"Learn to do Verification with AOP?

We've just learned OOP!"

Dr David Robinson, Jason Sprott, Gordon Allan

Verilab Ltd.

david.robinson@verilab.com, jason.sprott@verilab.com,
gordon.allan@verilab.com

ABSTRACT: Recent versions of Vera contain language structures to support Aspect Oriented Programming (AOP), a relatively new software paradigm developed to solve some of the perceived shortcomings of Object Oriented Programming (OOP). This paper provides an introduction to AOP that will simplify its adoption for verification engineers with OOP experience. We make the case for using AOP in Vera testbench designs, by showing several common testbench scenarios that are not easy to solve using OOP techniques alone. We show how the use of AOP can ease the creation, maintenance and reuse of testbenches. We also discuss some of the problems that can be introduced if AOP is misused.

1 Introduction

Recent developments in verification languages have required many verification engineers to learn the Object Oriented Programming (OOP) paradigm. For many, this learning process may not be quite finished. The latest release of Vera (6.2) supports Aspect Oriented Programming (AOP), a relatively new software design paradigm that attempts to address some of the perceived shortcomings of OOP. This paper is an introduction to AOP for verification engineers with OOP experience. It does so by answering some questions an OOP designer may have:

- "What is AOP?"
- "What would I use AOP for?"
- "Can I use it with OOP?"
- "How do I use it in my testbench?"

AOP introduces some new concepts, such as "aspects", "concerns", "cross-cuts", "advice" and "introduction" that may seem initially confusing. Unfortunately, unlike OOP, AOP principles are far less likely to be understood by local software experts; AOP is still new even in the software world. The first reaction on exposure to these concepts is typically "what do they do?" and the second is "why would I want to do that?". Both points will be covered in later sections. Once we have explained the basic concepts of AOP, we turn our attention to their use in actual testbenches.

This paper contains 5 further sections. In section 2, we make the case for AOP, showing several common testbench scenarios that cannot be easily solved using OOP techniques alone. In section 3, we introduce the concepts behind AOP, using OOP constructs wherever possible, to explain their behaviour.

Section 4 revisits the scenarios introduced in section 2, and shows how the new AOP techniques work in conjunction with existing OOP techniques to remove the previously identified deficiencies. Issues relating to the creation, maintenance and reuse of testbenches will be explored.

In section 5 we discuss some of the potential problems that can be introduced if AOP is misused. Our conclusions are presented in section 6.

2 "I know OOP, why do I need AOP? "

OOP has been used in many projects, and most OOP practitioners are satisfied with its ability to decompose a problem into interacting objects, each of which should handle a single functional concern[1]. This "separation of concerns" allows the OOP designer to encapsulate all the code dealing with a particular concern in one place. This divides the size of the design problem the designer has to deal with at any one time, easing the maintenance overhead of the design, and increasing the reuse potential of each object. However, after gaining experience in OOP, it starts to become clear that something is wrong. Even when a problem is decomposed into its separate concerns, testbench designers still have to concentrate on many different concerns at once, testbench maintenance has not become trivial, and simple-and-widespread class reuse has remained elusive [2]. We use the code in Figure 1 to show why OOP is not delivering on all of its promises.

This code represents a single task from a class that is used to manage a DMA controller. The DMA controller has a number of channels, and the configuration of a channel is specified in an object of type DMATransactionClass. The testbench can configure the DMA controller using an AHB bus functional model (BFM), which is referenced through the m_ahbBfm variable in the code. Simulation output is achieved through a variable of type LoggerClass, called m_log. Identifiers beginning with m_ are local class variables, and identifiers in all upper case are constant values defined elsewhere in the program.

[1] A concern can be loosely defined as a concept, goal or purpose [1]

```
task DMAManagementClass::enableChannel(integer channelNumber,
                                       DMATransactionClass transaction){
  if(m_enableExecutionTrace == 1){
    m_log.log(SEV_TRACE_DEBUG,
              psprintf("Entered DMAManagementClass::enableChannel at
                        time %0d ns", get_time(LO)));
    m_log.log(SEV_TRACE_DEBUG,psprintf("...Channel Number = %0d",
              channelNumber));
    m_log.log(SEV_TRACE_DEBUG, psprintf("...Transaction:"));
    transaction.display();
  }

  if(channelNumber < 0 || channelNumber > NUMBER_OF_CHANNELS - 1){
    m_log.log(SEV_FATAL, psprintf("DMAManagementClass::enableChannel
                                  %0d is not a valid channel number",
                                  channelNumber));
  }
  if(transaction == null){
    m_log.log(SEV_FATAL, "DMAManagementClass::enableChannel
                         Transaction is null");
  }

  // Get access to the BFM
  void = semaphore_get(WAIT, m_ahbBfm.getSemaphoreID(), 1);

  // Enable the channel
  m_channelEnables[channelNumber] = transaction.getChannelEnable();
  m_ahbBfm.write(CHANNEL_ENABLE_ADDR, m_channelEnables);

  // Release the BFM
  semaphore_put(m_ahbBfm.getSemaphoreID(), 1);

  if(m_enableExecutionTrace == 1){
    m_log.log(SEV_TRACE_DEBUG, psprintf("Exiting
    DMAManagementClass::enableChannel at time %0d ns", get_time(LO)));
  }
}
```

Figure 1 : Example of a typical class method

In an ideal world, the task in Figure 1 would only concern itself with enabling the specified DMA channel, but we can see that in a real system, it has more work to do. In this example, the majority of the code deals with the following non-DMA related issues:

- Program-execution trace. This requires every method in the design to:

    - log when it was entered (line 3), and what its parameters were, if relevant (lines 4 and 5)
    - log when it exited (line 22), and what its return value was, if relevant (not shown in this example)
    - do none of the above if the functionality was switched off for performance reasons (lines 2 and 21)
- Precondition checking. Every method in the design must:
    - check that its inputs are valid before using them (lines 8 to 13)
    - deal with errors in a consistent manner (lines 9 to 12)
- Resource access. Every method in the design accessing this resource must:
    - wait until it can get access to the BFM before enabling the channel (line 15)
    - release the BFM when it has finished using it (line 20)

In fact, only two lines (17 and 18) in Figure 1 deal with the intended functional concern of the task. This is called a dominant concern [1], and is made so by the designer; he could have chosen one of the other concerns to be dominant. It is the dominant concerns that impose a structure on the object hierarchy. The remaining lines are there to implement other concerns: the program-execution trace, the precondition checking and the resource access management. These three concerns are essential to the overall operation of the testbench, but they are irrelevant to the dominant concern of the task - enabling a DMA channel. However, there is no mechanism in OOP that allows these concerns to be separated from each other and encapsulated.

Two or more concerns are known as cross-cutting if their code cannot be neatly separated from each other. We do not typically refer to the dominant concern of an object as a cross-cutting concern, but instead apply the phrase to those concerns that cut across it. Two symptoms of cross-cutting concerns are code-tangling and code-scattering [2]. The former is where the code for multiple concerns is tangled together, forcing an object to deal with multiple concerns simultaneously, and the latter is where the code for a particular concern (say, resource access) appears in multiple places in the design.

Returning again to the code in Figure 1, consider the potential problems that exist with the cross-cutting concerns.

- They prevent reuse:
    - The DMAManagementClass cannot be reused in another testbench that does not implement program-execution trace, precondition checking and resource access in exactly the same way as in this testbench
- They make it easy to introduce bugs:
    - The program-execution trace is a system-wide concern, which means that the testbench designers have to implement it correctly in every method in the design. If the designer forgets to include it in a method, or for example, forgets to include the time in the log statements, the program trace will be incomplete. If the designer forgets to put the enable switch around it, it will always trace, even when it is meant to be switched off.
    - The precondition checking has to be done for every method that takes in a parameter that should be checked. If the designer of a task forgets to check, or uses the wrong bounds to check against, bugs can occur due to incorrect data being used by the testbench

    - The resource access concern will be duplicated within other methods in the class, and possibly in other classes. If done incorrectly in one of these, or left out completely, the whole scheme is rendered incorrect

- They complicate maintenance of the testbench:

    - If the trace information content or format needs to be changed, then every method in the design has to be visited and updated. If another check is required for a certain data type, then every method in the design that takes a parameter of that type has to be visited and updated. If the resource access scheme has to change, then every method in the design that accesses that resource has to be visited and updated.

It is these cross-cutting concerns that are a problem in OOP. The paradigm does not offer a neat solution to deal with those concerns that cannot be completely encapsulated from all other concerns within the design. Non-localised concerns end up being tangled in with the nicely encapsulated code, in an ad hoc manner. While experienced OOP practitioners can minimise the problems introduced by these cross-cutting concerns (e.g. by using design patterns [2]), they are hard to eradicate completely.

3 "OK, what is AOP?"

AOP is a new design paradigm that allows cross-cutting concerns to be separated and encapsulated. Dominant concerns are coded using OOP techniques as before, and new constructs are used to code the cross-cutting concerns and integrate them with the dominant concerns. Rather than explain all of the concepts in AOP, we will limit ourselves to the subset actually supported by the current version of Vera.

Join Point: A join point is a well defined point in the flow of a program [1] that can be used to reintegrate cross-cutting concerns. Method calls are the only join points supported by Vera.

Advice: Vera code that is to be executed at a ***join point***. Three types of ***advice*** exist: ***before***, ***after*** and ***around***. ***Before advice*** gets executed before the code originally at the ***join point***, and ***after advice*** gets executed after the code originally at the ***join point***. ***Around advice*** surrounds the code at the ***join point***, and it is up to the ***advice*** writer to decide when, if ever, to call the original code using the new proceed keyword. ***Advice*** can be emulated in OOP by extending the class and overriding the method (***join point***). ***Before***, ***after*** and ***around*** just depend on the placement of super.method() in relation to the new code.

`task MyExtendedClass::foo(){`
`  printf("Before Advice\n");`
`  super.foo();`
`}`
	Before Advice

`task MyExtendedClass::foo(){`
`  super.foo();`
`  printf("After Advice\n");`
`}`
	After Advice

`task MyExtendedClass::foo(){`
`  printf("Around Advice (before join point)\n");`
`  super.foo(); // Optional`
`  printf("Around Advice (after join point)\n");`
`}`
	Around Advice

Figure 2: Different types of advice emulated in OOP

Although OOP can be used to emulate advice, it cannot match all of its capabilities. For instance, if the OOP code had a handle to the base class of MyExtendedClass, and the original foo method was not virtual, then the original foo method will get called, and not the new method. Another limitation is that you cannot emulate advice on local methods.

Introduction: ***introductions*** in AOP introduce new members into a class, where a member can be a task, a function, a variable, a constraint definition or a coverage definition. Like ***advice***, this can be emulated using OOP simply be extending the class with the new members. However, there are some limitations that prevent OOP emulating all of the functionality of AOP introduction. In

OOP, new members that need to reference an existing local member cannot do so, and existing members cannot access any introduced members. Both of these are possible in AOP.

Aspect: An ***aspect*** is the AOP equivalent of an object. That is, it is the basic unit of encapsulation. An ***aspect*** modifies a single named-class, and contains the ***advice*** and/or ***introductions*** that will modify that class. A class can have more than one ***aspect*** attached, but an ***aspect*** cannot modify more than one class. This means that multi-class cross-cuts cannot be encapsulated in one ***aspect***, but they can at least be encapsulated in one file or directory if desired.

Weaver: The ***weaver*** is the tool that integrates the ***aspects*** with the objects. Conceptually, it acts like a pre-processor that takes the class, physically adds all ***introductions*** to the class header, and inserts the ***advice*** code into the ***join points***. Vera only supports static (compile time) weaving, so all ***introductions*** and ***advice*** apply to all instances of a class. That is, you cannot add an ***aspect*** to one object and have another of the same class behave as it would without the ***aspect***.

4 "How do I use it with OOP?"

For all verification engineers who are considering adopting AOP, the good news is that AOP works in conjunction with OOP. It does not require engineers to completely re-skill in order to use it, and can be gradually introduced into projects, minimising the chance of failure if difficulties are encountered. This is different from switching to OOP, where an "all or nothing" approach is normally required. However, reading the technical details of a new programming paradigm and deciding just to adopt it gradually, are not sufficient for its successful adoption. A new practitioner needs to know when to use the new techniques, and when not to; to know when to keep doing it "the old way". This is particularly true of AOP, because it is easy to over use, and experience has shown that this can easily create more problems than it solves (see section 5). Here, we are going to show how to use AOP to complement OOP to create a testbench that is of a higher quality than can be easily achieved through using either technique on its own.

As AOP works with OOP, we are going to assume an OOP based project as a starting point. It does not really matter if the testbench is being written from scratch, or if an existing testbench is being refactored to use AOP. The approach is the same in each case: identify a cross-cut, and code it as an aspect. Depending on the reason that AOP is being used, some cross-cuts may be easier to identify than others. For instance, if AOP is being used to make classes more reusable, a cross-cut that helps in code maintenance may not be identified, as the designer is focusing on other concerns. However, if code maintenance is the goal, the same aspect dealing with reuse may not be identified, as that is not what the designer is concentrating on. Therefore, the first decision to be made when using AOP is why it is being used. Is class reuse, reduced bug potential or code maintenance the main goal? There are other reasons, some of which will be covered, but the novice user should initially limit themselves to these three. The following list will help identify areas of code where AOP can be successfully used.

- Identifying cross-cuts that improve class reuse:
    - Does the class contain any code that would prevent it being used in another testbench?
        - Does it rely on a testbench-wide mechanism to control behaviour? If so, use an ***aspect*** to deal with it.
        - Does it contain code that controls access to an external resource that is used by the class? Can you guarantee that the same access mechanism would be used in *every* other testbench that might use this class? If not, use an ***aspect*** to deal with it.
- Identifying cross-cuts that reduce the potential for bugs:
    - Implement pre-condition checking as an ***aspect***. This makes it easier to ensure that all pre-conditions are checked and handled in a consistent way.

    - Does the class contain code common to different methods within the class, or to other classes?

        - Does this have to be manually implemented in each method? If so, consider using an ***aspect***. Pre-condition checking is an example of this, because each method has to check different pre-conditions. Trace debugging is another example.

        - Are you relying on other designers (or even yourself) to remember to implement it in other methods and classes? If so, consider using an ***aspect***.

        - Are you relying on other designers (or even yourself) to implement it *the same way* in other methods and classes? If so, consider using an ***aspect***.

- Identifying cross-cuts that make testbench maintenance easier:

    - Does the class contain code that, if changed, would require other methods and classes to be changed in the same way to remain consistent? If so, use an ***aspect***.

    - Does this code implement a policy that needs to be reviewed or enforced? Do other methods or classes implement this policy as well? If so, use an ***aspect*** to implement the policy, because it will be easier to review or enforce if all instances of it are grouped in one place. For instance, if all trace debugging code was contained in the same location, the logging format could be easily reviewed for consistency

Applying these questions to the code fragment presented in Figure 1 gives the following results:

Lines 2 to 7 and 21 to 23 deal with execution trace. This is controlled by a member variable called m_enableExecutionTrace, which has to be set to 1 for trace to occur. This is a testbench wide control mechanism, as all other methods in all other classes have to interpret m_enableExecutionTrace in the same way. These lines cause reuse problems, because they use a control mechanism which may not be present in other testbenches. The format of the trace may also be different in other testbenches, as a different logger, or a different debug level (other than SEV_TRACE_DEBUG), may be used. These lines also make bugs likely, because they must be manually implemented in every method in every class in the testbench. Failure to include execution trace, or failure to control it properly, would be a bug. Maintenance is also made harder because of this code. Say the mechanism was to be changed so that there were different levels of trace debug. Level 0 has no debug, level 1 just displays entry and exit, level 2 adds entry and exit times, and level 3 also includes parameter and return values. Changing this would involve modifying every method in the design; a time consuming process because the code that needs modified is not encapsulated in the same place.

Lines 8 to 13 implement precondition checking. For ultimate safety, all methods in the design should do this. There is no reuse problem with these lines, because the boundaries are unlikely to change in another testbench, and the one that does (the maximum channel number) has been encapsulated. However, we are relying on the method's author to add the checking, and get the values that the parameters are checked against correct. Experience shows that this is unlikely to occur for all methods. By using an ***aspect*** to implement this functionality, all precondition checking can be added at a later date[2], and stored in one location which will make reviews easier.

Lines 15 and 20 are used to secure access to the AHB BFM. This presents reuse problems, as access may be handled differently in other testbenches. Access may be freely available, or something other than a semaphore used. These lines are also bug prone, as the testbench relies on every method accessing the AHB BFM to include them. Forgetting to include them in even one method may lead to a complex debugging problem. Testbench maintenance is also affected by these lines of code, as they manually implement a testbench-wide, resource-access policy that will

[2] Experience also shows that this is also unlikely to happen, but at least we are now in a better position to do it, or work out how much has been done

be difficult to change. By using an ***aspect*** to implement resource access, the policy can be changed and reviewed more easily.

Refactoring the code presented in Figure 1 as described above results in the original code fragment being reduced to the two lines of code shown in Figure 3.

```
task DMAManagementClass::enableChannel(integer channelNumber,
                                       DMATransactionClass transaction){
  // Enable the channel
  m_channelEnables[channelNumber] = transaction.getChannelEnable();
  m_ahbBfm.write(CHANNEL_ENABLE_ADDR, m_channelEnables);
}
```

Figure 3 : The code from Figure 1 reduced to its core concern

```
extends trace_debug_DMAManagementClass (DMAManagementClass)
  dominates(ahb_bfm_resource_access_DMAManagementClass) {

  around task enableChannel(integer channelNumber,
                            DMATransactionClass transaction){
    m_log.log(SEV_TRACE_DEBUG, psprintf("Entered
                               DMAManagementClass::enableChannel at time
                               %0d ns",  get_time(LO)));
    m_log.log(SEV_TRACE_DEBUG, psprintf("...Channel Number = %0d",
                               channelNumber));

    proceed;   // Call the original task

    m_log.log(SEV_TRACE_DEBUG, psprintf("Exiting
                               DMAManagementClass::enableChannel at time
                               %0d ns",  get_time(LO)));
  }
}
```

Figure 4: The *aspect* that handles trace debugging for the code in Figure 3

```
extends
precondition_check_transaction_DMAManagementClass(DMAManagementClass)
  dominates(trace_debug_DMAManagementClass,
            ahb_bfm_resource_access_DMAManagementClass) {
```

```
  before task enableChannel(integer channelNumber,
                            DMATransactionClass transaction){
    if(transaction == null){
      m_log.log(SEV_FATAL, psprintf("DMAManagementClass::enableChannel
                                    called with transaction == null"));
    }
  }
}
```

Figure 5: The *aspect* that handles precondition checking of the transaction parameter for the code in Figure 3

```
extends ahb_bfm_resource_access_DMAManagementClass(DMAManagementClass) {
  around task enableChannel(integer channelNumber,
                             DMATransactionClass transaction){
    // Get access to the BFM
    void = semaphore_get(WAIT, m_ahbBfm.getSemaphoreID(), 1);

    proceed;  // Call the original task

    // Release the BFM
    semaphore_put(m_ahbBfm.getSemaphoreID(), 1);
  }
}
```

Figure 6: The *aspect* that handles resource access for the code in Figure 3

```
extends
precondition_check_channel_number_DMAManagementClass (DMAManagementClass)
  dominates(trace_debug_DMAManagementClass,
            ahb_bfm_resource_access_DMAManagementClass) {
 task aop_check_channel_number_precondition(integer param, string caller){
   if(param < 0){
    m_log.log(SEV_WARN, psprintf("%s called with a channel number = %0d.",
              caller, param));
    m_log.log(SEV_FATAL, psprintf("It must be greater than 0"));
   }

   if(param > NUMBER_OF_CHANNELS -1 ){
    m_log.log(SEV_WARN, psprintf("%s called with a channel number = %0d.",
              caller, param));
    m_log.log(SEV_FATAL, psprintf("It must be less than %0d",
              NUMBER_OF_CHANNELS));
   }
 }

 before task enableChannel(integer channelNumber,
                           DMATransactionClass transaction){
   aop_check_channel_number_precondition(channelNumber
                                "DMATransactionClass::enableChannel");
  }
}
```

Figure 7: The *aspect* that handles precondition checking of the channel number parameter for the code in Figure 3

One of the quoted advantages of AOP is that it allows common code scattered throughout the class hierarchy to be encapsulated, allowing easier debug and maintenance. However, there are limitations in Vera that prevent full encapsulation. The problem is that an ***advice*** block can only be attached to one method, so if *n* methods require the same ***advice*** (say, to check the channel number is valid), then *n* ***advice*** blocks have to be created. Vera files that contain AOP code cannot include non-AOP code, so it is not possible to declare a local function or task to implement the content of the ***advice*** block. It is however, possible to introduce a function or task into the class associated with the ***aspect***, as shown in Figure 7. It adds a new task to the DMAManagementClass called aop_check_channel_number_precondition() which does the actual checking. This task can be called by all ***advice*** blocks in the DMAManagementClass. Unfortunately, this cannot be added into other classes that need it, so the code will have to be duplicated in their ***aspect***.

Although we may have ended up with more code, because of the overhead of declaring an ***aspect***, the code we now have is of a higher quality. The class now deals exclusively with its main concern of managing a DMA controller. All of the infrastructure code that was tangled up with the class has been untangled and encapsulated in smaller units that can be changed, reviewed and replaced much more easily than before.

We have shown that AOP can be used to improve the reusability of a class, and to make a testbench less bug prone and more maintainable. While these are the most obvious reasons to use

AOP, they are not the only reasons. We also advocate using AOP techniques during the initial testbench development to set constraints that stop certain design features from being exercised. This could happen because they have not been implemented in the RTL, have not been supported yet in the testbench, or there is a bug that causes simulations to fail when a regression is about to be run. It is common to add constraints to a design to prevent features from being exercised. However, it is also common to forget to remove these constraints. By using AOP to apply these constraints, they can be contained in a separate location, and this can be checked quickly to see what features are not being exercised. As each feature is implemented or fixed, the constraint can be removed. At the end of the testbench development, the development constraints location should be empty. Figure 8 shows an ***aspect*** that constrains the size of a DMA block to 10 units.

```
extends constrain_block_size_DMATransactionClass (DMATransactionClass) {
  constraint block_size{
    m_blockSize == 10;
  }
}
```

Figure 8: An *aspect* that constrains the DMA Transaction for development reasons

5 "What are the risks of AOP?"

5.1 Too Much Too Soon

A risk to a project using AOP techniques, especially for the first time, is to attempt too much too soon. Once the basic concepts of AOP are understood, it is easy to see many places in a program where AOP might apply. However, at least in the first instance, it is not a good idea to implement every cross-cutting concern found as an ***aspect***. Managing programs with many ***aspects*** can be a big task, with some serious issues. Only in the cases that clearly justify the need should AOP be used. The other cross-cutting concerns can always be re-coded using AOP during a later refactoring phase [3].

AOP is not an ‘all or nothing’ methodology; since AOP complements OOP, it is possible to introduce instances of the methodology in a controlled and gradual way. Adopting AOP on a wide scale will require new policies, education and a good amount of actual coding practice.

5.2 The Temptation of the Quick Fix

When using AOP it is possible to add to, or alter, the functionality of any part of an existing class. Man years of AOP experience highlighted this feature as easily abused. It is possible to make quick fixes (patches) to faulty code in any class, by applying an ***aspect*** to it. This is not the intended use of AOP and tends to lead to code that is confusing and difficult to maintain. If there is faulty code in a class, it should be fixed in that class. The ‘fix’ is not a separate concern; it belongs as part of the class. If an ***aspect*** is used to fix the problem, only programs that implement that ***aspect*** will pick up the fix. In addition, there is nothing in the faulty class to tell a user that an ***aspect*** is required for the class to operate correctly.

5.3 Undeterminable Program Flow

The quick fix behaviour leads to the situation where it can become almost impossible to determine the execution flow of a testbench. During code analysis (debugging, maintenance, reviews, etc), looking at a method call on an object is not enough to determine what will actually be executed. This is because the functionality of a class may not be represented in one place. It is represented by the class, plus zero or more ***aspects***. This is a problem with AOP, and one that can be avoided by good CAD tools or by careful coding practises. By organising the ***aspect*** code into sensible files, and by carefully choosing the functionality to implement as ***aspects***, this problem can be minimised.

5.4 Tight Coupling and Fragile Code

The inter-relationships between ***aspects*** is something that should be considered in AOP. It is possible, and sometimes desirable, to have multiple ***aspects*** acting on the same area of functionality within a class. This can lead to ordering dependencies and functionality for a given area being distributed across multiple ***aspects***. In Vera there is a construct for managing the order in which these ***aspects*** are applied to the class: a ***dominates*** list (see Figure 4). This feature is necessary, but may come with a high maintenance price tag. When new ***aspects*** are added or removed, lists in other ***aspects*** may have to be altered accordingly. Where ***dominates*** lists are used, there is tight coupling between ***aspects*** and therefore rigid code. This can make programs difficult to change.

5.5 Safety

Since the behaviour of code can be altered by applying ***aspects***, it is possible to break previously working code. A bug could be introduced in the new ***aspect*** code, or in the ordering of ***aspects***. Where code has been tested and proved working, any outside 'tampering' must be assumed *capable* of having an adverse effect. In some organisations the ability to affect verified code in this way is treated as a significant safety concern. This is particularly true where code quality is important and code is shared with others outside the development team

6 Conclusions

The introduction of AOP into Vera promises to make testbenches easier to design, easier to reuse and easier to maintain. It does so by allowing designers the chance to concentrate on the real issues in a testbench (the dominant concerns that actually verify that the design is correct), and by separating out many of the non-reusable ***aspects*** of a class (the non-dominant concerns). This paper has introduced the concepts behind AOP, and made a case for its use in testbench design. We have discussed a number of areas where AOP can easily and successfully be used by the AOP novice, and presented some guidelines for those who want to experiment further.

7 References

[1] Ken Wing Kuen Lee, "An Introduction to Aspect-Oriented Programming", The Hong Kong University of Science and Technology

[2] Ramnivas Laddad, "I want my AOP!, Part 1", Java World, http://www.javaworld.com

[3] Arie van Deursen, Marius Marin, Leon Moonen, “Aspect Mining and Refactoring”