&
infunc(main), call(void free(void*)) && args(object) && infunc(main), after():
execution ($ main(...)), void*, object, free);
```

Figure 11. *Applying the REQUEST and RESPONSE macros on both the previous file example and memory allocations.*

3.4.2. Back to the Response/Request Properties

Using macros becomes interesting when we choose to apply them to both the response and request properties studied above. As we already explained the implementation of these properties and generic aspect code using macros, combining both should not be a problem. As a result, we have two macro functions, *REQUEST* and *RESPONSE*. The *REQUEST* macro takes 7 arguments:

- *IDName*: an identifier for the property and its items (methods, pointcuts, structures, etc.)
- *requestedJP*: the pointcut expression which matches the past event
- *requestingJP*: the pointcut expression which matches the present event
- *objType*: the type of the object on which the property applies, it must be identical to the type of the variable given in the pointcut *args* or *result*
- *object*: the name of the object on which the property applies, it can be any name, but must be identical to the name of the variable given in the pointcut *args* or *result* pointcuts
- *errRetType*: when a fault is detected, the type of the value which must be returned by the matched function
- *errRetVal*: when a fault is detected, the value which must be returned by the matched function

The *RESPONSE* macro takes 7 arguments:

- *IDName*: an identifier for the property and its items (methods, pointcuts, structures, etc.)
- *askingJP*: the pointcut expression which matches the present event
- *asnweringJP*: the pointcut expression which matches the future event
- *endEvent*: event matching the end of the execution (must contain the whole advice definition, without the advice code)
- *objType*: the type of the object on which the property applies, it must be identical to the type of the variable given in the pointcut *args* or *result*
- *object*: the name of the object on which the property applies, it can be any name, but must be identical to the name of the variable given in the pointcut *args* or *result* pointcuts

- *errFunc*: when a fault is detected, function that will be called on each object which led to the fault

Not only we can use these macros to verify request and response properties on the previous file example, but also on other methods, for example *malloc*: we can check that a call to *malloc* on an object eventually leads to a call to *free* on the same object (see Figure 11). Therefore, monitoring the response and request properties can be done now through only one (macro) function per property. One may also now implement new macros for verifying other properties (of any kind, supposing that it is already possible to monitor them without macros).

3.4.3. Weak Points

With these macros, information about the object cannot be added through a structure (e.g. the name of the file), as the structure should be provided by the user and hence its attributes are not known by the macro which therefore cannot easily deal with them generically.

One can also criticize the fact that here we just allow advices with no more no less than one argument. We cannot indeed have e.g. *args(var1, var2)* and act on both variables *var1* and *var2* in the advice code.

More generally, using macros to implement advices limits the level of expressiveness of the aspect code: the number of arguments one wants to deal with, access to context information, user-provided behavior, syntax constraints in the macros arguments which can easily lead to errors, etc. Another issue inherent to macros is also the debugging difficulty when errors occur. Of course, all these problems might be solvable with deep and accurate implementation of these macros, but the code would become at the same time proportionally complicated, as well as the use of the macros, where the number of their arguments would increase accordingly.

A further approach, thus, could be to encapsulate these macros into a graphical interface where the user would only need to choose, as "options" to select, the arguments of the macros, regarding his needs.

4. Conclusion

As research about runtime verification has been growing more and more with interest and ideas, aspect-oriented programming has been one of the most popular methods used for monitoring, because its paradigm directly handles instrumentation. However, until today, unlike AspectJ, no AOP tools for C has been appreciated as a reference, and most provided tools for this language are either outdated or contain some lacks in one part or another. Thus, we chose ACC, as it provides lots of features and its language is directly inspired from AspectJ. Unfortunately, it does not allow writing generic advices, which could have helped to monitor generic properties. Hence, we tried an approach, using macros, combined with hash table methods for C, to simulate the effect of generic advising. This approach succeeded in using one aspect implementation to monitor one property on any kind of object, but suffered from some consequent lacks in expressiveness and control of the monitoring code. Still, for some purposes, the method can be interesting, as it allows any kind of properties which can be monitored with ACC on a single type of element to be monitored generically (on any type), as abstract aspects would do.

References

[1] Bram Adams, "Co-evolution of Source Code and the Build System: Impact on the Introduction of AOSD in Legacy Systems", PhD thesis in Ghent University, Department of Information Technology, (March 2008).

[2] Bram Adams and Tom Tourwe, "Aspect orientation for C: express yourself", in 3rd Software-Engineering Properties of Languages and Aspect Technologies Workshop (SPLAT), AOSD (2005).

[3] Remi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc Segura-Devillechaise, and Mario Sudholt, "An expressive aspect language for system applications with Arachne", in Transactions on Aspect-Oriented Software Development I, Lecture Notes on Computer Science, vol. 3880, pp 174-213 (2006).

[4] Ylies Falcone, Klaus Havelund, and Giles Reger, "A tutorial on runtime verification", in Engineering Dependable Software Systems, vol. 34, pp 141-175 (2013).

[5] Klaus Havelund, "Runtime verification of C programs", in Proceedings of the 20th IFIP TC 6/WG 6.1 International Conference on Testing of Communicating Systems (TestCom 2008) and the 8th International Workshop on Formal Approaches to Testing of Software (FATES 2008), Lecture Notes in Computer Science, vol. 5047, pp 7-22 (2008).

[6] Klaus Havelund and Eric Van Wyk, "Aspect-oriented monitoring of C programs", in the 3rd Domain-Specific Aspect Languages Workshop (2008).

[7] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin, "Aspect-oriented programming", in Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 1997), Lecture Notes in Computer Science, vol. 1241, pp 220-242 (1997).

[8] Martin Leucker and Christian Schallhart, "A brief account of runtime verification", in The Journal of Logic and Algebraic Programming, vol. 78, pp 293-303 (2009).

[9] Daniel Lohmann, Georg Blaschke, and Olaf Spinczyk, "Generic advice: On the combination of AOP with generative programming in AspectC++", in Proceedings of the Third International Conference on Generative Programming and Component Engineering (GPCE 2004), Lecture Notes in Computer Science, vol. 3286, pp 55-74 (2004).

[10] E. M. Novikov, "An approach to implementation of aspect-oriented programming for C", in Programming and Computer Software, vol. 39, pp 194-206 (2013).

[11] Justin Seyster, Ketan Dixit, Xiaowan Huang, Radu Grosu, Klaus Havelund, Scott A. Smolka, Scott D. Stoller, and Erez Zadok, "Inter Aspect: Aspect-oriented instrumentation with GCC", in Formal Methods in System Design, vol. 41, pp 295-320 (December 2012).

[12] Olaf Spinczyk and Daniel Lohmann, “The design and implementation of AspectC++”, in Knowledge-Based Systems, vol. 20, pp 636-651 (2007).

[13] Charles Zhang and Hans-Arno Jacobsen, “TinyC^2: Towards building a dynamic weaving aspect language for C”, in Proceedings of the 2nd AOSD Workshop on Foundations of Aspect-Oriented Languages (March 2003).

[14] Zhe Chen, Zhemin Wang, Yunlong Zhu, Hongwei Xi, and Zhibin Yang, “Parametric runtime verification of C programs”, in Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2016), Lecture Notess in Computer Science, vol. 9636, pp 299-315 (2016).