27 12

SPLAT '05! Software-engineering Properties of Languages and Aspect Technologies

A one-day workshop to be held in conjunction with the
Fourth International Conference on
Aspect-Oriented Software Development (AOSD 2005),
March 14–18, 2005, Chicago, Illinois, USA
http://aosd.net/conference

Aspect Orientation for C: Express yourself

Position paper

Bram Adams
INTEC, Ghent University
Sint-Pietersnieuwstraat 41
9000 Ghent, Belgium
Bram.Adams@ugent.be

Tom Tourwé
Centrum voor Wiskunde en Informatica
P.O. Box 94079
1090 GB Amsterdam, Netherlands
Tom.Tourwe@cwi.nl

ABSTRACT
In this paper, we propose Aspicere, our effort to bring aspect-oriented software development to the C programming language. We focus primarily on the pointcut language offered by Aspicere, as it differs significantly from other aspect languages, for C as well as for Java. We illustrate the need for a more complex pointcut language, by means of a simple and prototypical concern, present in a real-world application, that can not be captured adequately with current-day aspect technologies.

Keywords
Aspect-Oriented Programming, legacy languages, C, logic programming.

1. INTRODUCTION
The current success of aspect-oriented software development (AOSD) is largely due to the AspectJ programming language. A simple proof of this are the many emails with questions about *AspectJ*-features sent to the *AOSD*-mailing list, as people keep on mixing the two up. Why aspect-oriented programming (AOP) in Java? Researchers tend to experiment and build on the current state-of-the-art, in this case Java or, more generally, OO-languages. It's only when the industry commits itself to a new technology, that these efforts propagate into older, established technologies. That's why only recently AOP starts to show up in non-OO languages like C, Cobol, Lisp, ...

In this paper, we will talk about an effort, called Aspicere, to bring AOP to the realm of C. Aspicere started as a spin-off from Cobble [10], an AOP-language for Cobol. Aspicere was the perfect opportunity to leverage the designed framework for Cobble in the context of another legacy language, C. An initial prototype of the tool [13] featured a (limited) pointcut language inspired by AspectC [4].

AOSD '05 Chicago, Illinois USA

Our goal in this paper is two-fold. First, we will show that these limited pointcut languages, as defined by current aspect languages such as AspectJ or AspectC, are incapable of capturing the pointcuts for some simple and prototypical aspects. Second, we present an alternative pointcut language, based on Prolog, that allows us to express a richer and more expressive set of pointcuts.

The next section introduces the concern that we will use as a running example through the paper. Section 3 shows that current pointcut languages are incapable of capturing the concern adequately, while Section 4 describes Aspicere's alternative pointcut language. Section 5 discusses related work, Section 6 presents future work, and we conclude in Section 7.

2. PARAMETER CHECKING CONCERN
In [2] and [3], Bruntink et al. present their experiences in migrating crosscutting concerns from embedded C code to aspects. Although it is not the primary focus of their papers, the authors discuss the difficulties they face when implementing a seemingly simple concern (parameter checking) in current-day, general-purpose aspect languages. We will use their example in this paper as well, as we feel it is both interesting and appealing: although most people would consider it a typical aspect, it appears to be difficult to capture adequately with current-day technologies.

The requirement for the concern stated in [3] is that each parameter of type pointer that is defined by a non-static function should be checked before its value is used. Parameter checks ensure that a `NULL` value is never dereferenced, which would lead to an error. Typically, such checks are placed at the beginning of a function. The implementation of a check consists of a simple `if` test, whose `then` clause assigns an error value to a special-purpose error variable, and logs that value in a global log file. See [3] for more details. Clearly, this concern is crosscutting and leads to a large amount of duplication.

In sections 3 and 4 we'll look at two aspect implementations of this crosscutting concern. The first one will be described using AspectC, the other one in a more expressive language, i.e. Aspicere.

3. THE ASPECTC WAY
3.1 Introduction to AspectC
In [4], the authors present AspectC, an AOP-language for C, which is conceptually a non-OO version of AspectJ [9].

It resembles AspectJ, except that:

- C functions are treated as static functions in Java;
- a limited joinpoint model is defined (only method calls);
- all aspects are singletons;
- no introduction facilities exist

Nevertheless, AspectC proved itself capable of modularising some path-specific customizations in the FreeBSD kernel. When looking closer at this crosscutting concern, we see that it is a very specific one. After checking out the original FreeBSD kernel code thoroughly (not a trivial task), the precise locations along the targeted paths first were pinpointed. The different aspects were then explicitly expressed in terms of these points in the source code. AspectC is totally capable of this highly specialized task.

Like AspectJ itself, AspectC heavily relies on name based matching of method calls, enriched by some limited incarnation of regular expressions. We will show in the next subsection that this is insufficient to capture a simple aspect adequately, even though this aspect is considered to be a prototypical aspect by many people.

3.2 Parameter checking concern in AspectC

When defining pointcuts for the parameter checking concern, the authors of [2] took considerable care such that:

- code duplication present in the original source code is reduced to a minimum in the aspect code;
- understandability of the (base and aspect) code is improved, or in other words, the intention behind the pointcuts is as clear as possible;
- the pointcuts are as robust as possible with respect to evolution, of the base code as well as of the aspect itself.

A generic parameter checking aspect is thus preferred, but is hard, if not impossible to define in AspectC. It is difficult to capture all functions that need parameter checking in a single pointcut, or even in a small number of pointcuts, because the signature of the functions differ. Some functions define only two parameters, others define three, and still others define five or six. Even if functions define the same number of parameters, these parameters are of different types, most of the time, and not all parameters are of pointer type. AspectC does not allow us to define a pointcut that generalises these differences.

A naive way for implementing a parameter checking aspect would be to define a pointcut for each non-static function separately, as follows:

```
aspect parameterCheckingAspect {
  ...
  pointcut pc1(queue *queue, void *data) :
     args(queue, data),
     execution(* queue_add(queue *queue, void *data));
  pointcut pc2(queue *queue, void **data) :
     args(queue, data),
     execution(* queue_pop(queue *queue, void **data));
  ...
  before(queue *queue, void *data) : pc1(queue,data) {
    if(queue == (queue *) NULL) {
      LOG(PARAMETER_ERROR);
    }
   if(data == (void *) NULL) {
     LOG(PARAMETER_ERROR);
  }}
  before(queue *queue, void **data) : pc2(queue,data) {
    if(queue == (queue *) NULL) {
      LOG(PARAMETER_ERROR);
    }
    if(data == (void *) NULL) {
      LOG(PARAMETER_ERROR);
    }
    if(*data != (void **) NULL) {
      LOG(PARAMETER_ERROR);
  }}
  ...
}
```

These pointcuts each pick out the execution of a particular function, by means of the `execution` pointcut, and expose all its parameters, by using the `args` join point. Clearly, such pointcuts will not improve the source code quality. First of all, the aspect is tightly coupled to the base code, because it hard codes the function names. If functions are added or removed, or the signature of existing functions is changed, the aspect is no longer correct. Second, since advice code needs to be specified for each non-static function, it still contains a lot of duplication that can not be factored out easily. Last, the understandability of the aspect can still be improved upon, because it is not explicitly apparent now that all parameters of the same type should perform the same check.

A more advanced way of defining a parameter checking aspect is to exploit the fact that parameters of the same type implement the same check. The aspect can thus pick out all non-static functions that define a parameter of a particular type in a single pointcut:

```
aspect parameterCheckingAspect {
  ...
  pointcut pc1(queue *queue) :
     args(queue) &&
     (execution(* queue_add(..)) ||
     execution(* queue_pop(..)));
  pointcut pc2(void *data) :
     args(data) &&
     (execution(* queue_add(..)));
  pointcut pc2(void **data) :
     args(data) &&
     (execution(* queue_pop(..)));

  before(queue *queue) : pc1(queue) {
    if(queue == (queue *) NULL) {
      LOG(PARAMETER_ERROR);
  }}
  before(void *data) : pc2(data) {
    if(data == (void *) NULL) {
      LOG(PARAMETER_ERROR);
  }}
  before(void **data) : pc3(data) {
    if(*data != (void *) NULL) {
      LOG(PARAMETER_ERROR);
  }}
  ...
}
```

In this way, each parameter type has its own pointcut and associated advice code. As such, advice code for a particular type of parameter is specified only once and reused. This solution is still not satisfactory, however. For different types of parameters, the advice code can still not be reused, since it differs slightly in the type cast that is being used. As a result, there is still a certain amount of code duplication. Additionally, the pointcuts are still a simple enumeration of functions, which tightly couples the aspect to the base code, and hampers evolvability of that base code.

4. ASPICERE

4.1 Rationale

Since AspectC's pointcut language is not expressive enough to capture the parameter checking concern in a satisfactory way, the authors in [3] developed a domain-specific language (DSL) for parameter checking. Although they show that their solution greatly improves the source code quality, it remains an ad-hoc solution. The DSL is only able to capture the parameter checking concern, so other concerns need other DSLs, and the construction of such DSLs is not at all trivial.

In order to express aspects such as the parameter checking aspect, a pointcut language should be based on more sophisticated mechanisms than mere name matching. In [8], the following language characteristics were distilled, based on pattern matching on the structure of the base program, in order to allow robust pointcut definitions:

- *Prolog-like unification* to make pattern matching and variable binding available in a very clean way.
- *A range of predicates to select joinpoints* with the right properties, and some general-purpose predicates expressing conditions on the properties themselves. Examples of the latter kind include standard Prolog predicates for list handling, mathematical operations, string manipulation, ...
- *the ability to access joinpoints shadows* and to express certain conditions on them. This effectively allows navigating through the (static) structure of the base program.
- *Parameterisable pointcut definitions* to allow reuse.
- *Recursion* to render the pointcut language computationally complete.

AspectJ's or AspectC's variable binding mechanism and regular expression matching are definitely weaker than true unification. Furthermore, the existing pointcut designators don't allow reusing bound variables. there are no static general-purpose predicates, only dynamic `if`-tests. Reuse of custom pointcut definitions is allowed, but recursion lacks.

The family of logic programming languages on the other hand, satisfies all of these proposed features. Using such a language for reasoning about a program is called Logic Meta-Programming (LMP), so in fact pointcut languages are just a special case of LMP. [1] and [7] discuss this in more detail with respect to OO-environments, together with the creation of DSALs (Domain-Specific Aspect Languages). These are DSLs expressed (in)directly in terms of a more primitive aspect language.

4.2 Design decisions

Based on the findings above, we adopted Prolog as the underlying vehicle of our pointcut language. As explained, this makes it much more expressive and also allows to hide complexity behind DSALs. Aspicere will consist of several layers of predicates with varying levels of complexity, which we will now present.

4.2.1 Hyperprimitive layer

Aspicere's weaver operates on the abstract syntax tree of a program, described in XML [10]. The pointcut language should thus be able to pick out specific AST nodes. The lowest layer of our weaver provides the appropriate predicates for doing just that, and shields higher levels from the details involved.

Besides an AST-based representation, our weaver can expose several graph-based representations as well, such as control- and data-flow graphs. This allows us to include a `cflow` predicate, for example. Additionally, the use of function pointers corresponds to some sort of dynamic dispatching, which makes decent pointer analysis a must-have. These specialised representations and analyses are hidden behind a predicate façade.

Both types of predicates are illustrated in Figure 1 (under the "HYPERprimitive layer"-label). `createJPs/2` is an example of the first kind of hyperprimitive predicates: it instantiates joinpoints out of the (XML-)AST of a base program. A prototype of an analysis predicate is given by `controlFlowGraph/1`. This provides access to a graph representation of the control flow in the application at hand.

Although we strive to make the set of joinpoint types offered in Aspicere as complete as possible, we should also provide the possibility of extending it. Power users could define special-purpose joinpoints, such as memory access joinpoints for example. These could have particular requirements on the AST or, more generally, the structure of a base program. This can be easily solved by writing a new predciate, regardless of which layer it's in.

4.2.2 Primitive pointcut layer

The primitive pointcut layer defines the equivalent predicates of the regular AspectC-keywords like `call`, `execution`, `cflow`, ... This way, we make sure that the functionality of AspectC is implied by our new pointcut language.

Additionally, the layer includes "structural" predicates, such as `caller/2`, `enclosingExecution/2`, ... These allow us to navigate through the (static) structure of the base program and to express structural patterns instead of relying solely on name patterns. Aspects become more robust and less dependent on (or: more oblivious to) a base program.

Both sets of predicates should be expressed in terms of the hyperprimitive layer and a collection of more general-purpose Prolog predicates like `member/2`, `nth/3`, `pointer/2`, ... Figure 1 shows some sample implementations for these predicates, with the semantics of most of them intuitively clear. `enclosingExecution/2` gives the "parent" joinpoint of e.g. a call joinpoint. Analogously, the general-purpose predicates are quite easily understood.

4.2.3 Aspect-specific language layer

Using the basic building blocks defined in lower layers, the aspect-specific language layer allows the user to design DSALs tailored to a specific concern, as explained in [1]. In the next section, we will show such a language for the parameter checking concern.

Unlike [1], we currently maintain more or less the lexical advice structure of AspectC. What we mean by this, is that a dedicated advice structure has been added to the C-language (shown in Figure 3), specifying:

- the return type in case of `around`-advice;
- the type of advice (`before`, `around` and `after`);
- the name of the advice followed by exported bound variables;
- the bound joinpoint variable where advice has to be woven;
- a pointcut designator;
- the advice code.

```
/*HYPERprimitive layer*/
createJPs(XPathQuery,JpList):- /*perform query, instantiate joinpoints and cache results*/.
controlFlowGraph(Graph):- /*connection to analysis tool*/.

/*PRIMITIVE layer*/
call(Name,Jp,QualifierList,Params,ReturnType):-
        createJPs(/*...*/,List), /*additional analysis needed*/
        member(Jp,List)
        .
execution(Name,Jp,QualifierList,Params,ReturnType):- /*analogous*/.
enclosingExecution(Jp_enc,Jp):- /*...*/.
dereferencesInBody(Jp,DeReferencedVars):- /*recursively defined or using dedicated analysis*/.
caller(Jp_caller,Jp_callee):-
        execution(Name,Jp_callee,Qualifiers,Params,ReturnType),
        call(Name,Jp_caller,Qualifiers,Params1,ReturnType),
        compatible(Params,Params1) %match types
        .

/*GENERAL-purpose*/
nth(N,Element,List):- /*Find N-th Element of List*/.
member(Element,List):- /*Element is member of List*/.
pointer(Type,BaseType):- /*...*/.
doublePointer(Type,BaseType):- /*...*/.
```

Figure 1: Illustration of the (hyper)primitive layers.

This makes our aspects a hybrid of pure C and a Prolog-based pointcut language. Although this somehow minimizes the transition from C to Aspicere, we lose the natural approach of [1] to tackle advice composition and interaction. This aspect of Aspicere still needs further investigation. The approach used e.g. in [12], using genuine methods instead of special advice constructs, is also a possible alternative.

4.3 Parameter checking concern in Aspicere

In Figure 2, we propose a solution for the parameter checking concern. We only need to define a mini-DSAL, since we build on the (hyper)primitive pointcuts included with the Aspicere system in Figure 1

4.3.1 Pointcut definitions

The algorithm we implemented to calculate the appropriate pointcuts is the following:

Algorithm 1. For each dereferenced pointer argument of a public (i.e. non-static) function g, look if there is an unprotected path leading from the real definition of the pointer variable to its dereference in g. This basically means we have to

1. crawl up from the execution of a function to each call site[1];
2. get the enclosing execution joinpoint jp at the call site;
3.
   - if the pointer is dereferenced in the body of jp and it was passed as an argument to jp, then the current path is clean (no new check needed) as it was jp's responsibility to check;
   - otherwise:
     (a) if the variable was passed as an argument to jp, then we must further investigate this path in step 1;
     (b) otherwise, we've found a previously unchecked path, such that a check is needed in g and the algorithm stops.

[1]Note that we navigate through the control flow in reverse, i.e. from a function to its callers (recursively).

Looking at `ioCheck/2` and `unprotectedPath/3` in Figure 2, we see that this algorithm can be expressed fluently and intuitively in our new pointcut language. `ioCheck/2` will select all non-static functions having a pointer argument, and hands control to `unprotectedPath/3`. This one will do the path checking as explained in the three steps of our algorithm.

The semantics of this crosscutting concern are now explicitly available and don't contain any reference to named functions: the algorithm filters out the right joinpoints solely by looking at the structure of a base program. Of course, this depends on the concern at hand, but the point is that concerns can now be described more robustly and using only static reasoning on the base program. Apart from the concern's checks themselves, there is no additional overhead imposed by residues of the weaving process.

4.3.2 Advice

Figure 3 shows the actual advice, using our `ioCheck/2`-predicate. The corresponding base type of the captured pointer parameter is a bound variable of the `before`-advice and can be used freely in its implementation. So, again we see an improvement on the solution given in 3.2. The advice only has to be described once, using an advice variable as a stand-in for the particular type of the intercepted function argument in each single case. This enhances maintainability. Next to this, as `ioCheck/2` doesn't differentiate between single or double pointers, we can also reuse it for other purposes. To illustrate this, we implemented a hypothetical extra check on double pointer arguments (the second advice shown).

5. RELATED WORK

Contrary to the Java situation, there aren't that many AOP-languages for C, let alone mature ones[2]. Some seem to be abandoned, like AspectC [4] and TinyC[2] [14]. Otherwise, there are also some C++ alternatives like AspectC++[3]

[2]`http://janus.cs.utwente.nl:8000/twiki/bin/view/Composer/AspectC` gives a nice overview and is also the home of ComposeC

[3]`http://www.aspectc.org`

```
/*DSAL*/
ioCheck(Jp,[Pointer,Type]):-
        execution(Name,Jp,Qualifiers,Params,_),
        \+member('static',Qualifiers), %only public functions
        dereferencesInBody(Jp,DeReferencedVars),
        member(Pointer,DeReferencedVars), %explicit dereference of Pointer ...
        nth(Index,[Pointer,Type],Params), %... which is a parameter
        unprotectedPath(Jp,Jp,Index)
        .

unprotectedPath(Jp_orig,Jp_current,Index):-
        caller(Jp_caller,Jp_current),
        call(_,Jp_caller,_,Params,_), %get Params
        nth(Index,Params,[Pointer,Type]), %get alias on caller-side
        dereferencesInBody(Jp_caller,DeReferencedVars),
        enclosingExecution(Jp_execution,Jp_caller),
        \+Jp_orig==Jp_execution, %avoid (in)direct recursion
        execution(_,Jp_execution,Qualifiers,ParamsExec,_), %get ParamsExec
        \+(
           member(Pointer,DeReferencedVars),
           member([Pointer,_],ParamsExec),
           member('static',Qualifiers)
          ), %unprotected dereference
        (
          nth(NextIndex,ParamsExec,Pointer), %is Pointer a parameter?
          !,
          unprotectedPath(Jp_orig,Jp_execution,NextIndex) %yes: crawl up
        ;
          %no parameter: check needed
          ),
        ! %stop as soon as one unprotected path found
        .
```

Figure 2: Encapsulating the parameter checking concern in an aspect written in Aspicere

```
/*User-defined predicates are fed separately to the weaver to avoid tight coupling
  of aspects to predicate definitions.*/

/*Advice for all pointer arguments*/
before inout(BaseType) on(Jp): ioCheck(Jp,[Pointer,Type]) && pointer(Type,BaseType){
        /*check in/out/outpointer-parameters using bound BaseType*/
}

/*Extra advice for double pointers*/
before inoutptr(BaseType) on(Jp): ioCheck(Jp,[Pointer,Type]) && doublePointer(Type,BaseType){
        /*extra check on double pointer-parameters using bound BaseType*/
}
```

Figure 3: The actual checking code encapsulated in two advice definitions.

[11] and C++/CF [6], but those produce and/or are written in C++. ComposeC builds further on Compose*, the composition filter approach applied on Microsoft's .NET-environment, but is just starting up.

This makes Arachne[4] currently the only existing related work. In [5], an expressive aspect language for Arachne is presented based on CLP (Constraint Logic Programming). As such, it has the power of unification, but pointcut designators and advice can't be named for the moment. So, recursion and incremental definition of predicates aren't provided yet. All constructs seem to be hardcoded keywords. The seq-construct would in Aspicere just be a higher-level predicate with a list-argument. Apart from seq, no other (static) structural navigation is included in the language.

6. FUTURE WORK

Currently, we have coupled a Prolog engine to our existing framework[5] and now we are defining and implementing the (hyper)primitive layer. To provide the analysis predicates mentioned in Section 4.2.1, one or more specialized analysis tools in our back-end are indispensable.

Topics like introduction (ITD in AspectJ), interaction of aspects and performance implications of our expressive pointcut language have to be examined in depth, as well as a sufficiently extensible weaving mechanism.

7. CONCLUSIONS

Although AspectJ-like AOP-languages are very popular, they lack certain expressive qualities. We showed this in the context of a realistic case study using Aspicere, an aspect language for C. The use of a logic programming language as evangelized by others in OO-contexts, really pays off for legacy environments like C as well. These logic programming languages allow us to express more robust pointcut definitions in a more semantical way, and likewise promote compehensibility and reuse of aspects. Full implementation of the presented system is on-going work, such that implications on performance are still unclear, as is the usability for truly dynamic crosscutting concerns.

Acknowledgments

We'd like to thank Kris De Schutter and Stijn Van Wonterghem for proofreading. Kris did also give a lot of good advice and helped to write down our algorithm.

8. REFERENCES

[1] J. Brichau, K. Mens, and K. D. Volder. Building composable aspect-specific languages with logic metaprogramming. In *GPCE '02: The ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, pages 110–127. Springer-Verlag, 2002.

[2] M. Bruntink, A. van Deursen, and T. Tourwé. An initial experiment in reverse engineering aspects from existing applications. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 306–307. IEEE Computer Society, 2004.

[3] M. Bruntink, A. van Deursen, and T. Tourwé. Isolating Crosscutting Concerns in System Software. Technical report, Centrum voor Wiskunde en Informatica, 2005.

[4] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using aspectc to improve the modularity of path-specific customization in operating system code. *SIGSOFT Softw. Eng. Notes*, 26(5):88–98, 2001.

[4]http://www.emn.fr/x-info/arachne/

[5]http://allserv.ugent.be/~kdschutt/aspicere/

[5] R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Ségura, and M. Südholt. An expressive aspect language for system applications with Arachne. In *Proceedings of 4th International Conference on Aspect-Oriented Software Development (AOSD'05)*, March 2005. To appear.

[6] M. Glandrup. Extending C++ using the concepts of composition filters. Master's thesis, University of Twente, November 1995.

[7] K. Gybels. Using a logic language to express cross-cutting through dynamic joinpoints. In *Proceedings of the Second German Workshop on Aspect-Oriented Software Development, Technical Report IAI-TR-2002-1*. Universität Bonn, 2002.

[8] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 60–69. ACM Press, 2003.

[9] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[10] R. Lämmel and K. De Schutter. What does aspect-oriented programming mean to Cobol? In *Proceedings of 4th International Conference on Aspect-Oriented Software Development (AOSD'05)*, March 2005. To appear.

[11] D. Lohmann, O. Spinczyk, and A. Gal. Aspect-Oriented Programming with C++ and AspectC++. Tutorial held during the AOSD 2004 conference (Lancaster, UK), March 2004.

[12] H. Rajan and K. Sullivan. Classpects: Unifying aspect- and object-oriented language design. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, St. Louis, Missouri, USA, May 2005. To appear.

[13] S. Van Wonterghem. Aspect-oriëntatie bij procedurele programmeertalen, zoals C. Master's thesis, Ghent University, 2004. In Dutch.

[14] C. Zhang and H.-A. Jacobsen. TinyC^2:towards building a dynamic weaving aspect language for C.