AVOIDING TWO-LEVEL SYSTEMS: USING A TEXTUAL ENVIRONMENT TO ADDRESS CROSS-CUTTING CONCERNS

David Greaves
University of Cambridge, Computer Laboratory
Cambridge, UK

Keywords: Aspect Oriented Programming, Meta-Programming, Textual Environment, Interceptor Function.

Abstract: We believe that, owing to the paucity of textual facilities in contemporary HLLs (high-level languages), large software systems frequently require an additional level of meta-programming to sufficiently address their cross-cutting concerns. A programming team can either implement its system by both writing the main application in a slightly customised language and the corresponding customised compiler for it, or it can use a macro pre-processor to provide the remaining cross-cutting requirements not found in the chosen HLL. With either method, a two-level system arises. This paper argues that textual macro-programming is an important cross-cutting medium, that existing proposals for sets of pre-defined AOP (aspect-oriented programming) join-points are overly constrictive and that a generalised meta-programming facility, based on a *textual environment* should instead be directly embedded in HLLs. The paper presents the semantics of the main additions required in an HLL designed with this feature. We recommend that the textual features must be compiled out as the reference semantics would generally be too inefficient if naively interpreted.

1 INTRODUCTION

Although the term is relatively new, *cross-cutting* requirements have always been found in large software projects, and have been met in various ways, aligned with various cutting/joining axes that we list in the next paragraph. This paper emphasises one particular cutting axis, that of the textual structure of the source file. We first provide a list of the main cutting axes, then list ways in which the textual axis is exploited by the C/C++ pre-processor and speak of alternatives when a pre-processor is not used. Finally we introduce our own textual-environment concept to HLL (high-level language) compilation and evaluate it in terms of how it addresses the facilities otherwise provided by a pre-processor or customised compiler.

By means of introduction, we list the common facilities provided for cross-cutting found in HLLs, starting with the most basic. We define a '*cross-cutting*' aspect of a language to be any mechanism within the language that provides a link from one part of the program to another. It reduces the effective diameter of the program by increasing the dimensionality of the interconnectivity.

Cross cutting axes we identify are:

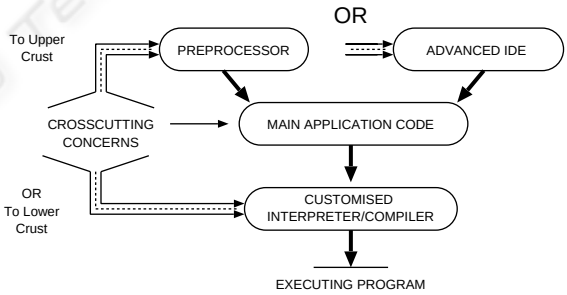

Figure 1: Cross-cutting aspects feeding either into the upper or the lower crust of the sandwich that contains the main application code.

1. **Shared Global Variables.** Shared variables are the most ancient and basic cross-cutting facility, according to our definition thereof. Although obvious, we list them for completeness.
2. **Thread Dynamics.** A thread weaves between subroutines, often held in separate textual files[1]

[1]Strangely, in the hardware description languages VHDL and Verilog, threads may not move between modules. This is one of the most distinguishing features of

Greaves D. (2006).
AVOIDING TWO-LEVEL SYSTEMS: USING A TEXTUAL ENVIRONMENT TO ADDRESS CROSS-CUTTING CONCERNS.
In *Proceedings of the First International Conference on Software and Data Technologies*, pages 71-76
DOI: 10.5220/0001318100710076

3. **Static and Dynamic Reference Environments.** Apart from access to global and dynamically allocated local variables, programing languages with dynamic-free variables, such as Pascal, OCAML and dialects of Algol, provide function closures where a function can make direct reference to values in its closing environment, even when it has been passed off for remote execution as an upcall (see following note).
4. **Computed Branches and Upcalls.** Languages that enable function pointers to be stored in variables, which is all modern languages, enable dynamic dispatch and remote invocation of these functions. Where one component stores an entry in the data-structure of another (often providing a so-called up-call), this is a cross-cutting feature.
5. **Object Static and Dynamic Hierarchy.** The static module inheritance graph of an OO language and the dynamic, actual instantiation of an object mesh at run time are both forms of cross-cutting. We note that Java decided not to provide access to dynamic function variables that are free in a method because it instead provides access to the fields in the surrounding object that may be dynamically allocated almost as easily. Programmers used to SML, OCAML, Haskell and so on often find this a nuisance, but the overhead of providing both a static chain and an object context, with the former not likely to be used by contemporary mainstream programmers, was seen as too great by the Java designers; hence they traded one form of cross-cutting for another.
6. **Long Jumps and Exceptions.** Long jumps and exceptions form another cross-cutting aspect (by our definition) whose usefulness is well proven.
7. **Macro Pre-Processing.** The C pre-processor is used to provide a whole gamet of cross-cutting facilities, which we list separately below. Pre-processing is often deprecated because it is crude, being not type-safe and offering the potential to make a program unreadable. This paper will hold that pre-processing should be replaced with a well-designed, yet simple, '*meta-programming*' facility that is a primary part of any compiled HLL.
8. **Constructors, Access Functions and Overloading.** OO languages allow the user to insert their own code at points where abstract datatypes are created, read or otherwise operated on, by writing constructors and methods bound to overloaded operators. These are useful joinpoints.
9. **Templates and Generics.** Some would argue that C++ templates and other similar HLL generics are artifacts to overcome antiquated, non-parametric, type systems, and would suggest using HM (Milner, 1978) typing instead. However, in whatever way the type system works, the facility to insert additional code in the template libraries provides a form of cross-cutting. Provision of some form of polymorphism, even just through '`void *`' casts, is a required cross-cutting form for any HLL used in a large system.

these languages that enforces a totally different programming style from that used in all (other) software.

We assert that the tacit motivation for AOP (aspect-oriented programming) is that those languages that do not normally use a pre-processor are restricted because the remaining parts of the above list are insufficiently expressive. The sandwich diagram, figure 1, shows that some cross-cutting requirements can be met by the facilities of the HLL *per se*, whereas the remainder are implemented using some form of meta-programming. Two forms are shown. Either a pre-processor is used, by which we imply to include the operations of this nature performed by an IDE (integrated development environment). This is the upper-crust approach. Otherwise, customisation of the compiler/interpreter is needed. This is the lower-crust approach. We believe that only one crust is needed to provide sufficient additional cross-cutting. In our terms, Aspect-J (Kiczales et al., 2001) is a lower-crust approach, where modification of the compiler serves this purpose and the modifications are sufficiently flexible to provide fairly generic meta-programming. One can also argue that the lower crust represents a frequent, major motivation for developing specialised HLL's, such as database languages. In the author's personal experience, where a project team has worked on an ML program some 100,000 lines long, occasional customisations to the ML compiler have proved invaluable when certain cross-cutting requirements have arisen. These all fall into one of the application categories listed below.

The C/C++ preprocessor embodies many individual functions and is defined in terms of multiple passes of the source files. However, a small and well-known core of operations is all that is commonly required. Although most readers of this paper will be well-familiar with the C preprocessor and its typical uses, it is worth listing those specific uses here, so that the reader can consider our assertion, in terms of each use, that AOP has been an attempt to re-provide these facilities when a pre-processor is not routinely used or is deprecated, or the run-time system cannot be customized. The list also serves as the basis for our evaluation criteria in the results section of this paper.

The C/C++ pre-processor is commonly used for the following (cross-cutting) functions. We note that the majority[2] of these functions can be provided using the

[2]When we say the 'majority' we could have put 'all' because the residual language is, of course, Turing complete.

other major cross-cutting forms listed above, such as sending a thread to a method or carrying extra parameters into a function, but that the alternative would be unnecessarily expensive compared with the pre-processing function.

- **Tie-Offs.** A tie-off is the permanent setting of a variable to a constant value. For instance, hard-wiring the name of a directory path or the IP address of a primary server.
- **Conditional Compilation.** From the point of view of this paper, conditional compilation is a tie-off to the guard expression of a conditional construct.
- **Textual Inclusion and Access to Textual Context.** The C pre-processor allows one file to include another and enables the name and line number of the current file to be accessed, using __FILE__ and __LINE__. Newer HLLs, like Python and Java, provide *reflection APIs* that allow much greater information about the textual structure of the program source to be read off. Our textual environment, introduced later, relies on this information being available.
- **Assertions**. Pre-processor assertions generally require both conditional compilation (for rendering a speed-enhanced version) and a long jump or exception mechanism. Provided these two cross-cutting axes exist, conventional assertions can be implemented. If we have textual inclusion and access to textual context, they can be placed in their own library and print the details of their caller.
- **Visualisation, Logging, Coverage Monitoring, etc..** The conditional compilation aspect of the pre-processor enables logging to be turned on and off with global switches, and access to textual context enables C/C++ macros to be conditional, as well as providing vital information to index the recorded data. There are many variations of logging, including visualisation of program resource consumption and code coverage testing. Logging was the example cross-cutting concern addressed in several AOP papers (Kiczales et al., 2001).
- **Memory Allocation and Tracking**. The pre-processor is often used to implement a '`new`' macro in C, overcoming a historical deficiency[3] and the tracking functions just require a cross-cutting logger.
- **Watchpoints.** Using the pre-processor, it is easy to provide breakpointing when a thread reaches a particular line of code; watching for a variable to attain a certain value requires that all writes are ensconced inside unpleasant macro calls; checking for the formation of a particular pattern in a data structure is not at all easy. The customised compiler approach is perhaps the easiest conventional way to watch for the latter, when available; otherwise specialist hardware techniques are used, which are beyond the scope of this paper.
- **Accessor Functions for Opaque Data-Structures.** Inter-procedure call optimisation has replaced the use of *in-lined* macros as the best way to access otherwise opaque data-structures. This use is obsolescent.
- **Inter-language Calls and Miscellaneous Application-Specific Uses.** There are many other applications for the pre-processor, but we assert that the remaining uses can be regarded as application-specific rather than cross-cutting. These include persistence, scheduling, and a whole host of library, operating-system and inter-language calls. The majority of these uses cannot be coded in the original HLL, or certainly require deferred linking, and hence are not cross-cutting aspects of the current application.

A textual (or typographical) technique known to all mathematicians is the distributive law. We assert it can be helpful in programming. For instance, if $g()$ does not produce side effects referenced by $f()$, and vice versa, then

$$f(\texttt{if}\ g()\ \texttt{then}\ A\ \texttt{else}\ B)$$

can be rewritten as

$$\texttt{if}\ g()\ \texttt{then}\ f(A)\ \texttt{else}\ f(B).$$

Our assertion is that the process of '*folding in*' is a required form of cross-cutting in large software systems, where the desire is to apply $f()$ at one point and have it executed at many, textually lower, points. Our approach is to pass items such as $f()$, as well as tie-offs that might effect various functions like $g()$, down through the textual structure of the program to the leaves where they will act.

2 EMBEDDED PRE-PROCESSOR

In this section we define what is essentially a powerful, embedded pre-processor. This is specifically designed to serve the cross-cutting aspects that have been met with a second level, that of macro-processing, as identified in the previous section.

We assume the abstract syntax tree of our HLL is very typical, like the following:

However, it would be tortuous to achieve the more-textual forms even using a reflection API.

[3]Aside: Some might argue it is not a deficiency: they say the fact that C does not always require a memory manager or any form of run-time system at all is one of its main strengths for bare-metal programming.

```
 | eval (s, t) (apply(f, args)) =
    let val (lambda(bv, body), s', t') = eval (s, t) (f)
        fun evalargs(nil, nil, s) = s
        |   evalargs(f::ft, a::at, s) = let val (a', s', _) = eval(s, t) a
            in (f,a')::evalargs(ft, at, s') end
        val s2 = evalargs(bv, args, s')
     val (r, s3, _) = eval (s2, t') body
     in (r, s3, t')
     end

| eval (s, t) (apply(Str f, args)) =
    let val (lambda(bv, body), s', t') = eval (s, t) (Str f)
        fun mv n = "_" ^ (Int.toString n)
        val mvl = ref nil
        fun evalargs(n, nil, nil, s) = s
        |   evalargs(n, f::ft, a::at, s) = let val (a', s', _) = eval(s, t) a
            val _ = mvl := (mv n, a')::(!mvl) in (f,a')::evalargs(n+1, ft, at, s') end
        val s2 = evalargs(1, bv, args, s')
        val mf = assoc(f, t')
        fun do_mf (tc_mf(lambda(bv, mb))) (a1, a2) =
                                    eval (!mvl @ ("arg1", a1)::("arg2", a2)::s, t) mb
        |   do_mf (_) (a1, a2) = (a1, s, t)
     val _ = do_mf mf (Int 0, Int 0)
     val (r, s3, _) = eval (s2, t') body
     val (r', _, _) = do_mf mf (Int 0, r)
     in (r', s3, t')
     end
```

Figure 2: The clause for Function Apply taken from our toy interpeter. For clarity, it is first shown without the entry and exit calls to the `mf` accessor joinpoint. The triple returned is the result, the modified environment σ' and the modified textual environment, **Te**'.

```
 datatype exp =
 Str of string | Int of int | Var of str
| plus of exp * exp | ...
| ... other common expression operators,
| lambda of exp list * exp
| meta of exp * exp
| apply of exp * exp list
| block of exp list
| defun of ...
```

Rather than presenting only the denotational semantics for the embedded pre-processor, we alternate the presentation by giving fragments of SML from a toy implementation of the interpreter. Hence, we write '`eval(ast, sigma, text)`', instead of $[\![\,\texttt{ast}\,]\!]_{(\sigma,\mathbf{Te})}$, where `ast` is a fragment of abstract syntax tree, σ is an association list for the environment, mapping variables to values, and **Te** is our new textual environment. In the compiler, as opposed to the interpreter, σ is a symbol table mapping variables to run-time store locations. Additional arguments would be needed to support either dynamic free variables and/or the OO '`this`' current object pointer, but these are bookwork and omitted for clarity. The demonstrator in SML can be downloaded from the following URL `http://www.cl.cam.ac.uk/users/djg/aspectsdemo`.

A full implementation of the textual environment, **Te**, would be too long to present in this paper, and its fine detail is not very important. The significant aspects are:

1. It is initialised as a set of bindings/tie-offs by command line flags, such as the -D flag used in C/C++, the source file path, using URI etc. and from other compile-time environment settings.
2. It is temporarily augmented, in the style of a LIFO stack, by an explicit '`meta`' construct as well as by entry to each nested block or textual inclusion.
3. A set of access functions and predicates provided as natives in the HLL are able to extract values and test properties of **Te**.
4. Names of variables and functions appearing elsewhere in the source program can be stored in **Te** to produce special behaviour where they occur.

The simplest access function would be the direct occurrence in an HLL expression of the name of a textual variable, bound only in **Te**. Variables are looked up in **Te** before σ to give precedence to tie-offs. To avoid over-pollution of the expression namespace, specific textual values should be extracted from **Te** with an HLL primitive such as '`_T()`'. For example, $[\![\,\texttt{\_T}(name)\,]\!]_{(\sigma,\mathbf{Te})} = \texttt{name}(\mathbf{Te})$, where `name` is one of many possible builtin accessor functions for textual context, e.g. one that retrieves the clos-

est textually-surrounding basic block name. Others would access line number and file and directory name.

Conditional compilation is implemented using the language's conventional conditional constructs, such as '`if`', '`case`' and '`?:`', where the guard expression makes access to **Te**.

The sequence rule, typically denoted in the concrete syntax with semicolon, is augmented by the HLL parser to supply additional context information such as the line number for each sequence operator. We denote the meta information at the semicolon by the suffix a;

$$\begin{array}{rcl} [\![\ e1;_a\ e2\ ]\!]_{(\sigma,t)} & = & \\ \texttt{let}\ (\_,\sigma',\mathbf{Te}') & = & [\![\ e1\ ]\!]_{(\sigma,\mathbf{Te})} \\ & \texttt{in} & [\![\ e2\ ]\!]_{(\sigma',a@\mathbf{Te}')} \end{array}$$

The sequence rule evaluates $e2$ using both environments returned from $e1$. Imperative basic blocks (generally denoted with braces or `begin/end`) are built out of the sequence rule in the normal way, except that the finally returned textual environment is the initial one that acted at the start of the block, thereby deleting bindings created inside the block.

User tie-offs (bindings) can be locally introduced into **Te** with the HLL '`meta`' construct that augments the textual context for the remainder of the current basic block.

$$[\![\ \texttt{meta}(v,e)\ ]\!]_{(\sigma,\mathbf{Te})} = (\bot,\sigma,(v,[\![\ e\ ]\!]_{(\texttt{nil},\mathbf{Te})})::\mathbf{Te})$$

Note that σ is ignored when evaluating e since it is not known until runtime in the compiler version.

The basic function application step, when using the textual environment, is presented in the top part of figure 2. This implements the procedure and function call operation, using call-by-value. The actual parameters are evaluated in order, leading to successive new bindings in σ, as well as any other side effects, before the body is evaluated in the final σ, denoted `s2`.[45] Note that the textual environment for evaluating the body, `t'`, is that from the function definition, rather that that of the caller. In an interpreter, it would be helpful to provide access to aspects of the caller's textual environment, but this would add considerable run-time overheads, if supported for separately-compiled modules.

[4]With the given simple code, the binding of the bound variable is left in the returned environment, $s3$, but ideally this would not be the case in reality, such as our own reference implementation.

[5]Where dynamic-free variables are used, the eval of f will return a closure to augment sigma during the eval of the body. This form of cross-cutting should be considered to be applied to all of our fragments, but we do not show it for clarity.

To provide logging and accessor functions for formal parameter, variable and field access operations we allow user-defined *interceptor* functions to be registered in **Te**, associated with any variable, that are called when that variable is read or written (second argument is a 1 for a write). The execution semantics for this are where $f : \alpha * \texttt{int} \rightarrow \alpha$ is associated with variable b, then, on a right-hand side, $[\![\ b\ ]\!]_{(\sigma,\mathbf{Te})}$ is replaced with $f([\![\ b\ ]\!]_{(\sigma,\mathbf{Te})},0)$ and, on the left-hand side $[\![\ b := e\ ]\!]_{(\sigma,\mathbf{Te})}$ is replaced with $b := f([\![\ e\ ]\!]_{(\sigma,\mathbf{Te})},1)$. This is a more general form of the tie-off, but can be defined with the same concrete syntax: namely `meta`(f,e). Locality of operation is controlled both by the restricted scope of the `meta` construct and the ability of the accessor functions to test **Te** to gate their behaviour. To achieve global control, starting values are established in **Te** on the compiler command line or via the IDE.

Interceptor functions also serve at the function call and return join points. They operate before the call, but after evaluation of the arguments, or at the return and on the return value. The `meta` construct can again be used to set up the desired action, associating a number of user constant tie-offs or user interceptor functions with the locally enclosing function or any named function called while the subsequent textual environment holds.

Specifically, '`meta`$(\texttt{af},\texttt{mf});e2$' causes all calls to `af` within $e2$ and any subsequent sequential commands in the same basic block to have their return value passed through function `mf`, for logging or tie-off. The second argument to `mf` is a 1 on the return stroke. On the before-call stroke, the second argument is a 0 and its return value is ignored. For callee side interception, `af` is replaced in the `meta` statement with null or some other token to signify the current function in all functions defined within block starting with $e2$.

The implementation of the caller's side interceptions is illustrated in the lower part of figure 2, although, for brevity, only one interceptor function, `mf`, is retrieved per function call instance. After evaluating the actual arguments, **Te** is searched by the caller for a definition pertaining to the called function. In an efficient interpreter, the search result would be cached, using whatever technique is already deployed for optimising branches, whereas for compilation the search is only made once anyway. Also shown in the code is a helpful facility to intercept the caller's arguments in either the pre- or post-call joinpoint. It would be handy to access these during the execution of `mf` using the formal names they are bound to in the callee, but this cannot be done if the callee is compiled separately or a computed branch is used where different formal identifiers are used in different destinations. Therefore, as a fallback, hardwired, stylised identifiers are always provided, such as _1, _2, to ac-

cess the actuals by positional index.

The implementation of the callee-side joinpoints is similar. As mentioned, a special token, such as the empty string, should be passed as the first argument to the `meta` statement to register an interceptor function for entry and exit to the currently textually enclosing procedure or function definition.

Another useful feature is for **Te** to contain a handle on the stack pointer so that calls can be associated with their returns. The actual stack pointer is easily mapped to a simple integer on most machines, and the integer can then be accessed as '`_sp`'. Although it is intended that only the relative values of the integer have meaning, bugs that arise from use of the actual values will tend to be machine-dependent. Compile-time static analysis can ensure that no reliance of actual values is used, but we have not implemented that.

Although we have only implemented an interpreter for the textual environment feature, we have spoken of the constraints and benefits arising form the compiled implementation. Compilation is certainly our intended medium, not least for efficiency. Standard techniques for converting an interpreter to a compiler (Futamura, 1999) are not made less practical by our approach.

Small-scale trials would be the best form of evaluation of the presented work, but we have not had a chance to start them. Nonetheless, if the reader now scans again the list of applications addressed by the two-level system, we believe it is more-or-less obvious that use of our textual environment, which is automatically augmented with meta-information by the command line, IDE, textual inclusion and named block and sequencing operators, can adequately address the application list.

3 CONCLUSION

This paper has presented an original and comprehensive definition of weaving and cross-cutting methods and applications. We asserted that the main uses are conventionally met using a single level of meta-programming (not a new assertion (Volder, 1999)). Two possible levels were presented: pre-processor and customised HLL. We used the C/C++ pre-processor as our main example owing to it being widely familiar. We then presented the essence of a general-purpose, single-level compilation technique that provides all of the methods and applications we found in the pre-processor. Our solution requires almost no additional syntax in a concrete implementation and is therefore claimed to be superior to other proposals.

An extension to our system would allow writes as well as reads to the textual environment. This would facilitate storage of meta-data needed for emerging dynamic-binding applications based on reflection APIs and so on.

Please note that although we have used a functional language (SML) to express the main evaluation function, our approach applies equally to imperative and functional target languages.

REFERENCES

Futamura, Y. (1999). Partial evaluation of computation process - an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355.

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375.

Volder, K. D. (1999). Aspect-oriented logic meta programming.