Avoiding Incorrect and Unpredictable Behaviour with Attribute-based Crosscutting

Donal Lafferty
lafferty@engineer.com
Distributed Systems Group
Department of Computer Science
Trinity College Dublin

ABSTRACT

For consistency with component-oriented programming, the implementation of aspect-based software product families should be decoupled from the components that they influence. One solution is to implement such families with language-independent aspect-oriented programming (AOP) in combination with property-based crosscutting. Language-independent AOP decouples the implementation language of such a family from that of the components to which family members are applied. Using property-based crosscutting avoids coupling the software product family with a specific set of components, because join points are selected according to some implementation commonality such as a containing type name rather than precise details of their implementation. However, experimental evidence shows that compiling a component's implementation to a language independent format can introduce new join points that match a property-based crosscut. These matches can result in unexpected behaviour. To assure correct and predictable behaviour when a software product family member is applied to an application it is better to use attribute-based crosscutting, in which join points are selected according to the appearance of attributes, called annotations in Java, on the join point's implementation.

1. INTRODUCTION

The use of aspects to implement software product families has been demonstrated by work on spontaneous containers [13]. Here, different aspects are responsible for implementing persistence, transaction processing and access control properties of a container. Moreover, different implementations of each property can exist. These aspects can be assembled to create a container that enforces network policies applied to a particular network node. Such a product family might find practical application in tradeshow venues [13], where services vary according to the status of the attendee. The containers that allow visitor PDAs to interact with each other will differ from those provided to exhibitors. For instance, access control capabilities may be required by exhibitors that make use of a tradeshow registration database that is not available to visitors.

Language independent AOP and the use of property-based crosscutting should reconcile the implementation of aspect-based software product families with component-oriented programming. Software components emphasize deployment and composition characteristics that allow components provided by one organization to be combined with components of another by a third-party unrelated to either organization [14]. Specifically, "a third party is one that cannot be expected to have access to the construction details of all the components involved." [14]. Thus, implementation source code should not be a factor in the ability of an aspect-based software product family member to bind with components in an application. Language-independence addresses this requirement by allowing aspects and components to be written in a variety of languages and freely intermixed [9]. Using property-based crosscutting [6], an aspect selections join points in application according to some commonality such as a shared containing type, some naming convention, or common parameter types in the case of methods. Sufficiently general property-based crosscutting allows aspects to be used with a variety of components without the need to customize the aspect on a component by component basis. Using language independent AOP and property-based crosscutting should allow third parties to treat aspect-based software product families and application components being composed as black boxes.

The vehicle for testing this theory was Weave.NET [7], a language independent aspect weaver that supports a pointcut-advice mechanism [10] and allows clear-box crosscutting. Weave.NET targets Microsoft's Common Language Infrastructure (CLI) [3]. The CLI simplifies the task of implementing Weave.NET by providing a language-independent substrate to which components, regardless of implementation language, are compiled. The CLI standardizes the metadata descriptions of types and type members implemented by CLI components, regardless of implementation language, and component behaviour is written in a language-neutral binary format. With respect to the clear-box / black-box distinction [4], a black-box technique manipulates components in terms of their public interfaces, while a clear-box technique manipulates the parsed language structures used to write these interfaces. Clear-box techniques, such as those available with the pointcut-advice mechanism AspectJ [15], often offer a richer set of join points, because they provide a better representation of all the structures of a programming language used to write the component. In the case of components, language constructs that are expressed directly in byte code, such as accesses to type members, can be modified [4].

In our experiments, initially reported in [7], we applied members of a simple diagnostics software product family to a Fibonacci series enumerator, and we noted an inability to make strong assurances of correct and predictable behaviour of the resulting application. In our experiments, implementations of a recursive Fibonacci series enumerator algorithm written in SML.NET [5], VisualBasic.NET [12] and C# [11] were composed with logging and profiling functionality implemented using aspects. Using language independent AOP and property-based crosscutting, the same logging and profiling aspects could be applied to each Fibonacci enumerator component without change. However, we noted that in the case of the component implemented in

SML.NET it was not possible to predict the behaviour of the final application based on the source code of the Fibonacci algorithm. During the translation to a language neutral format, the SML.NET compiler introduced addition join points to the internal implementation of component interfaces that matched the logging aspect.

Rather than change to black box crosscutting, we avoided inadvertent join point matches using attributed-based crosscutting [7, 8]. The additional join points added during component compilation did not influence the component implementation, and so predictability could have been restored by limiting the weaver being used. However, it was possible to restore predictability with attribute-based crosscutting rather than by crippling the weaver. With *attribute-based crosscutting*, join point selection is written in terms of attributes. Attributes [3], also called annotations [1], offer a programming-language mechanism for associating additional information with the metadata descriptions of types and their members. Language support for attributes typically includes the ability to define new attributes and the ability to place attribute declarations along side the definition of types and their members. Annotation of a program element with an attribute causes additional data to become associated with the metadata description of that program element; however, annotating code with attributes does not influence program behaviour. For instance, an attribute might append the metadata of a method with the name of the programmer implementing the method. This information would be associated with the metadata description of the method. Rather than using the method name in a pointcut specification, the attribute name can be used instead. Since an attribute applied to source code appears only once in the compiled assembly, attribute-based crosscutting avoids inadvertent join point selection and thus results in more predictable behaviour.

In the remainder of this paper, we present the experimental results that identified problems with property-based crosscutting and that identified attribute-based crosscutting as a possible solution. In section 2, we summarize the composition scenario in terms of the software product family functionality being implemented and the components to which family members are being applied. In section 3, we present an implementation of the software product family based on property-based crosscutting solution and evaluate its drawbacks. In section 4, we do the same for an attribute-based crosscutting solution, and explain what drawbacks in the previous section are avoided. Finally, we summarize our findings in section 5.

2. EXPERIMENTAL SCENARIO

To evaluate the usefulness of property-based crosscutting in the context of language-independent AOP, we chose to use a software product family that provided application diagnostics. The software product family provides profiling and logging functionality features, and it is written using the aspect model provided by Weave.NET. One, the other, or both diagnostics features can be applied to methods of an application. However, in this paper we focus on the application of logging functionality to components implementing a common Fibonacci series algorithm. The algorithm enumerates series elements, and it was chosen based on the observation that Fibonacci algorithms are a common means of demonstrating AOP techniques [2].

2.1 Target Application

The specific algorithm targeted for our language-independence tests is a component implementing a recursive algorithm that enumerates members of the Fibonacci series. The algorithm, shown in Figure 2.1, includes two methods, one that generates elements in the Fibonacci series, and a second that reports a series of elements generated using the former method. Components containing these methods have been written in C#, VB.NET and SML.NET. The C# version shown in Figure 2.1 is typical of the algorithm, which is recursive regardless of the programming language used.

```
public class FibonacciSeries {
  public void FibSeries(int seriesLen){
    for (int i = 0; i<= seriesLen; i++) {
      long result = Fibonacci(i);
      System.Console.WriteLine("Element\t"+
                      i+ "\tvalue \t"+result);
    }
  }

  public long Fibonacci(int n){
    if (n > 1)
      return this.Fibonacci(n-1)
            + this.Fibonacci(n-2);
    return 1;
  }
}
```

Figure 2.1: C# source for algorithm to enumerate Fibonacci series elements.

2.2 Test Aspect

Our Fibonacci algorithm lacks an explicit indication of its complexity, but this is remedied by adding diagnostics functionality that provides logging. This logging is implemented via an aspect that reports the start and end of execution join points. Logging is a fairly simple concept made simpler by limiting the aspect to reporting the start and end of a method execution to the application console rather than logging to a file. A sample implementation of logging behaviour written in SML.NET is shown in Figure 2.2. The method names in the source allude to the kind of advice they implement. The appearance of multiple methods with the prefix `LogAfterJoinPointXXX` corresponds to the use of different return types in the methods of the Fibonacci series algorithm.

3. LOGGING VIA PROPERTY-BASED CROSSCUTTING

Property-based crosscutting is consistent with language independent AOP, and it allows an aspect to be created that can be applied to various components without the changes to the crosscutting specification However, the crosscutting specifications suffer the drawback of being error prone. Also, obtaining assurances of correct program behaviour is difficult, as program behaviour cannot be determined from inspection of component source code.

When the logging aspect is written using property-based crosscutting, the crosscutting specification is the same regardless of the language implementing the Fibonacci algorithm with which logging is woven. The logging aspect's XML-based crosscutting specification is shown in Figure 3.1 and it defines a named pointcut called `SomeMethodExecution` that identifies method invocations that take an integer as a parameter, regardless of the

return type. The crosscutting specifications are reusable without modification in that they can be applied to components without change. Reuse then relies on the component's types being the same, in terms of members and member signatures, regardless of implementing language.

```
 structure Aspect_ML_Logging =
 struct
 _classtype Logger() : TCD.CS.DSG.Weave.Reflect.Aspect()
  with
   LogBeforeJoinPointInt (param:int) =
   let
    val                    jptInfo                    =
valOf(this.##get_JoinPointStaticPart());
   in
    print "Join point: ";
    print (valOf(jptInfo.#toShortString())); print "\n";
    print "Execution parameter: ";
    print (Int.toString(param)); print "\n"
   end
  and
   LogAfterJoinPointLong(param:int, result:Int64.int)=
   let
    val                    jptInfo                    =
valOf(this.##get_JoinPointStaticPart());
   in
    print "Join point: ";
    print (valOf(jptInfo.#toShortString())); print "\n";
    print "Execution parameter: ";
    print (Int.toString(param)); print "\n";
    print "Execution result:    ";
    print (Int64.toString(result)); print "\n"
   end
  and
   LogAfterJoinPointVoid (param:int) =
   let
    val                    jptInfo                    =
valOf(this.##get_JoinPointStaticPart());
   in
    print "Join point: ";
    print (valOf(jptInfo.#toShortString())); print "\n";
    print "Execution parameter: " ;
    print (Int.toString(param)); print "\n";
    print "Execution result:    NONE!"; print "\n"
   end
  end
 end
```

Figure 2.2: Implementation of logging behaviour written in SML.NET

During weaving trials in which the logging aspect was applied to each implementation of the Fibonacci algorithm, we noted that the specification of types in XML was error prone, because mapping from language-based type names to CLI type names must be done manually. Writing a custom crosscutting specification in XML involves using metadata descriptions to select join points. To make Weave.NET aspects language independent, the crosscutting specifications are specified in terms of CLI types, and not the development language types with which a programmer will be familiar. The need to map from development language types to CLI types is acute in the case of primitive types, whose CLI names vary considerably from those used in the source code of a component. For example, Table 3.1 shows the mappings between SML.NET primitive types, their C# equivalent and their CLR name. These tables show no overlap between the programming language type names and those used by the CLI.

```
 <item>
   <named_pointcut>
     <modifier><public/></modifier>
     <name>SomeMethodExecution</name>
     <local_var_ref>
       <var_type>Int32</var_type>
       <var_name>data</var_name>
     </local_var_ref>
     <pointcut>
       <and>
         <pointcut><primitive>
           <execution>
             <method_signature>
               <return_type>
                 <type_name>*</type_name>
               </return_type>
               <join_point_type>
                 <type_name>*</type_name>
               </join_point_type>
               <method_name>*</method_name>
               <parameters>
                 <parameter>
                   <type_name>Int32</type_name>
                 </parameter>
               </parameters>
             </method_signature>
           </execution>
         </primitive></pointcut>
         <pointcut><primitive>
           <args>
             <parameter>
<formal_parameter_name>data</formal_parameter_nam
e>
             </parameter>
           </args>
         </primitive></pointcut>
       </and>
     </pointcut>
   </named_pointcut>
 </item>
```

Figure 3.1: A pointcut identifying method execution join points to which logging is applied.

Table 3.1: Mapping between CLI (.NET) types and C# / SML.NET equivalents, taken from [5].

.NET type	C♯ type	SML.NET type
System.Boolean	bool	bool
System.Byte	byte	Word8.word
System.Char	char	char
System.Double	double	real
System.Single	float	Real32.real
System.Int32	int	int
System.Int64	long	Int64.int
System.Int16	short	Int16.int
System.SByte	sbyte	Int8.int
System.String	string	string
System.UInt16	ushort	Word16.word
System.UInt32	uint	word
System.UInt64	ulong	Word64.word
System.Exception	System.Exception	exn
System.Object	object	object

```
structure App_Noninvasive_ML_Fibonacci
 : sig val main: string option array option->unit
    end =
struct
_classtype FibonacciSeries()
 with
  Fibonacci (n) =
    case(n) of
      0 => (Int64.fromInt(1))
    | 1 => (Int64.fromInt(1))
    | n => (this.#Fibonacci(n-1) +
            this.#Fibonacci(n-2))
 and
  FibSeries (n) =
    case(n) of
     ~1 => ()
    | n => (this.#FibSeries (n-1);
            print "Element\t";
            print (Int.toString (n));
            print "\t value \t";
        print(Int64.toString(this.#Fibonacci(n)));
            print "\n" )
 end
 fun SelfTest (elements, times) =
  let
         val fibML = FibonacciSeries()
  in
         case(times) of
           0 => ()
         | n => (fibML.#FibSeries(elements);
                 SelfTest(elements, times-1))
  end
 fun main  (a : string option array option) =
  let
    val elements = 10
    val times = 1
  in
    SelfTest(elements, times)
  end
end
```

Figure 3.2: SML.NET implementation of application to calculate Fibonacci Series elements.

We did experiment with making it easier to simplify type specification by allowing the use of truncated versions of CLI type names in which the namespace is removed. Hence, the use of `Int32` rather than `System.Int32` in the XML of Figure 3.1. While these truncated versions are shorter to write, they make mistakes easier to make. For example, in writing "`System.String`", we found the capitalization of `System` to be a reminder to capitalise the '`String`' portion. When the namespace was removed, it was easier to forget that the CLI type was being used, and so we reverted to using language-specific monikers. For example, '`string`', all lower case, was used instead of '`String`' with the capital first letter. These mistakes are hard to spot, since it appears that the type is correctly written

Generally, user types present less difficulty, as their name and namespace holds across language boundaries. There are still quirks when user types are exported as nested classes. For instance, class types exported by SML.NET are nested in their respective module. Thus, a class `Logger` defined in module `ML_Logger` would be accessed using the moniker `Aspect_ML_Logger+Logger`.

Our evaluation also noted a severe problem with the accidental selection of join points that could not be overcome using source code analysis tools. In Figure 3.1, the regular expressions are used in the pointcut designator's argument to create a property-based crosscut. However, such regular expressions can make unexpected join point selections. Before evaluation, we made the general assumption that these extra join points could be spotted in source code. If this were the case, then with a careful examination of component source could be used to predict the behaviour of the final application and on the basis of this prediction correct behaviour could be ascertained. However, evaluation tests involving components written in SML.NET indicate superfluous join points are not always visible from source. Assemblies generated by the SML.NET compiler can contain considerably more types than could be inferred from the source code. For example, Figure 3.2 defines a SML module with methods `main` and `SelfTest` at the module level and methods `Fibonacci` and `FibSeries` in the class

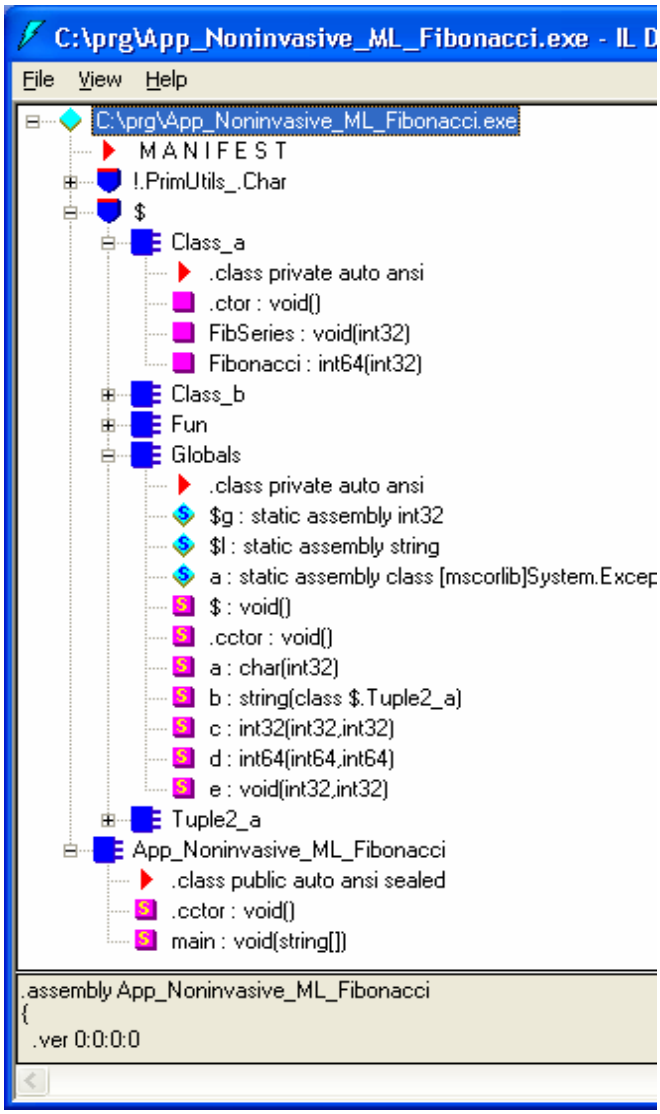

Figure 3.3: Disassembler view of types contained in a component written in SML.NET source in Figure 3.2.

`FibonacciSeries`. Using the directive "`export App_Noninvasive_ML_Fibonacci`" to compile this source results in a CLI component containing a surprising number of additional types. As shown in Figure 3.3, a disassembler view of

the type definitions in the component uncovers a large number of types for which there are no explicit declarations in the source code. As expected, there is a type corresponding to the module that contains the implementation of `main` and `SelfTest`, and there is a class corresponding to the `FibonacciSeries` class declaration that contains the implementations of `Fibonacci` and `FibSeries`. The difficulty is that there are other types such as `Globals` with methods such as "`static char a(int32 A_0)`" that would match the property-based crosscut for logging shown in Figure 3.1.

Our tests verified that property-based crosscutting has useful reusability characteristics, but its drawbacks make it quite difficult to use when obtaining strong assurances of correct and predictable behaviour is a concern. Difficulties in specifying XML using a language independent type system made it difficult to write crosscutting specifications by hand. While these could be overcome with diligence, the inadvertent join point selection could not. New join points introduced by the compiler when a component was compiled could not be determined through source code analysis, whether done by a human or via an automated application.

4. LOGGING VIA ATTRIBUTE-BASED CROSSCUTTING

To contrast attribute-based crosscutting, we revised the logging aspect to exploit attribute types for join point selection, and composed this new logging aspect with the Fibonacci series algorithm implementations. Our evaluation noted the use of attribute-based crosscutting offers a more succinct and accurate means of applying crosscutting functionality, and attributed-based crosscutting avoided the unexpected join point selection that prevented application behaviour from being predicted from source code analysis.

Attribute-based crosscutting specifications complement an attribute type [8] that allows access to aspect functionality through annotation of component source. In contrast to property-based crosscutting, attribute-based crosscutting uses attribute type names in place of join point implementation details such as types and type member signatures. When using attribute type names, the grammar for pointcut specifications is unchanged when it comes to the primitive pointcut designators available, but the parameters used for designators are changed. Rather than signature or type name arguments, primitive pointcut designators are parameterized with attribute tags describing the attribute type name. In the case of the Weave.NET, these attributes are implemented by the CLI's custom attribute types.

The contrast between property-based and attribute-based crosscutting can be seen in Figure 4.1. The top pane of the figure contains the execution pointcut specification used in Figure 3.1 to select execution join points for logging. In this pane, the selection of method execution join points is based on a partial method signature. In the bottom pane, the specification is revised to select methods tagged with an attribute type with the name `Logging`. This second version contains considerably fewer terms than the first, but it is reliant on the ability to annotating method source with an attribute type.

An example application of attribute types is shown in Figure 4.2, where methods of the Fibonacci series algorithm are bound to logging functionality. This example emphasizes the attribute annotations by marking them in bold.

```
<execution>
  <method_signature>
    <return_type>
      <type_name>*</type_name>
    </return_type>
    <join_point_type>
      <type_name>*</type_name>
    </join_point_type>
    <method_name>*</method_name>
    <parameters>
      <parameter>
        <type_name>Int32</type_name>
      </parameter>
    </parameters>
  </method_signature>
</execution>
```

```
<execution>
  <attribute>Logging</attribute>
</execution>
```

Figure 4.1: Contrast between property-based crosscutting (top) and an aspect-based crosscutting (bottom).

```
public class FibonacciSeries {
 [Logging]
  public void FibSeries(int seriesLen) {
    for (int i = 0; i<= seriesLen; i++) {
      long result = Fibonacci(i);
      System.Console.WriteLine("Element \t"+
                        i+ "\tvalue \t"+result);
    }
  }
 [Logging]
  public long Fibonacci(int n) {
    if (n > 1)
      return this.Fibonacci(n-1)
           + this.Fibonacci(n-2);

    return 1;
  }
}
```

Figure 4.2: Fibonacci series enumerator annotated with attributes to identify methods for logging.

In our tests, we noted attribute-based crosscutting provides an alternative means of identifying CLI metadata that avoids mistakes made with property-based crosscutting specifications that are extremely difficult to detect. Recall that writing property-based crosscutting involves specifying join points in terms of metadata that is native to the CLI. There is little help available from the weaver for detecting erroneous type specifications, as it is hard to design a weaver that can distinguish between types that are specified correctly and those that are specified in error. For instance, the method parameters in Figure 4.2 are of type `int`. `int` is the C# moniker for the CLI type `System.Int32`, and thus the short form `Int32` appears in the property-based crosscut in Figure 4.1. Should the type `int` appear accidentally in the crosscutting specification, one would expect the weaver to complain. However, it is legitimate for a programmer to define a custom CLI type by the name of `int` in a different namespace. Even if we require that type names in the crosscutting specifications include a full namespace, `int` is still a valid user defined type. Attribute-based property selection avoids the issue of detecting errors made when the language type name is mapped to the CLI type name mappings, since the placement of attributes on types or type members avoids the need to deal with join point

selection in terms of CLI-specific type names. In effect, the use of attributes represents the introduction of language-independent monikers for types and type members.

```
structure App_Invasive_ML_Fibonacci
  : sig val main: string option array option -> unit
    end =
struct
_classtype FibonacciSeries()
 with
  {Aspect_CS_Logging.Logging()} Fibonacci (n) =
    case(n) of
      0 => (Int64.fromInt(1))
    | 1 => (Int64.fromInt(1))
    | n => (this.#Fibonacci(n-1) + this.#Fibonacci(n-2))
 and
  {Aspect_CS_Logging.Logging()} FibSeries (n) =
    case(n) of
     ~1 => ()
    | n => (this.#FibSeries (n-1);
            print "Element\t"; print (Int.toString (n));
            print "\t value \t";
            print (Int64.toString(this.#Fibonacci (n)));
            print "\n" )
 end
      ...
end
```

Figure 4.3: SML.NET implementation of Figure 3.2 updated to exploit custom attributes.

Also, attribute-based property selection avoided unexpected join point selection, since attributes follow the implementation of the tagged method. With revisions to include attributes, the SML-based Fibonacci series algorithm in Figure 3.2 takes on the appearance of that of Figure 4.3, where attributes appear in bold. Note that the definitions of `main` and `SelfTest` have been removed for brevity. As before, additional helper types will appear in the compiled assembly. However, an examination of the metadata of the assembly indicates that only those methods explicitly tagged at the source code level will have their metadata description annotated by the logging attribute in the compiled assembly. Thus, applying logging on the basis of attributes rather than method signature, constrains logging to the `Fibonacci` and `FibSeries` methods.

So, in addition to making it simpler to specify aspect-based software product families, attribute-based property eliminated problems with predictability that had prevented strong assurances of correct application behaviour from being made from an examination of component source.

5. CONCLUSIONS

In the context of language-independent AOP, attribute-based crosscutting specifications have two important advantages over property-based crosscutting. Use of attributes represents the introduction of language-independent monikers for types and type members that simplify the specification of crosscutting in a language independent fashion. Second, attribute-based crosscutting will not inadvertently match unwanted join points introduced during the translation of a component source to a language-independent substrate. This occurs because only those component structures explicitly annotated with an attribute at the source code level will have their metadata description annotated by that logging in the compiled component.

6. REFERENCES

1. Bloch, J. JSR-000175 A Metadata Facility for the JavaTM Programming Language, http://jcp.org/aboutJava/communityprocess/review/jsr175/, 2003.
2. Costanza, P. Dynamically scoped functions as the essence of AOP. *ACM SIGPLAN Notices, 38* (8). 29 - 36.
3. ECMA International. Standard ECMA-335 Common Language Infrastructure (CLI), http://www.ecma-international.org/publications/standards/ecma-335.htm, Geneva, 2003.
4. Filman, R.E. and Friedman, D.P. Aspect-Oriented Programming is Quantification and Obliviousness *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2000)*, Minneapolis, USA, 2000.
5. Kennedy, A., Russo, C. and Benton, N. SML.NET 1.1 User Guide, http://wwww.cl.cam.ac.uk/Research/TSG/SMLNET/smlnet.pdf, Cambridge, U.K., 2003.
6. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G. Getting Started with AspectJ *Communications of the ACM*, 2001, 59-65.
7. Lafferty, D. Aspect-Based Properties *Dept of Computer Science*, Trinity College Dublin, Dublin, 2004.
8. Lafferty, D. and Cahill, P.V., Attribute Types. in *Submitted for review to ECOOP 2005*, (Glasgow, Scotland, 2005), Springer.
9. Lafferty, D. and Cahill, V., Language-Independent Aspect-Oriented Programming. in *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2003)*, (Anaheim, California, USA, 2003).
10. Masuhara, H. and Kiczales, G., Modeling Crosscutting in Aspect-Oriented Mechanisms. in *European Conference on Object-Oriented Programming (ECOOP 2003)*, (Darmstadt, Germany, 2003), Springer-Verlag.
11. Microsoft. Standard ECMA-334 C# Language Specification, ECMA International - European association for standardizing information and communication systems, 2001.
12. Microsoft. Visual Basic Development Center, http://msdn.microsoft.com/vbasic, 2004.
13. Popovici, A., Alonso, G. and Gross, T. Spontaneous Container Services *European Conference on Object-Oriented Programming (ECOOP 2003)*, Springer, Darmstadt, Germany, 2003.
14. Szyperski, C., Gruntz, D. and Murer, S. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, London, 2002.
15. The AspectJ Team. The AspectJ Programming Guide (V1.0.6), http://download.eclipse.org/technology/ajdt/aspectj-docs-1.0.6.tgz, 2002.