Declaring and Enforcing Dependencies Between .NET Custom Attributes

Vasian Cepa and Mira Mezini

Software Technology Group, Department of Computer Science
Darmstadt University of Technology, Germany
{cepa,mezini}@informatik.tu-darmstadt.de

Abstract. Custom attributes as e.g., supported by the .NET framework complemented by pre- or post-processing tools can be used to integrate domain-specific concepts into general-purpose language technology, representing an interesting alternative to domain-specific languages in supporting model-driven development. For this purpose, it is important that dependency relationships between custom attributes, e.g., stating that a certain attribute requires or excludes another attribute, can be specified and checked for during model processing (compilation). Such dependencies can be viewed as an important part of expressing the meta-model of the domain-specific concepts represented by custom attributes.
In this paper, we present an approach to specifying and enforcing dependencies between .NET custom attributes, which naturally extends the built-in .NET support. In this approach, dependencies are specified declaratively by using custom attributes to decorate other custom attributes. Once the dependency declaration is made part of the custom attribute support, one can write tools that enforce dependencies based on .NET meta-program API-s like CodeDom or Reflection. In this paper, we present such a tool, called ADC (for attribute dependency checker).

1 Introduction

.NET [24] has native support for introducing custom attributes [13] which can be used to decorate program elements. Elsewhere [26], we have argued that custom attributes combined with API-s like `Reflection` and `CodeDom`[1] provide built-in support for integrating domain specific abstractions without the burden of rewriting the parser and/or the compiler of the language.

For illustration, assume that we want to extend an object-oriented language with constructs that facilitate building web services. In the simple example of Fig. 1 we have introduced two new keywords `webservice` and `webmethod` for this purpose which are used to define the `TravelAgent` as a web service component. While nice to have, this extension requires that the programmer understands and modifies the existing grammar rules to add support for the new keywords to the parser.

[1] All non cited references about .NET come from MSDN [22] documentation.

G. Karsai and E. Visser (Eds.): GPCE 2004, LNCS 3286, pp. 283–297, 2004.

```
webservice TravelAgent {
  ...
  webmethod GetHotels(){...}
  ...
}
```

Fig. 1. A domain-specific extension to implement web services

With attributes being built-in elements of the general purpose .NET programming framework, new (domain specific) abstractions can be expressed by annotating existing language elements. Meta-programming APIs can than be used to build transformers that identify decorated elements and transform the AST to integrate the semantics of the attributes. Hence, a language framework with support for attributes is capable of expressing executable (domain-specific) models, representing an interesting alternative to implementing domain-specific languages by means of traditional compiler development tools.

In a .NET language that supports attributes like C# we would write the same web service extensions of Fig. 1 by introducing two custom attributes as shown in Fig. 2. The `TravelAgent` class itself is decorated with the attribute `[WebService]`, whereas its public methods that constitute the web service interface should have a `[WebMethod]` attribute. Introducing new attributes is supported by .NET compilers, thus, we do not need to deal with grammar modification issues which makes it easier to extend a .NET language like C# with domain-specific constructs.

```
[WebService]
class TravelAgent {
  ...
  [WebMethod]
  public void GetHotels(){...}
  ...
}
```

Fig. 2. A web service class with two inter-depended attributes

.NET follows a hybrid approach with respect to attributes: It distinguishes between *predefined* and *custom* attributes. Predefined attributes are used by various API-s of the .NET platform. For example, `[System.Diagnostics.ConditionalAttribute]` is used by the preprocessor to condition the inclusion of methods in the compiled version. The compiler relies on *attribute provider* libraries to interpret the predefined attributes. In contrast to predefined attributes, custom attributes do not, in general, have a meaning to the compiler. Code to interpret a custom attribute has to be implemented by the developer that uses the attribute to introduce domain-specific concepts. Given that in .NET an attribute is defined in a class, the interpretation code can be placed inside the attribute class itself; however, when we need a bigger context to properly interpret the attribute, we place the interpretation code in a separate module.

We observe that there is some custom attribute interpretation code which is so common place and general that we may need to repeat it over and over. An example which we are concerned with in this paper is code needed to enforce dependencies between attributes, requiring e.g., that a certain attribute is present in the program hierarchy before another attribute can be used. A grammar rule such as `webservice := webmethod+` explicitly defines a context relation between `webservice` and `webmethod`. That is, a web method will appear only inside a web service and vice-versa, a web service will contain web methods. Depending on whether we consider the class or its methods, there are two constrains we want to enforce: (a) public methods of a class decorated as `[WebService]` should be decorated with the `[WebMethod]` attribute, (b) any method decorated with a `[WebMethod]` attribute should be declared within a class decorated with the `[WebService]` attribute. That is, the two attributes are inter-dependent. In general, however, the dependency relation need not be symmetric.

Specifying dependency relationships between custom attributes and checking for them during model processing (compilation) is important, in order to make effective use of custom attributes for supporting modeling with domain-specific abstractions directly at the code level. Such dependencies can be viewed as an important part of expressing the meta-model of the domain-specific concepts represented by custom attributes. Furthermore, specifying such dependencies is important to support feature-oriented modeling [15]. If we assume that feature-specific models are expressed in terms of custom attributes corresponding to domain-specific concepts, then expressing feature dependency relationships, requires a means to express and enforce respective dependencies between attributes.

How can attributed dependencies specified and checked? One can leave it to each specific transformer tool which will process the tag-decorated code to check dependencies between attributes. However, with this alternative the dependencies are not explicitly specified and we have to repeat the same checking logic in every transformer. A better alternative would be to declaratively specify dependencies as we do with grammar rules and process such specifications in a generic way before/after the compilation of the decorated code, but before the attributes are processed. Extending the .NET support for attributes to enable declaring and checking dependency constraints between custom attributes based on the second alternative is the main contribution of this paper. We do so in a way that is natural for .NET attribute programmers, without introducing any external notation apart from what is already present in .NET.

Instead of requiring that developers of transformation tools repeat dependency checks over and over, we propose to extend .NET with a new custom attribute, that allows to express such relations declaratively by decorating the involved attributes. This is similar to the use of the predefined `[System.AttributeUsageAttribute]` used in .NET to decorate a custom attribute, providing information about the lexical scope in which the attribute at hand can be used. Based on these usage attributes, any time a custom attribute is encountered in

a program, the compiler can check, if it is being used in the right lexical context and report an error if this is not the case.

We adopt the idea underlying `[System.AttributeUsageAttribute]` to introduce new custom checks, now using custom attributes. That is, given that in .NET attributes are themselves program elements, we recursively use the mechanism of decorating program elements with attributes to extend the .NET support for attributes with dependency declarations. Using custom attributes to decorate other custom attributes is a natural way to extend .NET's attribute support. Given that we cannot change the compiler, we need a way to interpret such dependency attributes. This can be done with a *pre-processor* tool which applies the checks before the compilation using `CodeDom`[2], or with a *post-processor* tool which applies the checks after the compilation, given that .NET saves the attribute information as part of the IL (Intermediate Language) meta-data.

The technique frees the programmer that writes attribute interpreters from repeating code by centralizing checks to be part of the compilation process; the programmer only declares the dependencies without taking care of how they are resolved and enforced. There are of course many ways to declare and enforce such architectural dependencies [21]. However, using attributes to decorate other attributes is a very natural way for .NET.

The remainder of the paper is organized as follows. Sec. 2 presents our dependency constraint model, and shows by means of examples how our model can be used to specify dependency relationships between domain specific concepts expressed via attributes. Sec. 3 presents ADC - an attribute dependency checker tool for enforcing dependencies based on our dependency model. Sec. 4 presents related work. We summarize the paper in Sec. 5.

2 The Attribute Dependency Model

We distinguish between (a) *required* dependencies - stating that a given attribute requires another one in order to be used, and (b) *disallowed* dependencies - stating that a given attribute cannot be used, if another attribute is present. Furthermore, children nodes in a program's structural hierarchy can declare dependency constrains for parents of any level, and vice-versa. An attribute of a certain program element instance may require that certain attributes are present in the set of the attributes of the structural children of the program element at hand. For example, a *Class* attribute may require a certain attribute to be present in the class' *Methods*. The reverse is also true: An attribute of a child structural element instance may require a certain attribute to be present in the set of the attributes specified for the parent instance. In our model, we generalize these notions to any depth of the structural tree.

On the contrary, sibling nodes in a program's structural hierarchy are not allowed to put constraints on their respective attributes. The attributes of a program element instance cannot place any constrain on the attributes of sibling instances. For example, the attributes that a *Field* instance is decorated with

[2] When `ICodeParser` is implemented.

cannot imply anything about the attributes of *Method* instances or attributes of other *Field* instances. In the same structural level we cannot say anything about the siblings that will be there. However, the attributes of a program element instance can place constrains on other attributes of the same instance. For example, a method attribute a_{m1} of a method m may require another attribute a_{m2} to be present for m.

The semantics of the *disallowed* relation on the structural tree elements and instances can be specified similarly and will not be repeated here.

2.1 The `[DependencyAttribute]` Class

.NET custom attributes are classes derived from the class `System.Attribute`. They may have arguments specified either as constructor parameters - unnamed arguments -, or as properties of the attribute class which generate *getter* and *setter* methods in C# - named arguments. Attribute classes may also contain methods and state like any other class. Using properties to specify attribute arguments is more flexible than using constructors. The reason is that .NET does not support complex types to be passed as parameters to the constructors[3]. Hence, we define an attribute class `DependencyAttribute`[4] which has one `Required*` and one `Disallowed*` property for each program element type for which attributes are supported, as shown in Fig. 3. Given that the number of the node types in a program's structural tree (`Assembly`, `Class`, `Method`, etc.) is limited, it makes sense to enumerate such operations. This makes the code easier to understand compared to having a single dependency property for all meta-element types.

```
[AttributeUsage(AttributeTargets.Class)]
public class DependencyAttribute : System.Attribute {
  ...
  public DependencyAttribute() {...}
  public Type[] RequiredAssemblyAttributes {...}
  public Type[] DisallowedAssemblyAttributes {...}
  public Type[] RequiredClassAttributes {...}
  public Type[] DisallowedClassAttributes   {...}
  public Type[] RequiredMethodAttributes {...}
  public Type[] DisallowedMethodAttributes {...}
}
```

Fig. 3. The implementation of dependency attribute

[3] Only basic constant types and `System.Type` can be used. `System.Object` is also listed in the documentation because it is the parent of simple types and of `System.Type`. However, this does not mean that arbitrary objects can be passed as constructor parameters.

[4] When used in code the suffix `Attribute` may be omitted from the name of an attribute class.

However, the current .NET implementation seems to restrict the complexity of the validation logic that one can implement inside a property of a custom attribute. It is unclear in the .NET documentation whether that code is ever activated. Furthermore, we found that if the code inside a property is more than a simple assignment, that property may be not included in the attribute class without warnings from the compiler. Thus, we must keep the code of the `DependencyAttribute` properties simple and postpone checks, e.g., that the attributes passed as a parameter to a property have the right `[System.AttributeUsageAttribute]` target[5], until the dependency checking is performed.

The `DependencyAttribute` only stores the required/disallowed attribute arrays and implements only code for printing these arrays as strings needed for error and log reporting. It does not implement any code to interpret the dependencies and its implementation has no other module dependencies. As a consequence, `DependencyAttribute` is independent of any particular dependency checker implementation and can be distributed and used alone to decorate attribute libraries.

The current implementation of our dependency model only supports `Assemblies`, `Classes` and `Methods`. Adding support for `Fields` is trivial (see the implementation details in the next section). Readers familiar with .NET may note that we have skipped `namespace`-s[6] in the list above. The reason is that a `namespace` is a logical rather than a physical concept, so even though we can theoretically decorate a `namespace` with attributes, practically there is not a single physical place where to store the attributes, since a `namespace` may be expanded in many modules and assemblies[7].

2.2 Using the Dependency Attribute

Fig. 4 shows how the `DependencyAttribute` can be used in code to enforce the dependency semantics of the attributes `[WebService]` and `[WebMethod]` from the example in Fig. 2. Note the use of the C# `'typeof'` operator to obtain an instance of the class type of each attribute.

The following example shows how other checks involving declarative attributes can be expressed using the dependency attribute. It is motivated by implementation restrictions of the EJB [7] container programing model. The EJB specification states, among other restrictions, that components whose instances will be managed as *virtual instances* [20] should not pass *this* as a parameter or return value; the underlying idea is that it makes no sense to return a direct pointer to an object that will be reused with other internal state later by the container.

[5] E.g., `AttributeTargets.Method` should be present in the declaration of an attribute included in the list `RequiredMethodAttributes`.

[6] For readers with a Java background `namespace` maps roughly to a `package`; an `Assembly` maps roughly to a JAR file; the `Assembly` attributes map roughly to custom JAR manifest entries.

[7] This is the reason why .NET does not list `namespace` as an entry in the `AttributeTargets` enumeration.

```
[Depedency(RequiredMethodAttributes(new Type[]{typeof(WebMethod)})]
[AttributeUsage(AttributeTargets.Class)]
class WebService : System.Attribute { ... }

[Depedency(RequiredClassAttributes(new Type[]{typeof(WebService)})]
[AttributeUsage(AttributeTargets.Method)]
class WebMethod : System.Attribute { ... }
```

Fig. 4. Using the dependency attribute

While the EJB implementations currently do not rely on tags, we can imagine that tags are used in the same way as EJB marking interfaces [14][8]. Let us suppose that the lifetime of an instance of a class is going to be managed by the container only when the class declaration is decorated with the tag `[VirtualInstance]` (Fig. 5). For a class `C` tagged with the attribute `[VirtualInstance]`, the restriction about *this* must hold. We express this requirement explicitly by means of a new tag `[NoThis]`. Let us suppose that the method `initialize()` of class `C` is invoked by the container when a virtual instance need to be initialized. We place no restrictions on this method's signature, but annotate it with a `[InitInstance]` attribute to distinguish it for later processing. In this context, we use `[NoThis]` as an attribute for decorating program elements and use the dependency attribute to state that it is required whenever `[VirtualInstance]` and/or `[InitInstance]` are used, as in Fig. 6.

```
[VirtualInstance]
class C {
  ...
  [InitInstance]
  public C initialize(Id id){...}
  ...
}
```

Fig. 5. A class that requires virtual instance support

That is, we require `[NoThis]` to be used explicitly in order to check this restriction. Optimally, `[NoThis]` should be used as a meta-attribute to decorate the attributes `[VirtualInstance]` and `[InitInstance]`, i.e., at the same abstraction level as `[DependencyAttribute]`, letting an extra tool to check for it. However, this is out of the scope of the discussion here, since all we are interested in is to use the EJB restriction "not allowed to pass this" only as a means to illustrate the dependency attribute. The less declarative notation by using the `[DependencyAttribute]` does not remove the need that the corresponding container transformer must later enforce `[NoThis]` semantics in the appropriate way. It only offers a first and quick automated test of the correct usage, which saves us from defining a tool for only enforcing declaratively the

[8] Actually, Java 1.5 annotations will be used in EJB 3.0 instead of marking interfaces.

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
class NoThis : System.Attribute { ... }

[Depedency(RequiredMethodAttributes(new Type[]{typeof(NoThis)})]
[AttributeUsage(AttributeTargets.Class)]
class VirtualInstance : System.Attribute { ... }

[Depedency(RequiredClassAttributes(new Type[]{typeof(VirtualInstance)}),
   RequiredMethodAttributes(new Type[]{typeof(NoThis)})]
[AttributeUsage(AttributeTargets.Method)]
class InitInstance : System.Attribute { ... }
```

Fig. 6. Using dependency attribute to check `[NoThis]` constraints

`[NoThis]` attribute semantics. Of course, the price of this convenience is that the programmer must then use `[NoThis]` explicitly in code.

The example illustrates also the non-symmetry of the dependency relation as the `[VirtualInstance]` class attribute requires `[NoThis]` method attribute to be present, but `[NoThis]` method attribute can be used also in methods inside classes that do not have a `[VirtualInstance]` attribute.

3 The Attribute Dependency Checker (ADC) Tool

The Attribute Dependency Checker (ADC) tool, which can be downloaded from [27], is implemented as a *post-processor* using the `Reflection API`. After the code is compiled and linked one can run the ADC tool over the IL binaries to detect dependency errors, if any. Alternatively, ADC could be implemented as a *pre-processor* tool to be run before the source is compiled using the `CodeDom API`[9].

Fig. 7 shows the classes of ADC and their relations. Almost all the logic of the dependency checker is found in the abstract class `AttributeDependency-Checker`. It uses several helper classes and interfaces (a) to filter the processed elements (`IDependencyFilter`), (b) to log information about the progress of the checking process (`ICheckLogger`), and (c) to report errors (`ErrorReport`). The `IContextMap` class encapsulates the meta-model structure in a single place using a special internal coding. For illustration, Fig. 8 shows how the ADC library can be used to check the attribute dependency constrains for all elements of a given .NET assembly.

In order to implement the semantics of the dependency attribute we need to first build the dependency sets for each structural element by processing the element and all its structural children. After the dependency sets are constructed, we can check the dependency constrains of the element. That is, we need a post-order transversal of the structural tree. A boolean flag in `AttributeDependencyChecker` controls whether the inherited attributes of the structural elements are processed. The actions performed during a call to the `Check(t)`

[9] A third party implementation of `ICodeParser` for C# is presented in [12].

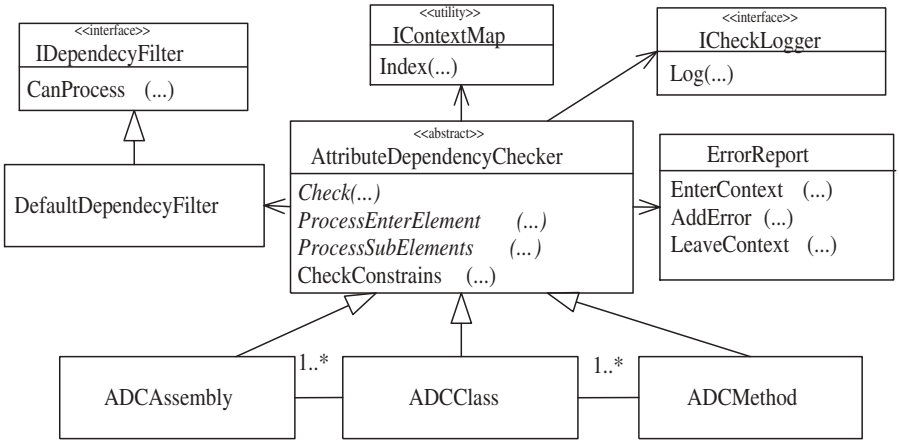

Fig. 7. The run-time attribute dependency checker structure

```
Assembly a = ...; // obtain an assembly
ADCAssembly c = new ADCAssembly();
c.Filter = ...;
c.Logger = ...;
c.Check(a);
if(c.errors.HasWarnings())
{ // process: c.errors.GetWarnings() ... }
if(c.errors.HasErrors())
{ // process: c.errors.GetErrors()   ... }
```

Fig. 8. Using the run-time attribute dependency checker in code

method, where `t` is the current program element whose attribute dependencies are being checked for, are illustrated in Fig. 9.

First, the filter object is used to check whether the element at hand should be processed (step (2) in Fig. 9). Filters can be used to put arbitrary constrains on the elements that will be processed, e.g., using pattern matching on names. The `DefaultDependencyFilter` processes all the elements. The ADC tool uses a customized filter called `ClassDependecyFilter` derived from `DefaultDependencyFilter` that can restrict checking to a subset of classes whose names are given in the command line. More sophisticated filters can be written and used in a programmatic way. Filters can also be used to implement profiling by keeping track of various counters; e.g., `ClassDependecyFilter` counts the number of classes and methods processed.

Next, the call to `ProcessEnterElement()` (step (3) in Fig. 9) sets the proper `ErrorReport` context (explained later) (step (4) in Fig. 9) to be used when processing the sub-elements of the element at hand. The `ProcessSubElements()` method (step (6) in Fig. 9) calls the `Check()` method of all sub-elements. As shown in Fig. 7, the specific attribute checkers for different meta-elements, e.g.,

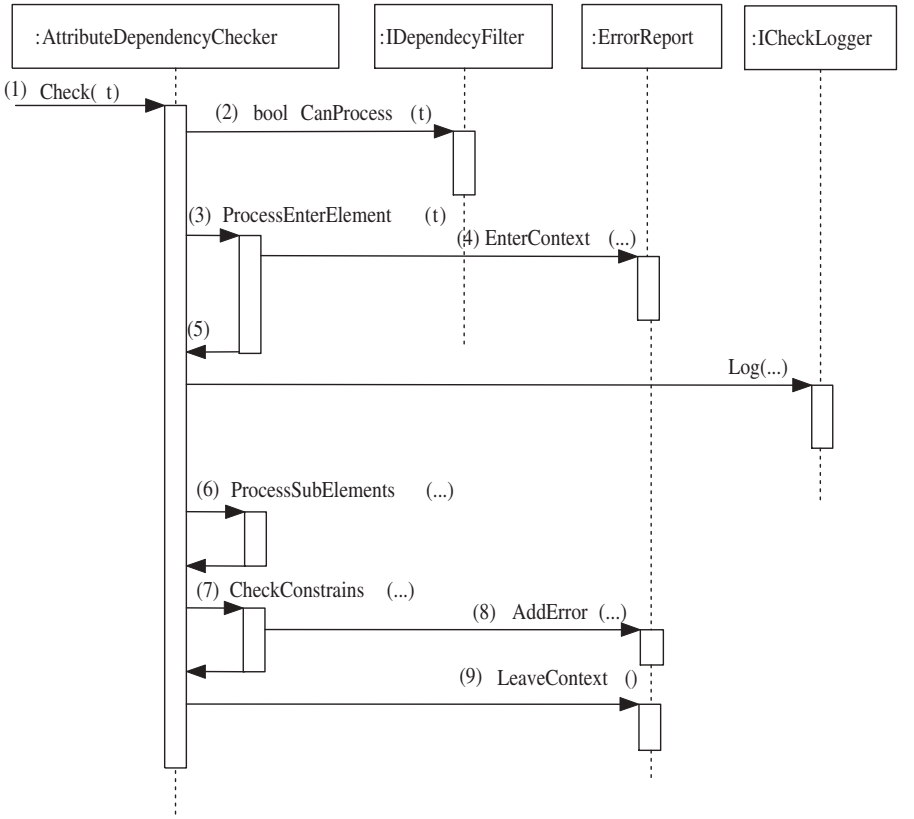

Fig. 9. UML sequential diagram of `Check()` method call

`ADCAssembly`, `ADCClass`, etc., are derived form `AttributeDependencyChecker`, by implementing the abstract methods: `Check(object t)`, `ProcessEnterElement(object t)`, and `ProcessSubElements(ref ArrayList ctx, object t)`. For illustration, Fig. 10 shows the implementation of the `ProcessSubElements` method in `ADCClass`. As we see, only the browsing logic of finding the sub elements is part of this method.

The context to be used during the processing of a node and its descendants (the parameter `ctx` in the signature of `ProcessSubElements`) is managed in an `ArrayList` similarly to the method call stack frames in a compiler [2]. A frame in a context contains the attributes and the dependency attributes of a particular element. When we browse the structural tree (by calling `ProcessSubElements`), we fill up the context passing it to every processed sub-element. Each element, when it is processed, can modify the dependency information of context frames of its parents. When we leave a sub-element its frame is removed from the context. After all sub-elements of a given element are processed we have all the required information in the context stack frames to check the dependencies of

```
protected override void ProcessSubElements(ref ArrayList ctx, object t) {
  MethodInfo[] m = ((Type)t).GetMethods(
    BindingFlags.Instance |
    BindingFlags.Public |
    BindingFlags.DeclaredOnly |
    BindingFlags.NonPublic);

  foreach(MethodInfo mi in m) {
    ADCMethod adc = new ADCMethod();
    CopyStateTo(adc);
    adc.InitialContext = ctx;
    adc.Check(mi);
  }
}
```

Fig. 10. `ADCClass` implementation of `ProcessSubElements` method

the given element. We compare then the actually present attributes with the total dependency attributes for the current frame, using set operations, in the `CheckConstrains()` method (step (7) in in Fig. 9).

The `:ErrorReport` object maintains its own context (set up in step (4) in in Fig. 9) so that when an error is reported (step (8) in in Fig. 9) it can be embedded within the proper structural context. The `ErrorReport` context is used to report messages in a useful way, as illustrated by the error message:

```
| Required CLASS attribute missing:
| adctests.CA01Attribute @ adctests->adctests.nunit.TDependencyUtils
```

This error message specifies that the required class attribute `adctests.CA01-Attribute` is missing in class `adctests.nunit.TDependencyUtils`, part of `adctests` assembly.

By default `ErrorReport` accumulates the errors, but this behavior can be changed via a switch, so it will break the checker execution if an error happens by throwing a `ADCException`. `ErrorReport` contains also logic to accumulate or immediately report the improper usages in code of the parameters passed to the `DependencyAttribute` itself. An example is passing an attribute declared with a class lexical scope as an argument to a `RequiredMethodAttribute` property.

Finally, the `ICheckLogger` interface (see step (5) in Fig. 9) allows the programmer to associate a customized logger with the checker. If the logger is not `null`, a hierarchy of the processed elements with details about their attributes and attribute dependencies is printed. A filter could also be used for custom logging. All objects shown in Fig. 9, but `:AttributeDependecyChecker`, are singletons and are passed to the processing of the sub-elements as part of the context.

The implementation of the class `AttributeDependencyChecker` is generic w.r.t the implementation of both `DependencyAttribute` and the meta-model elements, which means that we can reuse its implementation with new attributes

as well as with other meta-models. The `AttributeDependencyChecker` achieves this generality by using a combination of the following three techniques:

- First, all the hierarchy information of the supported meta-model is factored out into two static methods (tables) of the `IContextMap` utility class. `AttributeDependencyChecker` uses `IContextMap` to implement a strategy pattern [6]. By changing the `IContextMap` class, users can change the supported meta-model. Theoretically, the information in `IContextMap` would be enough to check the dependencies, i.e., no specific checker classes for different elements of the meta-model, e.g., `ADCClass` would be needed. However, the .NET Reflection API design is not consistent in browsing the meta-elements hierarchy. Unlike other API-s, e.g., XML DOM [4], that have a single base interface, `Node`, from which all elements are derived, the .NET Reflection API does not expose a single generic interface for meta-element types. The rationale being that the number of meta-elements is limited. However, this requires that when adding new meta-elements to the ADC library, we need to derive special classes for them which contain only sub-element browsing code, as described above.
- Second, given the structure of the meta-model is present in the `DependencyAttribute` properties, we use reflection inside the `Check(...)` method over any `DependencyAttribute` properties and map them to the internal `IContextMap` context. The use of the reflection ensures that if we add or remove attributes to the `DependencyAttribute` class, the implementation of the `AttributeDependencyChecker` does not need to be changed. Another generic alternative would be to generate this code based on the `DependencyAttribute` implementation, but this would require to re-generate and re-compile the `AttributeDependencyChecker` for every different version of the `DependencyAttribute` implementation.
- Third, we use the template method pattern [6] to call abstract methods that need to be implemented in the derived classes, like the `ProcessSubElements` method required to browse the sub-elements. The entire checking logic is part of the abstract class `AttributeDependencyChecker`.

The resulting ADC library can be easily extended to support new meta-elements. If we need to add a new type of checker for attributes of another meta-element, we need to derive a class from `AttributeDependencyChecker`, implementing the abstract methods discussed above. In addition, the `IContextMap` class needs to be modified to accommodate the hierarchical structural relation of the new element with the existing elements.

4 Related Work

Using attributes to denote additional custom semantics about an entity is intuitive and is used all around in computer science [16]. Different names used for attributes, range from *tags* to *annotations*. Explicit annotation [1] of source code elements with attributes falls between domain specific languages [3] and

generative programming techniques. Explicit attributes can extend the model supported by a generic language, without changing its front-end compiler tools [28] and can be used to drive code transformations [25]. In OMG MDA [5] *tags* are used to *mark* model elements. In the MDA MOF [29] and UML [8] standards, tags have no semantics to the standards themselves. They are used during model transformations as hints by the transformation tools.

Hedin [9] describes how attribute extension grammars can be used to enforce properties about library components that can not be enforced otherwise with object-oriented systems. The work is superseded by language technologies like .NET that directly support attributes and offer API-s to access the AST information along with the decorated attributes. Our approach is situated at a higher abstraction level, using attributes to define declarative rules that must hold between attributes.

Declaring and checking attribute dependencies is one example of explicitly enforcing architectural principles [21]. In fact, attributes offer a unified way to express evolution invariants in languages that support explicit annotations, given that any structural entity can be decorated independently of the syntax. This makes attributes attractive for expressing law-governed system evolution rules. We can express architectural principles that must hold between program entities, as attribute dependencies between architectural attributes used to decorate program entities. System wide invariants can be expressed as `Assembly` attributes and rules can be expressed by meta-attributes over architectural attributes.

Abadi *et al* [17] state that there is a central notion of dependency and abstract any kind of dependency into a Dependency Code Calculus (DCC) based on a computational lambda calculus. Such a formal abstraction can be interesting for proving properties of dependent system elements, but it must be specialized to a specific domain to be of real usage, yielding in different special purpose calculuses. However, some of the dependency problems mentioned in [17] like slicing calculus do not map directly into source code program dependencies and cannot be expressed as source code attributes.

Aspect-oriented programing (AOP) [11] techniques can be also used to enforce architectural decisions. Its usefulness in program generation [15] is based on its global view of the system, which is required to enforce system wide properties. An example how AspectJ [10] (an AOP tool for Java) can be used to enforce system constrains is given in [18]. However as noted there, there are some system wide constrains like name capitalization which cannot be enforced directly with AspectJ. This is because AspectJ abstracts program meta-information between: (a) pointcut declarations - that encapsulate meta-element selection and context; and (b) advice implementations - that make implicit meta-element manipulation. Since the enumeration of all possible meta-operations as declarative constructs is impossible and was not a design goal of AspectJ, there are meta-level programs that cannot be expressed as AspectJ programs.

Our declarative attribute approach is a new natural generative pattern [19] to enforce domain-specific [3] meta-models over attributes. Other problems apart of attribute dependency can be also generalized at the attribute level. However we

must note that currently .NET supports only structural elements to be decorated with attributes. We cannot place attributes inside methods, liming rules that can be enforced by our approach.

There are also many generic tools like [30, 31] designed for enforcing rules about a program not covered directly by the programming language mechanisms used. We demonstrated how declarative dependencies can be expressed as attributes to decorate custom attributes, providing a natural way to extend language technologies like .NET where attributes are full status entities. Our approach is however not suited for checking arbitrary program restrictions, which may require customized imperative implementations.

5 Summary

The .NET compiler support for checking custom attributes is limited. We took advantage of the fact that .NET attributes are full status types in the .NET framework, and extended the .NET compiler attribute support with custom declarative checks using attributes themselves. We showed how to decorate custom attribute declarations with other attributes that define additional declarative semantics about the custom attributes. This is a natural way to extend .NET attribute support using pre-processor or post-processor tools, being thus a convenient alternative for supporting domain-specific language constructs [3] and various program transformation techniques [23].

We showed how the attribute dependency problem can be formalized and expressed declaratively as an custom attribute. We described the implementation of the ADC [27] tool designed to check such dependencies. ADC is implemented in a very generic and extensible way. Other attribute enforcement checks can be expressed declaratively in the same way for .NET-like language technologies.

References

1. K. De Volder G. C. Murphy A. Bryant, A. Catton. Explicit Programming. *In Proc. of AOSD '02, ACM Press*, pages 10–18, 2002.
2. A. V. Aho, R. Sethi, J. D. Ullman. *Compilers Principles, Techniques and Tools.* Addisson Wesley, 1988.
3. A. van Deursen, P. Klint, J. Visser. Domain-Specific Languages. *ACM SIGPLAN Notices, Volume 35*, pages 26–36, 2000.
4. B. McLaughlin. *Java and XML.* O'Reilly, Second edition, 2001.
5. D. S. Frankel. *Model Driven Architecture - Applying MDA to Enterprise Computing.* Wiley, 2003.
6. E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns.* Addison-Wesley, 1995.
7. E. Roman, S. Ambler, T. Jewell. *Mastering Enterprise JavaBeans.* Wiley, 2001.
8. G. Booch, I. Jacobson, J. Rumbaugh. *The Unified Modeling Language User Guide.* Addison-Wesley, 1998.
9. G. Hedin. Attribute Extension - A Technique for Enforcing Programming Conventions. *Nordic Jounral of Computing*, 1997.

10. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold. An Overview of AspectJ. *In Proc. of ECOOP '01, Springer-Verlag, LNCS 2072*, pages 327–353, 2001.
11. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin. Aspect-Oriented Programming. *In Proc. ECOOP '97, Springer-Verlag, LNCS 1241*, pages 220–243, 1997.
12. I. Zderadicka. CS CODEDOM Parser. `http://ivanz.webpark.cz/csparser.html`, 2002.
13. J. Liberty. *Programming C#*. O'Reilly, 2001.
14. J. Newkirk, A. Vorontsov. How .NET's Custom Attributes Affect Design. *IEEE SOFTWARE, Volume 19(5)*, pages 18–20, September / October 2002.
15. K. Czarnecki, U. W. Eisenecker. *Generative Programming.* Addison-Wesley, 2000.
16. D. Knuth. The Genesis of Attribute Grammars. *In Proc. of International Workshop WAGA*, 1990.
17. M. Abadi, A. Banerjee, N. Heintze, J. G. Riecke. A Core Calculus of Dependency. *In Proc. of the 26th ACM SIGPLAN-SIGACT on Principles of Programming Languages (POPL)*, pages 147–160, 1999.
18. M. Shomrat, A. Yehudai. Obvious or Not? Regulating Architectural Decisions Using Aspect-Oriented Programming. *In Proc. of Aspect-Oriented Software Development - AOSD 01*, 2001.
19. M. Voelter. A Collection of Patterns for Program Generation. *In Proc. EuroPLoP*, 2003.
20. M. Voelter, A. Schmid, E. Wolf. *Server Components Patterns, Illustrated with EJB.* Wiley & Sons, 2002.
21. N. H. Minsky. Why Should Architectural Principles be Enforced? *In Proc. of IEEE Computer Security, Dependability, and Assurance: From Needs to Solutions*, 1998.
22. .NET Framework MSDN Documentation. `ms-help://MS.VSCC/MS.MSDNVS/Netstart/html/sdkstart.htm`, 2002.
23. R. Paige. Future Directions in Program Transformations. *ACM Computing Surveys, Volume 28*, pages 170–170, 1996.
24. J. Prosise. *Programming Microsoft .NET.* Microsoft Press, 2002.
25. V. Cepa. Implementing Tag-Driven Transformers with Tango. *Proc. of 8th International Conference on Software Reuse - LNCS 3107*, pages 296–307.
26. V. Cepa, M. Mezini. Language Support for Model-Driven Software Development. *(Editor M. Aksit) Special Issue Science of Computer Programming (Elsevier) on MDA: Foundations and Applications Model Driven Architecture*, 2004.
27. .NET Attribute Dependency Checker (ADC) Tool. `http://www.st.informatik.tu-darmstadt.de/static/staff/Cepa/tools/adc/index.html`, 2003.
28. W. Taha, T. Sheard. Multi-stage Programming. *ACM SIGPLAN Notices*, 32(8), 1997.
29. Meta Object Facility (MOF) Specification Version 1.4. `http://www.omg.org`, 2002.
30. PMD Java Source Code Scanner. `http://pmd.sourceforge.net`, 2003.
31. Borland TogetherJ. `http://www.borland.com/together/`, 2003.