AspectAda – Aspect Oriented Programming for Ada95

Knut H. Pedersen and Constantinos Constantinides
Department of Computer Science and Software Engineering
Concordia University
Montreal, Quebec, H3G 1M8, Canada
{kh_peder, cc}@cs.concordia.ca

ABSTRACT

Concerns for concurrent systems are not always easy to modularize within single units using traditional programming languages. The concept of aspect orientation can be applied to allow a modular implementation of these concerns. Existing programming languages has been extended with new language features to support aspect-orientation. The most dominant of these language extensions are AspectJ for the Java programming language. However, Java is not ideal for all types of applications, or there exists legacy systems that need to be maintained.

This paper presents AspectAda a new language extension to Ada95 and an AspectAda weaver tool built with the Ada Semantic Interface Specifciation (ASIS). The AspectAda language provides powerful language elements to facilitate aspect oriented programming in domains where Ada's capabilities are of high interest and the AspectAda weaver tool demonstrates the AspectAda language in action.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Contructs and Features D.1.m [**Programming Techniques**]: Aspect-Oriented Programming; D.2.3 [**Software Engineering**] Coding Tools and Techniques; D.2.7 [**Software Engineering**]: Maintenance, and Enhancement.

General Terms: Design, Languages, Reliability.

Keywords: Aspect-Oriented Programming (AOP), Concurrent Programming, Ada95, Ada Semantic Interface Specification (ASIS), Inheritance Anomaly

1. INTRODUCTION AND MOTIVATION

Despite the success of object-orientation in the effort to achieve separation of concerns, certain properties in complex object-oriented systems cannot be directly mapped from the problem domain to the solution space as a one-to-one mapping, and thus they cannot be localized in single modular units. The symptoms imposed by this phenomenon manifest themselves as the scattering of concerns across the decomposition hierarchy of the system and the tangling of concerns in modular units. As a result of these symptoms, the benefits of OOP cannot be fully utilized as developers are faced with a number of problems such as low level of cohesion of modular units, strong coupling between modular units, low level of reusability of code, low level of adaptability and difficult comprehensibility resulting in programs that are more error prone. Aspect-Oriented Programming (AOP) [11] is a term adopted to describe an increasing number of technologies and approaches that support the explicit capture of crosscutting concerns (also referred to as aspects) whereby the implementation of functional components and aspects is performed (relatively) separately, and their composition and coordination (referred to as weaving) is specified by a set of rules.

With the set of linguistic constructs as well as the weaving tool and IDE support that it provides, AspectJ [2] is perhaps the most notable AOP technology that provides aspect-oriented capabilities to the Java programming language. The contribution of AspectJ is twofold: (1) it has provided a benchmark language (including a joinpoint model) and (2) it has provided the community with an aspect-oriented vocabulary. Furthermore, AspectJ has influenced the design dimensions of other languages such as AspectC++ [15].

The Ada programming language provides rich support for concurrent and real-time programming that other programming languages lack. As concurrency is inherently a crosscutting concern, Ada poses the symptoms of scattering and tangling of concerns. In this paper we will present an aspect-oriented extension to the Ada95 language called AspectAda and we will describe the provisions of the language through a case study.

The outline of this paper is as follows: Section 2 gives an introduction to the language concepts using many code examples. Case studies applying AspectAda to well known cases are studied in section 3. Following this in section 4 the implementation of a prototype AspectAda weaver is outlined. In section 5 we discuss the effect of aspect-oriented programming using AspectAda on software quality. Future work is discussed in section 6, followed by the conclusion in section 7.

2. ASPECTADA LANGUAGE DESIGN

The AspectAda language is influenced by the language design of AspectJ. Aspect definitions are comprised by three essential notions which will be described in the following subsections: (1) joinpoints, (2) pointcuts and (3) advices. To illustrate the AspectAda language with examples we apply it to a Vector container type (not generic) similar to the Ada.Containers.Vectors proposed for Ada2005 [1].

2.1 The joinpoint model

In the literature, a joinpoint is defined as *"points in the component code where aspects can interfere"[15] or "a well-defined instant in the execution of a program."* [2] General-purpose aspect-oriented languages deploy joinpoints (or collections of joinpoints, called pointcuts) in order to define the (aspectual) behavior to be inserted at specific joinpoints within functional components. AspectJ provides a rich joinpoint model that includes calls to, or executions of methods, execution of an object constructor or destructor, or static initialization of a class.

In the Ada context, the call and execution of a method are transformed to call or execution of a subprogram of a package. Additionally Ada defines protected types with subprograms and entry points and tasks with entry points. These would equally well qualify as joinpoints. Nested subprograms are also allowed in Ada and would be a specialized subprogram joinpoint. Construction and destruction are only applicable for controlled types in Ada. The Initialize and Finalize methods will be equal to the AspectJ support for Java constructors and destructors, while Adjust (for adjustment after assignment) would make a new kind of joinpoint. AspectJ's join point for static initialization of a class aligns well with the statement block of the package body that is executed during elaboration of the package. The supported joinpoints for the AspectAda language can be summarized to be call or execution subprograms and entry points, initialization, finalization of controlled types and package elaboration. An example of a joinpoint from AspectAda is the execution of a procedure:

```
package Vectors
...
  procedure Append_Item(Container : in out Vector;
                        Item : in Element) is
  begin
     Container(Container.Last + 1) := Item;
     Container.Last := Container.Last + 1;
  end AddItem;
...
end Vectors;
```

The joinpoint is the actual execution of the procedure. Each joinpoint can be uniquely referenced from a pointcut using the qualified name and the parameters of the joinpoint.

2.2 Pointcuts

Pointcuts are collections of joinpoints. The pointcut describes the collection of joinpoints with a pointcut expression. The pointcut expression language is a special purpose language that provides a way of specifying identifier patterns, packagename patterns and to compose these to composite expressions using conditional operators. The pointcut expression language of AspectAda is strongly influenced by the AspectJ pointcut expression language. A small variation in AspectAda is that pointcuts are aligned to the Ada type system. AspectAda extends the language with a specific pointcut type called Pointcut. A normal object declaration with Pointcut as the subtype indication and an assignment of the pointcut expression is the mechanism to define a named pointcut in AspectAda. AspectJ on the other side extends the Java language with a new language construct not similar to the other Java constructs. The following example shows the definition of Add_Item_PC that is a named pointcut in AspectAda. It refers to all executions of the named procedure Vectors.Append_Item(in out Vector; in Element).

```
Add_Item_PC : Pointcut := execution
   Vectors.Append_Item(in out Vectors.Vector;
                       in Vectors.Element);
```

The Add_Item_PC uses an execution *pointcut function* that designates the procedure named Vectors.Append with a *match expression.* The procedures designated have two parameters where the first is an in out parameter of type Vectors.Vector and the second is an in parameter of type Vectors.Element. The identifier for the name of the parameters are not part of the pattern matching expression in AspectAda.

The Add_Item_PC exclusively designates the previously described Append_Item joinpoint. Hence the size of this pointcut's joinpoint collection is only one. The AspectAda support wildcards in order to build more powerful pointcut expressions. A search pattern can be specified that performs pattern matching of the joinpoint identifier as well as for the joinpoint parameters. The pointcut expression language has similarities with regular expressions for files in UNIX or DOS. It uses a case insensitive "globbing pattern" to match the joinpoints qualified name. The parameter matching is using a special purpose pattern matching language equal to AspectJ. Two dots represent any parameter sequence including no parameters at all. The *dotdot* construction can also be mixed with concrete specification of parameter modes and type names. The following example designates any subprogram in the package Vectors where the subprogram name starts with Append and the first parameter is an in out of Container type, and there are zero or more parameters following. Ada supports both named and positional parameter associations for subprograms. However, AspectAda only supports positional parameter associations.

```
Append_Container_PC : Pointcut :=
  execution Vectors.Append*(in out Container; ..);
```

To compose even more complex pointcuts AspectAda supports the conditional operators *and*, *or*, *xor* and *not*. In AspectAda these have the same semantics and precedence as they have in Ada. A previously defined named pointcut can also be reused in other pointcuts to create composite pointcuts without repetition of the pointcut expressions. A more complete example illustrating these capabilities is to define a pointcut that designates the execution of joinpoints which modify a Vector type.

```
Modify_Container_PC : Pointcut :=
   Append_Container_PC or
   execution Vectors.Insert*(..) or
   execution Vectors.Delete*(..);
```

Pointcuts are modularized in a specific AspectAda program unit named *weaver* that only contains pointcut definitions and generic aspect initialization (see subsection 2.3).

```
weaver Container_Composition is
  Append_Container_PC : Pointcut :=
    execution Vectors.Append*
      (in out Container; ..);

  Modify_Container_PC : Pointcut :=
     Append_Container_PC or
     execution Vectors.Insert*(..) or
     execution Vectors.Delete*(..);
...
end Container_Composition;
```

2.3 Aspect Types and Advices

An advice defines a block of code that specifies some behavior to be executed upon reaching certain pointcuts. Essentially, we can view a pointcut as a set of events to which we execute certain advices. There are three ways to associate an advice with a pointcut: (1) Run an advice *before* the pointcut, (2) run the advice just *after* the pointcut and (3) run the advice instead of running the pointcut (with the provision for the pointcut to resume normal execution). This type of advice is called an *around* advice.

Advices are declared for an *aspect type* similar to declaration of subprograms associated with Ada tagged types for object oriented programming in Ada95. State often needs to be shared between executions of related advices or between execution of the same advice. This is similar to how tagged records share state for object-oriented methods. In AspectAda new aspect types are defined by deriving from an AspectAda aspect type defined in the AspectAda standard run-time library. The syntax for the type declaration is equal to declaration of Ada record type extensions. The following is an example that derives from the predefined AspectAda type Simple_Aspect. The aspect type has a state that is used as a counter for the number of times the advice was executed.

```
type Counter_Aspect is
  new Simple_Aspect with
  record
     Number_Of_Times : Natural := 0;
  end record;
```

AspectAda defines two parent types for aspect types: Simple_Aspect and Detailed_Aspect. Simple_Aspect adds minimal overhead, while Detalied_Aspect have access to information about the current joinpont. The rationale to separate these is to allow performance for some aspects used in performance critical code, while for other aspects allowing information about current joinpoint. Examples of such aspects can be tracing aspects and aspects collecting statistics. The joinpoint information supported by Detailed_Aspect is the qualified name of the joinpoint identifier and the parameter names, parameter mode and the typename of each parameter.

Advice declarations and bodies are similar in syntax to procedure declarations and bodies. The difference is the keyword procedure is substituted with advice. The rationale for this is that the invocation semantics are very different. Procedures are executed by a procedure call in the Ada code. Advices on the other hand are executed if there exist a pointcut that designates the joinpoint that is to be executed and is associated with the advice. An advice body can have the same declarative part and sequence of statements as procedures can have. An illustration of this with AspectAda:

```
advice Before(Count_A : in out Counter_Aspect) is
begin
  Count_A.Number_Of_Times :=
    Count_A.Number_Of_Times + 1;
  Put_Line("Call count : " &
    Natural'Image(Count_A.Number_Of_Times);
end Before;
```

A before advice is declared for the Counter_Aspect aspect type. The advice declares the Counter_Aspect as an in out parameter mode and hence can both read and modify it within the statement block of the advice. The before advice is executed before the designated pointcut.

After advice is equal to before advice except that they run after the pointcut is executed. Around advice is the third type of advices and run instead of the joinpoint. This does not necessarly imply that the joinpoint is not executed, since a special proceed statement can be used to resume normal execution of the joinpoint. The proceed statement takes no arguments in AspectAda, since the arguments are not available in the advice code. Around advices can have exception handlers that intercept the normal exception flow. Exception handlers in advices are useful for advices that have specific behavior based on if an exception is thrown or not. An example of this is to unlock a mutex in case of exceptions that are raised.

```
advice Around(Lock_A : in out Lock_Aspect) is
begin
  Lock_A.Control.Lock;
  Proceed;
  Lock_A.Control.Unlock;
when others =>
  Lock_A.Control.Unlock;
  raise;
end Before;
```

An advice defined for a derived type of the Detailed_Aspect type can access information about the joinpoint that it is advicing. The information about the joinpoint can be used to build advices that add helpful debug information. An Example of this is a logger that log entry and exit of subprograms of a package.

```
advice Around(Logger_A : in Logger_Aspect) is
begin
  Put_Line("Enter => " &
     Image(Get_Join_Point(Logger_A).all));
  Proceed;
  Put_Line("Exit => " &
     Image(Get_Join_Point(Logger_A).all));
when others =>
  Put_Line("Exit (Exception) => " &
     Image(Get_Join_Point(Logger_A).all));
end Before;
```

Advices and pointcuts have to be associated. Aspect types can have advices that are connected to different pointcuts. An aspect type can also have more than one advice of a specific type; usually these advices are associated with two different pointcuts. The language support thus needs syntax to associate an advice with a pointcut. In AspectAda this is achieved using an *advice clause* placed after the advice declaration. The advice clause uses a new attribute called Pointcut to indicate that it sets the pointcut of the advice. If an aspect has two around advices there would be a name conflict. AspectAda resolves this by allowing the developer to use a prefix before the advice type name.

```
advice Before(Count_A : in out Counter_Aspect);
for Before'Pointcut use All_Creates_PC;
```

2.4 Aspects

The unit of modularity for aspect types and advices are called an aspect. Aspects are similar in syntax to Ada packages and just as packages consist of an *aspect specification* and an *aspect body*. The language constructions allowed in an aspect are extensions to what is allowed in packages. The extensions are that advices are allowed to be declared and the use of the aspect clause. Aspects typically group entities related to one aspect type as packages group entities related to an object in object oriented programming.

A minimum for an aspect specification includes an aspect type declaration and one advice declarations that take the aspect type as a parameter. The aspect body has to contain the advice bodies of the declared advices from the aspect specification. The aspect also has supports for a block of statements to be executed upon elaboration of the aspect. This can be used to initialize the state of the aspect type instance. Currently AspectAda does not support control of elaboration sequence.

The advices of an aspect are associated with a pointcut. AspectAda separates pointcut definitions from aspects and advices specification and bodies. Aspects thus need to be parameterized with pointcuts to bind a pointcut to an advice of an aspect. Ada generics are the means AspectAda use to parameterize the aspects. Thus AspectAda aspects are always generic aspects with at least one pointcut as a generic formal parameter. The following is a complete aspect specification and aspect body for the counter aspect.

```
generic
  To_Count : Pointcut;
aspect Counter_Aspects is
  type Counter_Aspect is
    new Aspect_Ada.Simple_Aspect with
    record
      Number_Of_Times : Natural := 0;
    end record;

  advice Before(Count_A : in out Counter_Aspect);
  for Before'Pointcut use To_Count;
end Counter_Aspects;

aspect body Counter_Aspects is
  advice Before(Count_A : in out Counter_Aspect)
  is begin
    Count_A.Number_Of_Times :=
      Count_A.Number_Of_Times + 1;
  Put_Line("Call count : " &
    Natural'Image(Count_A.Number_Of_Times);
  end Before;
end Counter_Aspects;
```

The rationale for separating an aspect into a specification and a body come from the vision that AspectAda should be as close as possible in syntax to Ada itself. Currently the separation does not add value in terms of reduced visibility or that aspect bodies can separately compiled without weaving to all designated joinpoints. AspectAda could in the future support separate compilation of aspect bodies to gain compilation performance at the cost of additional run time performance, since a new method call need to be done in the weaved source code.

2.5 Composition rules

In order to associate an aspect with one or more pointcuts AspectAda has a specific program unit called weaver. The weaver program unit contains all the weaving/composition rules for associating aspects with pointcuts. The weaver program unit contains pointcut definitions and aspect instantiations. The syntax of the weaver program unit is similar to Ada packages, but can only contain pointcut definitions and aspect instantiations. An illustration of a complete weaver program unit containing two pointcuts and one aspect instantiation can be seen below.

```
weaver Container_Composition is
  Append_Container_PC : Pointcut :=
    execution Vectors.Append*
      (in out Container; ..);

  Modify_Container_PC : Pointcut :=
    Append_Container_PC or
    execution Vectors.Insert*(..) or
    execution Vectors.Delete*(..);

  aspect Vector_Modification_Counter is
    new Counter_Aspect
      (To_Count => Modify_Container_PC);
end Container_Composition;
```

2.6 AspectAda vs. AspectJ

AspectJ is the benchmarking aspect-oriented language extension. AspectJ has evolved in several years and is a popular open-source product with many contributors. AspectAda is in its first version an only implement the basic features of AspectJ that are important concepts in terms of aspect-oriented programming. For reference we list capabilites AspectJ has that AspectAda does not have.

- AspectJ has cababilites to have joinpoint context exposed in the advice. As an example the parameters to the execution of joinpoint or the object the joinpoint is excecuted on can be accessed in the advice. This is powerful in order to build more domain specific aspects. As an example design-by-contract [13] could be modularized as aspects as opposed to Ada's pragma Assert statements.
- AspectJ supports aspect introductions where field, method or a parent interface is introduced into a class without the class itself being aware of the added field, method or interface. Introductions are static modifications of the behavior of the class.
- The pointcut expression language of AspectJ is richer than what currently exist in AspectAda. AspectJ supports a richer set of pointcut functions than AspectAda's call and execution pointcut expressions. Examples are set and get of object attributes, execution of exception handlers and filtering based on control flow of the context a joinpoint is executed.
- AspectJ also supports definition of compile time declarations that allows you to add compile-time warnings and errors. Aspects can define compile time warnings or errors to be issued based on a pointcut expression. An example is issuing an error if the Java code contains any call to System.out.println within the joinpoints designated by the pointcut.

3. CASE STUDY

3.1 Concurrency

Ada's language support for concurrent programming is one of the strengths in using Ada to build highly reliable concurrent programs. Concurrency is also one of the concerns that are well known to be crosscutting when studying it in aspect-oriented programming. Concurrent object-oriented programming is also known to suffer from the "inheritance anomalies" [12] problem. In this section we illustrate Ada's capabilities to implement a reliable version of the readers and writers problem and compare it to an implementation using AspectAda to modularize the crosscutting concerns. We also illustrate concurrent object-oriented programming in Ada, and compare it with an implementation where synchronization is modularized in an aspect.

3.1.1 Reader and Writers

A non shareable resource as a file should need a concurrency policy such that if one task is writing to the file, then no other process should be writing or reading. If, however, there is no writer process, then any number of processes should have read access [6].

A single protected type can be used to implement the readers and writers problem:

```
protected Shared_Data is
   function Read return Data_Item;
   procedure Write(New_Value : in Data_Item);
private
   The_Data : Data_Item := Default_Data;
end Shared_Data;


protected body Shared_Data is
   function Read return Data_Item is
   begin
      return The_Data;
   end Read;

   procedure Write(New_Value : in Data_Item) is
   begin
      The_Data := New_Value;
   end Write;
end Shared_Data;
```

Ada has indeed support for the readers and writers problem that makes it easy to implement. Compared to Java this is very elegant. However this simple approach has three drawbacks [3]

1. The programmer cannot easily control the order of access to the protected object; specifically, it is not possible to give preference to write operations over reads
2. If the read or write operations are potentially blocking, then they cannot be made from within a protected object. [3]
3. Protected types are not possible to extend as tagged types can be extended.

If we analyze the concerns of the readers and writers problem we can separate it into two concerns:

1. Business logic: the actual reading and writing of the file
2. Concurrency policy: which client should get access to the resource and what preference should be made between waiting readers and writers.

To overcome these difficulties a protected object must be used to implement a *concurrency control protocol* for the read and write operations (rather than encapsulate them.) The following code does this whilst giving preference to writes over reads:

```
package Readers_Writers is
  procedure Read(I : out Item);
  procedure Write(I : Item);
end Readers_Writers;


package body Readers_Writers is
  protected Control is
    entry Start_Read;
    procedure Stop_Read;
    entry Request_Write;
    procedure Stop_Write;
  private
    Readers : Natural := 0;
    Writers : Boolean := False;
  end Control;

  procedure Read_File(I : out Item) is
  begin
    Control.Start_Read;
       Read_File(I);
    Control.Stop_Read;
  end Read;

  procedure Write_File(I : Item) is
  begin
    Control.Request_Write;
    Control.Start_Write;
      Write_File(I);
    Control.Stop_Write;
  end Write;

  protected body Control is
    entry Start_Read when not Writers and
                    Request_Write'Count = 0 is
    begin
      Readers := Readers + 1;
    end Start_Read;

    procedure Stop_Read is
    begin
      Readers := Readers - 1;
    end Stop_Read;

    entry Request_Write when not Writers is
    begin
      Writers := True;
    end Request_Write;

    entry Start_Write when Readers = 0 is
    begin
      null;
    end Start_Write;

    procedure Stop_Write is
    begin
      Writers := False;
    end Stop_Write;
  end Control;
end Readers_Writers;
```

The entry protocol for a writer requires two steps; the first waits until there are no further writers and then sets the writers flag to true. This will stop any further readers. When all current readers have exited (and readers count is zero) the writer can enter. (The use of ‘Count on the Start_Read barrier ensures that waiting writers are given preference. This implements that writers have preference. The above protocol does solve the problems outlined. However the solution is not very robust to unexpected situations as exceptions thrown or very long writes. In order to develop higly reliable software we need to make it robust to unexpected situations.

3.1.1.1 Making a Robust Readers and Writers Algorithm

The first extension to be made is to handle exceptions propagated from the file I/O operations. These should be caught and the protocol should transfer to a state to not end in a deadlock.

```
 procedure Read_File(I : out Item) is
 begin
   Control.Start_Read;
   Read_File(I);
   Control.Stop_Read;
 exception
   when others =>
     Control.Stop_Read;
     raise READ_FAILED;
 end Read;

procedure Write_File(I : Item) is
begin
  Control.Request_Write;
  Control.Start_Write;
  Write_File(I);
  Control.Stop_Write;
exception
  when others =>
    Control.Stop_Write;
    raise WRITE_FAILED;
end Write;
```

The second extension to be made is to handle an I/O operation that does not return. This can be handled by setting an upper time bound on the operation using a select statement.

```
procedure Read_File(I : out Item) is
begin
  Control.Start_Read;
  select
    delay 10.0;
    raise READ_TIMEOUT;
  then abort
    Read_File(I);
  end select;
  Control.Stop_Read;
exception
  when READ_TIMEOUT =>
    Control.Stop_Read;
    raise;
  when others =>
    Control.Stop_Read;
     raise READ_FAILED;
end Read;


procedure Write_File(I : Item;
                     Failed : out Boolean)
is
begin
  Control.Request_Write;
  Control.Start_Write;
  select
    delay 10.0;
    raise WRITE_TIMEOUT;
  then abort
     Write_File(I);
  end select;
  Control.Stop_Write;
exception
  when WRITE_TIMEOUT =>
    Control.Stop_Write;
    raise;
  when others =>
    Control.Stop_Write;
    raise WRITE_FAILED;
end Write;
```

The above solution is much more robust than the initial solution using a protocol to control the readers and writers. In some situations the protocol could need to be made even more robust. The protocol can allow aborting readers that take long time in preference to writers (using a nested ATC.) Another extension can be to properly handle client tasks that abort their read or write operation.

This illustrated that in order to overcome one or more of the limitations of the Ada protected type for the readers and writers problem, a protocol and specific robustness code is needed to be added. The protocol and the additional code tangle the actual business logic that is to read and write to a file. The advanced readers and writers problem is an example of a concern that Ada cannot modularize well and is crosscutting. In the above example it is crosscutting with code tangling, but code scattering is not too bad since it is only a single Read_File and a single Write_File procedure. Next we will se how AspectAda could be used to modularize the crosscutting concern from the robust readers and writers problem.

3.1.1.2 An AspectAda Approach to the Problem

AspectAda has the capability to modularize crosscutting concerns. We would like to separate the concerns of the readers and writers problem into two concerns: one that is the domain of reading and writing the file and the other is in the concurrency policy domain:

The pure business logic concern would in Ada be similar to the simple solution with the protected type. However, as we have discussed there are limitations of the protected type in Ada. The modularization of the business logic only is shown below using Ada and two subprograms.

```
procedure Read(I : out Item) is
begin
  Read_File(I);
end Read;

procedure Write(I : Item) is
begin
  Write_File(I);
end Write;
```

However only having the business logic will not provide the concurrency policy needed for the problem. We use an aspect to modularize the concurrency policy concern. The aspect is generic taking a pointcut for the reading pointcut and one for the writing pointcut. The semantics are different for the protocol with regard to if it executes a Read joinpoint or a Write joinpoint. We use two around advices to express the actions for the concurrency policy protocol. Each of the around advice is connected to one of the generic formal pointcuts. Around advices provides mechanisms both to execute the business logic in a select statement and to catch exceptions to avoid protocol deadlocks. The aspect specification also contains the declaration of the protected type that realizes the concurrency policy. The declaration of the aspect type declares the protected type realizing the concurrency policy as a component and the protected object is thus accessible to both advices.

```
generic
  Readers_Pointcut : Pointcut;
  Writers_Pointcut : Pointcut;
aspect Readers_Writers_Writer_Pref is
  protected type Control_Protocol is
    entry Start_Read;
    procedure Stop_Read;
    entry Request_Write;
    procedure Stop_Write;
  private
    Readers : Natural := 0; -- # current readers
    Writers : Boolean := False; -- Writers present
  end Control;

  type Readers_Writers_Aspect is
    new Aspect_Ada.Simple_Aspect with
    record
      Control : Control_Protocol;
    end record;

  advice Around_Reader
    (RW_Aspect : in out Readers_Writers_Aspect);
  for Around_Reader'Pointcut use Readers_Pointcut;




  advice Around_Writer
    (RW_Aspect : in out Readers_Writers_Aspect);
  for Around_Writer'Pointcut use Writers_Pointcut;

end Readers_Writers_Writer_Pref;
```

The body of the protected type realizing the concurrency policy protocol is equal to what was seen inside the business logic of the Ada package in the non aspect-oriented version. The around advices are executed instead of the joinpoint inserting behavior both before and after actually executing the joinpoint itself using the proceed statement. Around statements is used since they have support for exception handlers catching exceptions thrown from the joinpoint execution.

```
aspect body Readers_Writers_Writer_Pref is
  protected body Control is
    entry Start_Read when not Writers and
            Request_Write'Count = 0
    is
    begin
      Readers := Readers + 1;
    end Start_Read;

    procedure Stop_Read is
    begin
       Readers := Readers - 1;
    end Stop_Read;

    entry Request_Write when not Writers is
    begin
      Writers := True;
    end Request_Write;

    entry Start_Write when Readers = 0 is
    begin
      null;
    end Start_Write;

    procedure Stop_Write is
    begin
      Writers := False;
    end Stop_Write;
  end Control;


  advice Around_Readers
    (RW_Aspect : in out Readers_Writers_Aspect)
  is
  begin
    RW_Aspect.Control.Start_Read;
    select
      delay 10.0;
      raise READ_TIMEOUT;
    then abort
      Proceed; -- Call the actual join point
    end select;
    RW_Aspect.Control.Stop_Read;
  exception
    when READ_TIMEOUT =>
      RW_Aspect.Control.Stop_Read;
      raise;
    when others =>
      RW_Aspect.Control.Stop_Read;
      raise READ_FAILED;
  end Around_Readers;

  advice Around_Writers
    (RW_Aspect : in out Readers_Writers_Aspect)
  is
  begin
    RW_Aspect.Control.Request_Write;
    RW_Aspect.Control.Start_Write;
    select
      delay 10.0;
      raise WRITE_TIMEOUT;
    then abort
      Proceed; -- Call the actual join point
    end select;
    RW_Aspect.Control.Stop_Write;
  exception
    when WRITE_TIMEOUT =>
      RW_Aspect.Control.Stop_Write;
      raise;
    when others =>
      RW_Aspect.Control.Stop_Write;
      raise WRITE_FAILED;
  end Around_Writers;
end Readers_Writers_Write_Pref;
```

The above aspect is a reusable aspect. It can be used for implementation of robust readers and writers protocol without knowledge of what the shared resource is. The aspect does not contain any direct reference to the joinpoints it should apply to, only that it expects two pointcuts when instantiated. In AspectAda aspects are associated with the business logic through the weaving/compositon rules. The rules define the appropriate pointcuts for the system, and instantiates the aspects to be used for the crosscutting concerns.

```
weaver My_Rules is
  My_File1_Get_PC : Pointcut :=
    execution(File_Reader1.Get*(..));

  My_File1_Set_PC : Pointcut :=
    execution(File_Reader1.Set*(..));

  aspect File1_Reader_Writer_Protocol is
    new Readers_Writers_Writer_Pref
      (Readers_Pointcut => My_File1_Get_PC,
       Writers_Pointcut => My_File1_Set_PC);
end My_Rules;
```

3.1.2 Object oriented concurrent system

Integrating object-orientation with concurrency is difficult. Ada95 successfully introduced object oriented for the sequential part of the language. However, Ada95 did not directly support object-orientation for protected types and tasks. The combination of object-orientation with mechanisms for concurrent programming are known to give raise to the so-called "inheritance anomaly" [12] that refer to the problems arising by coexistence of inheritance and concurrency in concurrent object-oriented languages. Proposals have been made for how Ada95 can be extended with support for object-oriented concurrent programming. One of these is the extensible protected types [17]. This solution centers around the notion of an extensible (tagged) protected type. Another proposal is to use inheritance of interfaces for protected and task types [16]. This solution is simpler since it does not involve inheritance of code or data for tasks and protected types, only interface inheritance.

Aspect-oriented programming provides means for separation of concerns and is one of the innovative approaches to fight inheritance anomaly. We illustrate AspectAda's capabilities to deal with inheritance through a simple example of *synchronization provided by the base (root) type* [3]:

3.1.2.1 Ada95 Synchronized Tagged Type

The mix of concurrency with extensible types requires the tagged type to implement a concurrency protocol. The concurrency protocol controls the access to the state of the object that is access through the objects methods.

```
package Protected_Objects is
  type Protected_Type is abstract limited private;

  procedure Op1(O : in out Protected_Type);
  procedure Op2(O : in out Protected_Type);
private
  type Obj_Type is tagged limited
    record
      ....
      L : Mutex;
    end record;
end Protected_Objects;
```

The base type defines a Mutex in terms of a protected object. The Mutex is implemented to lock and unlock the object to control access to the shared state. In this example we assume an implementation of the protected object that allows the same task to acquire the lock more than once. This is known as a reentrant lock. A more efficient implementation could be to use the readers and writers protocol from previous case study to allow multiple readers at the same time.

```
procedure Op1(O : in out Protected_Type) is
begin
 O.L.Lock;
 -- perform operation
 O.L.Unlock;
exception
  when others =>
    O.L.Unlock;
    raise;
end Op1;
```

Locking is performed when entering and exiting the every procedure of the object. Unexpected errors in terms of exceptions need to unlock the lock to implement liveness. The unexpected errors are handled in the exception block which unlocks the lock and reraises the exception.

A type extension can be made that calls the operations of the parent type:

```
package Protected_Objects.Extended is
  type Extended_Type is
    new Protected_Type with private;

  procedure Op1(O : in out Extended_Type);
private
  type Extended_Type is new Protected_Type with
    record
      ...
    end record
end Protected_Objects.Extended;
```

The body of the overridden Op1 procedure performs some manipulation of data and forwards some of the processing to the parent types method. The overridden method manipulates data of the object and need to obtain the lock before manipulating the data. The parent type's Op1 method also requests the lock and is allowed since it is the same task that requests the lock which supports reentrant locking.

```
procedure Op1(O : in out Extended_Type) is
begin
  O.L.Lock;
   -- potentially some extra manipilation of data
  OP1(Protected_Type(O));
  -- potentially some extra manipulation of data
  O.L.Unlock;
exception
  when others =>
    O.L.Unlock;
    raise;
end Op1;
```

The solution to object-oriented concurrent programming illustrates some of the effect of not having integrated protected types with object-orientation in Ada95. The synchronization concern is not well modularized since the protected object cannot be used as the base object in an inheritance hierarchy. The case study have the two symptoms of crosscutting concerns: code scattering and code tangling. The lock and unlock calls are scattered out on all subprograms of the objects in the inheritance hierarchy. The business logic of the objects is tangled with code to lock and unlock and the special exception handler to handle exceptions thrown.

3.1.2.2 AspectAda Synchronization Aspect

AspectAda has the capability to modularize the synchronization concern for tagged types inside aspects. We declare a new generic aspect with one generic formal pointcut parameter, a new aspect type and one around advice. Compared to the readers and writers concurrency policy aspect this aspect has only one generic formal pointcut parameter since there are no difference between readers and writers using the simple mutual exclusion protocol. The aspect type has a component for the protected type implementing the mutex, and the single around advice has an advice clause to associate it with the generic formal pointcut parameter.

```
generic
   Procedure_PC : Pointcut;
aspect Reentrant_Locks is
  type Lock_Aspect is new Simple_Aspect with
    record
      L : Mutex;
    end record;

  advice Around(A : in out Lock_Aspect);
  for Around'Pointcut use Procedure_PC;
end Reentrant_Locks;
```

```
aspect body Reentrannt_Locks is
  advice Around(A : in out Lock_Aspect) is
  begin
    A.L.Lock;
    Proceed;
    A.L.Unlock;
  exception
    when others =>
      A.L.Unlock;
      raise;
  end Around;
end Reentrant_Locks;
```

The aspect defined modularizes the synchronization concern using the mutex defined for the aspect type. An around advice is used for this aspect too, since an exception block is needed to be able to unlock the lock in case of exceptions.

The association of the pointcut to the generic aspect is done in the weaving/composition rules program unit. We define a pointcut that designate the subprograms declared for the tagged type Protected_Type and for the Extended_Type that extends the Protected_Type. Finally the aspect Reentrant_Lock_Object is instantiated with this named pointcut.

```
weaver My_Rules is
   My_Hierarchy : Pointcut :=
      execution(Protected_Objects.*(..) and
      execution(Protected_Objects.Extended.*(..);

  aspect Reentrant_Lock_Object is
    new Reentrant_Locks(My_Hierarchy);
end My_Rules;
```

The result of using AspectAda is that the Ada program contains business logic in the methods declared for the objects and the aspect contains the synchronization logic and these two concerns are associated using the weaver/composition rules. We illustrate this by showing the object methods for the aspect-oriented system.

```
procedure Op1(O : in out Protected_Type) is
begin
 -- perform operation
end Op1;
```

We see that the business logic is not tangled with synchronization logic. The same applies to the overriding method:

```
procedure Op1(O : in out Extended_Type) is
begin
   -- potentially some extra manipilation of data
  OP1(Protected_Type(O));
  -- potentially some extra manipulation of data
end Op1;
```

The removal of the synchronization code of the object methods will be true for all methods. Hence the scattering of the synchronization code is also removed. We can based on this conclude that ApsectAda has managed to modularize the crosscutting concern removing code tangling and code scattering.

4. WEAVER IMPLEMENTATION

4.1 Architecture

The prototype of a compiler for the AspectAda language is a preprocessor like compiler that takes Ada95 code, aspect code, and weaving/composition rules to generate weaved source code. The generated source code is compiled, bound and linked with a standard Ada95 compiler to an executable. AspectAda also has a runtime component for accessing static information of a join-point (subprogram name, parameter names etc.) The *conceptual design view* of the AspectAda compiler is shown in **Figure 1** using the architectural notation from [9].

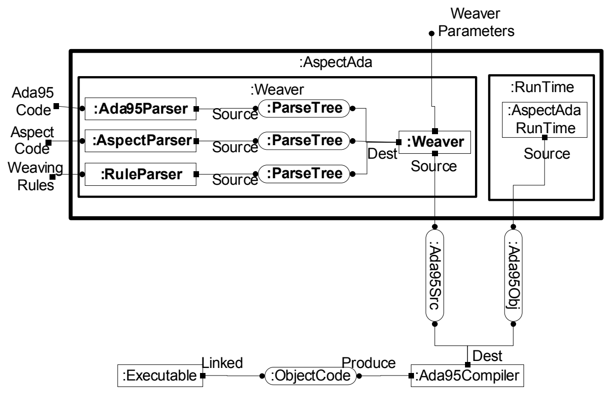


Figure 1. Conceptual Architecture

The AspectAda tool is separated into two subsystems: a weaver and a run-time. The weaver subsystem realizes the executable for the AspectAda tool. The run-time is a library that is used by the produced code to access information about the current joinpoint.

The input to the weaver subsystem (external ports) is of three types: Ada95 source code, aspect source code and composition rules. Additionally the weaving can be configured through a set of weaver parameters. The three parsers (components) are connected to the weaver through three separate parse trees (connectors). The parse trees can be traversed and queried from the weaver. The weaver is performing the merging of aspects to the Ada95 source code and is producing Ada95 source code. The produced source code has source code dependency to the AspectAda run-time library and is thus bound and linked with the object code of the run-time library. The compilation, binding and linking of the generated source code is achieved using a standard Ada95 compiler that produces object code and an executable.

The module architecture of the AspectAda compiler reflects the conceptual architecture. The two subsystems, weaver and runtime, are also modular subsystems. The majority of functionality is in weaver subsystem. To support the parsing of the input languages we selected to use two third party libraries: AdaGOOP and ASIS. AdaGOOP [4, 5] is a flex/yacc like parser generator that supports generation of an object-oriented syntax tree with the visitor design pattern [8]. AdaGOOP is used for parsing the aspect source code and the weaving/composition rules. The aspect source code is complete Ada95 grammar with extensions to recognize advices. The weaving/composition rules contain pointcut definitions and aspect definitions. The grammar for this is a mix of AspectJ grammar with a touch of Ada95 like grammar. Ada95 source code parsing is done using the Ada Semantic Interface Specification (ASIS) [10]. ASIS is an ISO standardized interface that provides semantic and syntactic information to the clients. The weaving module needs to select the appropriate joinpoints that are designated by the pointcuts in the Ada95 source code and to merge advices with Ada95 code to produce weaved source code.

The modular architectural view (see Figure 2) shows the layering and subsystems. The parsing of AspectAda's language extensions are encapsulated in two separate subsystems: Aspect Parser and pointcut parser. Each subsystem uses a separate grammar and produces a separate annotated parse tree. The JoinPointModel layer is the internal representation of joinpoints, pointcuts, aspects and advices. The representation of these concepts is abstracted from the actual syntax and semantics to separate AspectAda language from the aspect-oriented model. Having such an abstract internal representation is good to hide the underlying language, since the concrete syntax of AspectAda grammar is expected to evolve. Above the JoinPointModel is the layer that first performs selection to populate the joinpointmodel with the designated joinpoints and the merging of the advices to the designated joinpoints. Both the selection subsystem and the merging subsystem use ASIS. ASIS is wrapped with a thin layer to have object-oriented interface for the features used in selection and merging. The selection subsystem use an ASIS iterator to iterate over Ada compilation units, when the node that is iterated is an AspectAda joinpoint a pattern matching with all the pointcuts is performed. After the iteration of all compilation units is performed the pointcut expressions are evaluated using the conditional operators. This will generate a collection of joinpoints selected by a named pointcut based on the pointcut expression composing the named pointcut. The merging subsystem also utilizes ASIS for the merging of advices to the selected joinpoints. The top layer is the controller that has a command line interface and can be integrated with an integrated development environment (IDE) like the GNAT Programming System (GPS)

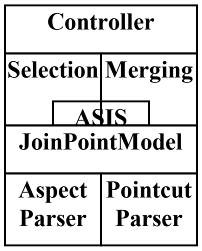


Figure 2. Weaver Layering and Subsystems

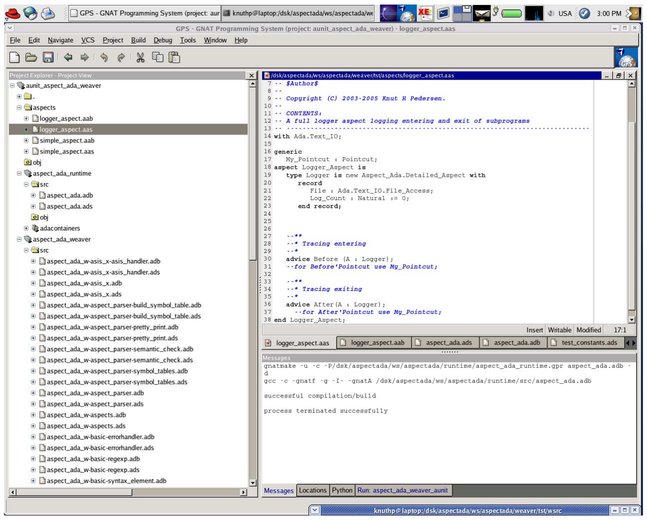
The use of ASIS and AdaGOOP was beneficial. The parsing was simplified. As a criticism of ASIS we would like to mention that ASIS does not provide an easy interface to implement this kind of merging. Only iteration of the abstract syntax tree is possible and the abstract syntax tree does not contain all terminal symbols. We believe the possibility to iterate a concrete syntax tree with terminal symbols would have made the merging subsystem easier to implement.

4.2 AspectAda Generated Code

The AspectAda prototype performs a simple merging. The advice code is inserted straight into the joinpoint location. This produces efficient code for the executable system. The tradeoff using this approach is that changes in an advice body imply that all source code needs run through the AspectAda compiler once more. In other words the separation of aspect specifications and aspect bodies has no effect of compile time performance. An alternative solution would be to transform the advices to subprograms that where called from the joinpoint. With such an approach a change in the aspect body would only result in recompilation of the subprogram simulating the advice. Such a solution would give additional run-time overhead since a subprogram call had to be done for every advice to be executed.

4.3 Integration with GPS

Part of the success of AspectJ as an aspect-oriented programming language is its tool support in the popular Eclipse IDE. The GNAT programming system (GPS) is a popular open source IDE for the Ada programming language. GPS can be extended to support new languages and tools. AspectAda has developed a plugin for GPS that integrates AspectAda with GPS. Aspect source files and weaving/composition rule files are recognized and are part of the GPS project view.


GPS - GNAT Programming System (project: aunit_aspect_ada_weaver) - logger_aspect.aas
File Edit Navigate VCS Project Build Debug Tools Window Help
/dsk/aspectada/ws/aspectada/weaver/tst/aspects/logger_aspect.aas
Insert Writable Modified 17:1
logger_aspect.aas
logger_aspect.aab
aspect_ada.ads
aspect_ada.adb
test_constants.ads
Messages Locations Python Run: aspect_ada_weaver_aunit
knuthp@laptop:/dsk/aspectada/ws/aspectada/weaver/tst/wsrc

```
-- $Author$
--
-- Copyright (C) 2003-2005 Knut H Pedersen.
--
-- CONTENTS:
-- A full logger aspect logging entering and exit of subprograms
-- -----------------------------------------------------------------------
with Ada.Text_IO;

generic
   My_Pointcut : Pointcut;
aspect Logger_Aspect is
   type Logger is new Aspect_Ada.Detailed_Aspect with
      record
         File : Ada.Text_IO.File_Access;
         Log_Count : Natural := 0;
      end record;



   --**
   --* Tracing entering
   --*
   advice Before (A : Logger);
   --for Before'Pointcut use My_Pointcut;

   --**
   --* Tracing exiting
   --*
   advice After(A : Logger);
     --for After'Pointcut use My_Pointcut;
end Logger_Aspect;
```

```
gnatmake -u -c -P/dsk/aspectada/ws/aspectada/runtime/aspect_ada_runtime.gpr aspect_ada.adb -
d
gcc -c -gnatf -g -I- -gnatA /dsk/aspectada/ws/aspectada/runtime/src/aspect_ada.adb

successful compilation/build

process terminated successfully
```

Figure 3. AspectAda integrated with GPS

5. EFFECTS ON SOFTWARE QUALITY

Building high quality software is a core concern of every software company, and for the users of Ada it is one of the main reasons for choosing Ada as opposed to other languages. A main objective with aspect-oriented programming is that it improves software quality for systems with crosscutting concerns. This section provides reasoning about how AspectAda as a language extension and as a tool affects software quality of design and implementation based on the case study.

The main benefit of aspect oriented programming is improved support for separation of concerns. Separation of concerns [7, 14] is a well known sound software engineering practice for quality design. The success of object-oriented programming is largely due to the structured way it enables to separate the concerns. Aspect oriented programming takes the separation of concern one step further by providing mechanisms for separation and modularization of crosscutting concerns. Measurement of how well a concern is separated is done using two qualitative criteria: cohesion and coupling. Cohesion is a measure of the relative functional strength of a module. Coupling is a measure of the relative interdependence among modules. High cohesion and low coupling indicates high quality design.

5.1 Coupling and cohesion

We start the analysis with the cohesion of the readers and writers problem. Previously we analyzed the readers and writers problem to be composed of two concerns: The first concern is to read and write the file (business logic) and the second concern is to handle the concurrency policy crosscutting concern. If a module performs a single task, the module has high cohesion. The more tasks the module performs the lower the cohesion of the module is. In the Ada95 implementation of the readers and writers problem the Ada package had two concerns resulting in a module implementing two tasks. In the aspect-oriented solution the two concerns where separated into two modules. One module was the Ada95 code to read and write the file, and the other was the advices that enforced the concurrency policy. The AspectAda implementation of the readers and writers allow having two modules that each performs a single task, hence the cohesion is higher. However, the separation into two modules results in a coupling between the aspect and the Ada95 package. One could think that this new coupling was an undesired effect of the aspect extraction, but in fact two modules cannot collaborate to perform the desired functionality without a minimal coupling. In aspect-oriented programming the coupling is kept at this minimum. In addition aspect-oriented programming has the specialized pointcut language to express the coupling between aspects and Ada95 components. We think the quality measured in terms of coupling and cohesion was improved using AspectAda to separate the concerns for the readers and writers problem.

Analyzing the object-oriented concurrent system from the case study gives a similar result with regards to cohesion. The objects perform two tasks: The task of the object and the task to synchronize access to the shared data. A similar analysis as for the readers and writers problem is applicable for the improvement of cohesion of the object-oriented concurrent system. The object-oriented concurrent system has in addition scattering of the lock over all methods accessing the shared data. Hence the system has high coupling from the business logic object to the lock methods of the mutex. By using AspectAda we extract this coupling and place it into the weaver/composition rules. With this aspect extraction both the business logic component and the lock aspect can be easily reused, in order to reuse the aspect or business logic, only the weaving/composition rules need to be changed/re-written. We think AspectAda has improved the reusability of business logic in the object and synchronization logic in the aspect.

5.2 Adaptability

Flexibility and adaptability has been sought by many researchers and developers. In there effort several different approaches have been invented to make software more flexible and adaptable. Examples of some are pattern and framework approaches, component based approaches, reflection approaches. Ada as well provides several means to support development of more adaptable software and the most significant is the use of generics. Aspect-oriented programming as well provides means to improve adaptability, where the main improvement is the separation of concerns. By separating and modularizing concerns in aspects and weaving/composition rules it is easier to adapt the business logic to use another policy for one or more of the crosscutting concerns. One example is adapting the logging in the logger example. Adapting the aspect-oriented logging to alter to another logger interface is easy, since the logging code is placed only one place, in the aspect. Adaptability of the Ada95 implementation depends more on the interface of the new and old logger. If the parameters to the logger are compliant only the body of the logger can be modified to forward to the new logging interface. However if the parameters are not compliant changes has to be done in all places where the logger is called.

Another example of improved adaptability in the aspect-oriented system is to change the synchronization for the object-oriented concurrent system. The case study uses a simple mutex, while in another system there could be a need for performance improvements, and a read/write lock would be preferred. In the Ada95 system such a change would require to modify the object locking to use a readlock or a writelock in the methods depending on if they change the shared state or not. Adapting this in the aspect-oriented solution would require developing a read-/writelock aspect. This aspect would be similar to the aspect for readers and writers problem in the case study. The Ada95 code for the object-oriented concurrent system would involve changes to all methods performing mutual exclusion. In addition the object code would now either use mutual exclusion or reader/writer synchronization, and hence the code would need to be maintained in two branches depending on synchronization.

From this reasoning we see that aspect-oriented programming's capability to separate crosscutting concerns has impact on the adaptability of the system. Adaptability related to the crosscutting concerns are easy to realize and the business logic can be left untouched if the policy for the crosscutting concern is changed.

5.3 Code Comprehensibility

The code comprehensibility of software is important. In general software is written once, but read several times, especially during maintenance. Code comprehensibility is difficult to quantify. Comprehensibility is a matter of taste of the reader. Some like big functions so they can read line by line, others like smaller methods that abstract part of the implementation as in object-oriented programming. As an effect of this what is a good reading technique for structured programming might not be a good technique for object-oriented programs. Reading object-oriented programs from the first to last line to understand the system is difficult, but for structured programs this technique is indeed good.

Aspect-oriented programming has two important effects on the comprehensibility. First business logic is not tangled with code to realize a crosscutting concern. Second the crosscutting concern and the business logic is associated through weaving/composition rules. Not having tangled code is good, since it is easy to find what the business logic is. However, this has a cost, a package or a set of packages cannot be read separately to reason about the behavior. Aspects and the composition rules might modify the behavior completely. An example is an around advice not calling the actual joinpoint with the proceed statement. To comprehend the collaboration between advices and joinpoints the support of an aspect aware IDE is invaluable. Without tool support to trace from a pointcut to the designated joinpoints comprehending the effect of aspects is difficult. As an effect comprehending aspect-oriented systems needs adapting new techniques on how to read the code. We think that comprehensibility of aspect-oriented programs depends on tool support, without toolsupport comprehensibility of the effect of advices is complex. On the other hand the comprehensibility of the business logic is improved, so there are a trade-off people has to perform when evaluating if they should use aspect-oriented programming.

5.4 Traceability

Software traceability has been recognized as a significant factor of efficient software project management and software systems quality. The traceability from the requirements through the design into the resulting code is an important part of the overall software traceability. This traceability is provided by mapping concerns in the problem space to the solution space. High quality traceability would allow a seamless move from the concern in the problem space to a small set of modules in the solution space. In some cases the feature from the problem space is mapped to a large set of modules in the solution space, and the feature is scattered on a large set of modules. Traceability of such features can be improved if it involved a smaller set of modules in the solution space. The reason that the mapping of the feature results in a large set of modules is sometimes an indication of bad software design and can be fixed by iterating on the design itself. However in some cases the feature cannot be modularized using the language constructs of the implementation language. For such features aspect-oriented programming is a good solution to improve traceability. An illustration of this is the logger. Initially logging where scattered out on a large number of modules. However, when implemented using AspectAda logging were modularized into one aspect and its corresponding aspect instantiation. The impact on the traceability is that the logging feature can now be traced only to these two modules. (It should be noticed that the concern still affects a large number of modules, but the complexity of the mapping is managed by the AspectAda weaver tool as opposed to the developer.)

Traceability from the solution space to the problem space is also important in the software lifecycle. Good traceability here would be a module with high cohesion, and only implemented one feature. The case study illustrates that the readers and writers problem had three concerns: business logic of reading the file, synchronization to always have consistent shared data and scheduling to controlling the policy of access among the users of the shared data.

6. FUTURE WORK

AspectAda language is in an early phase. To achieve further support for separation of concerns we have identified areas of future research. Some of these are:

- AspectAda does not provide language support to expose the objects and types at the joinpoint to the advice. The advice has only access to metadata like name, and parameter mode, type and name. AspectJ has language support to expose objects, and arguments at the joinpoint. Adding this support would greatly increase the power of the AspectAda. As an example design by contract could be done using advices to modularize the contract.
- Research is done to have joinpoints with finer granularity than execution or call of subprograms and entry points etc. Examples of such joinpoints are loop statements or even single lines of code. Ada supports the concept of named block statements that could be possible to designate through the AspectAda pointcut expression language.
- Ada has the notion of generic packages, procedures and functions. AspectAda need to define exactly what a joinpoint in a generic unit is. It could either be the generic subprogram or it could be the instance of the generic subprogram.
- Inheritance is known to increase reusability. Aspect types in AspectAda are tagged types and could be extended. Research needs to be done to design the language to use aspect type extension of user defined aspects to increase reusability of aspects.
- The success of AspectJ relies on tool support, and especially IDE support. AspectAda need to continue the integration with GPS to support complete weaving and compilation of executables. Also browsing capabilities to visualize which advices applies to a joinpoint and which joinpoints are designated by a pointcut eases the task of the developer, maintainer and the code reviewer in terms of comprehensibility.

7. CONCLUSION

We have presented an aspect-oriented language extension to Ada95 and a prototype compiler for the language extensions. Based on a small case study we illustrated the effect this language extension is expected to have on software quality. From our case study we found that Ada with its support for concurrent programming can modularize several of the concerns that are crosscutting in other languages like Java. However, for concurrency we also identified some limitations for Ada where AspectAda provides improvements to separate crosscutting concerns. Concurrency is not the only crosscutting concern, and for the logger AspectAda provides separation of concerns that are not possible using Ada95.

The prototype implementation of an AspectAda compiler is still in an early stage. The compiler cannot yet deal with all AspectAda language constructs in all possible contexts. However the clear architecture of the compiler and the powerful support it has of ASIS ease further experiments with the aspect-oriented extension of Ada. As seen from the future work section the AspectAda language is still under development and we are open for suggestions on how to further optimize the syntax and semantics.

8. REFERENCES

[1] ARG *Container Library, Ada Issue 302*, http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-20302.TXT

[2] The AspectJ Team. *The AspectJ Programmers Guide*. http://www.eclipse.org/aspectj

[3] Burns, A., Wellings, A.. *Concurrency in Ada: 2nd edition*, Cambridge University Press, 1998

[4] Carlisle, M. C, *An Automatic Object-Oriented Parser Generator for Ada*, Ada Letters, Vol XX, Number 2, June 2000

[5] Carlisle, M. C., Sward, R.E. *An Automatic "Visitor" Generator for Ada*, Ada Letters, Vol XXII, Number 3, September 2002

[6] Courtois, P.J., Heymans, F., Parnas, D.L.: *Concurrent Control with "Readers" and "Writers"*, CACM October 1971

[7] Dijkstra E.W. *A Discipline of programming*, Prentica Hall, Englewood Cliffs, NJ, 1976

[8] Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addision-Wesley, Reading, MA, 1995

[9] Hofmeister C., Nord, R., Soni, D. *Applied Software* Architecture. Addison Wesley, 2000.

[10] ISO/IEC 15291:1999 Information technology -- Programming languages -- Ada Semantic Interface Specification (ASIS)

[11] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J-M., Irwin, J. *Aspect-Oriented Programming*, ECOOP'97 Conference proceedings, LNCS 1241, June 1997, pp. 220 – 242.

[12] Matsuoka, S. and Yonezawa, A. *Analysis of inheritance anomaly in object-oriented concurrent programming languages*. In Research Directions in Concurrent Object-Oriented Programming. MIT Press, 1993

[13] Meyer, B. *Applying Design by Contract*. IEEE Computer 25(10): 40-51 (1992)

[14] Parnas, D. L. *On the Criteria to be used in Decomposing Systems into Modules*. In Communications of the ACM. Vol. 15. No. 12. December 1972, pp. 1053-1058.

[15] Spinczyk, O., Gal, A., Schröder-Preikschat, W. *AspectC++: An Aspect-Oriented Extension to C++*, Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002) , Sydney, Australia, February 18-21, 2002

[16] Taft, S. T. *Object-Oriented Programming Enhancements in* Ada *200Y,* Ada User Journal, June 2003

[17] Wellings, A. J., Johnson, B., Sanden, B., Kienzle, J., Wolf, T., Michell, S. *Integrating object-oriented programming and protected objects in Ada 95* ACM Transactions on Programming Languages and Systems (TOPLAS) archive Vol 22 , Issue 3, May 2000, pp 506 – 539