Proceedings of the 7th International Conference on Applied Informatics
Eger, Hungary, January 28–31, 2007. Vol. 2. pp. 277–284.

Runtime access control in C#*

Krisztián Pócza, Mihály Biczó, Zoltán Porkoláb

Dept. of Programming Languages and Compilers
Fac. of Informatics, Eötvös Loránd University
e-mail: kpocza@kpocza.net, mihaly.biczo@t-online.hu, gsd@elte.hu

Abstract

Compile time or runtime visibility and access control checking is the key part of modern languages and runtime environments. They enforce responsibility separation, implementation and security policies. The Eiffel programming language defines sophisticated selective access control, but most modern programming languages like C++, C# and Java do not have this feature only a subset or combination of the following access modifiers: public, private, protected, internal and friend. The .NET Framework enforces some security policies in runtime called Code Access Security but this additional mechanism is capable only to restrict external resource access for programs written in any .NET-language like C#.

In this paper we describe the existing access control features of the C# language then show a scenario where a more sophisticated access control is required. We introduce a method level access control checking mechanism to C# which is able to enforce Eiffel-like selective export in runtime. Our implementation does not require the modification of the compiler and the caller, only the callee, and introduces minimal syntactic overhead. It can be a practical solution for modular systems where runtime security is important.

Keywords: object-oriented programming, access control, runtime, C#

MSC: 68N15 Programming languages

1. Introduction

Compile time or runtime visibility and access control checking is the key part of modern languages and runtime environments. The two main fundamental concepts of object oriented programming languages are encapsulation and inheritance [6]. Encapsulation means that the programmer collects related services and data in a single type or class and enforces strong cohesion between them while hides the internal implementation and data structures from the outside world. Inheritance is a parent child relationship between types or classes which attract polymorphism

*Supported by the Hungarian Ministry of Education under Grant FKFP0018/2002.

based on the virtual nature of methods. Access control checking can be performed in compile time and in runtime also.

Using access control we can determine whether particular classes and class members can be accessed from and published to the outside world. Access control enforces a built-in security system in point of class member access. Different parts and modules can see and invoke only a slice of data stored and services implemented in the program.

In this paper we describe the existing access control features of different programming languages especially the C# language, then show a scenario where a more sophisticated access control is required. We introduce a method level access control checking mechanism to C# which is able to enforce Eiffel-like selective export [2] in runtime. Our implementation does not require the modification of the compiler and the caller, only the callee, and introduces minimal syntactic overhead. It can be a practical solution for modular systems where runtime security is important. After that we show the performance of our solution and discuss the results.

2. Access control features of different languages

In this section we describe the existing access control features of different programming languages especially the C# language.

The C++ language can be regarded as the ancestor of many modern programming languages like C#, D and Java therefore we describe the access control features of the C++ language [3] first. Access control can be performed at class level and class member level also. The default class level access control in C++ is private which can be changed to public. It means that by default a class is not visible to the outside world but this behaviour can be overridden. Class members can be declared as public, private and protected. Public members are reachable from any method of any class while private members can be reached only from the current class methods. Class members declared as protected can be reached from the methods of the current class and the derived classes. In C++ there is a special kind of methods and classes called friend. Friend methods and classes can access all the private and protected members of a particular class which accepts them as friend. C++ has three kinds of inheritance mode (public, private and protected) which controls the access control of the inherited members in the derived class. This results in a matrix which can be seen in Table 1.

	Access modifier in the base class		
Inheritance mode	**Public**	**Private**	**Protected**
Public	*public*	*private*	*protected*
Private	*private*	*private*	*private*
Protected	*protected*	*private*	*protected*

Table 1: Inheritance modes and access modifiers in C++

Members that become public in the derived class can be accessed in the derived class and also from the outside world. Members that become protected can be accessed only in the derived class, while members that become private are hidden in the in the derived class and cannot be accessed from the outside world.

The Java programming language [7] does not have such complex access control features like C++. Class level access control in Java can be public or package-private. Package-private visibility means that a particular class is visible only in the package where it was defined. At class member level, Java has four levels of access control: public, private, protected and package-private (default). In contrast to C++, Java does not have different kinds of inheritance levels. The inherited class members of the base class behave in the same way as in the public inheritance mode of C++.

The Eiffel language has a very different approach [2] to access control that the previously described C++ and Java languages. It has selective export, which means that different class members (features in Eiffel's terminology) can be accessed from different set of classes. For example we can define that feature "A" can be accessed by everybody, feature "B" cannot be accessed from the outside world, while feature "C" can be accessed from "Class1" and "Class2", and feature "D" can be accessed from "Class1" and "Class3".

In the Ruby programming language [4] access control is determined dynamically, as the program runs because Ruby is a fully interpreted language. The access control implementation of Ruby is very near to the other popular object oriented languages; therefore it is only interesting because the access control checking is done in runtime not in compile time.

In C# [1], the access control mechanism is very similar to Java's implementation. C# has two levels of access control: class and class member level. A class can be public, private and internal. Public classes are accessible by everybody; private classes can be accessed from the current namespace. Internal (default) classes behave in the same way as package-private classes in Java; they are accessible from the current assembly. At class member level C# has five different access modifiers: public, private, protected, internal and protected internal. Public, private, protected and internal members behave in the same way as in Java. Protected internal members behave as if they were protected and internal at the same time.

3. Motivation

In the previous section we described how modern object oriented programming languages implement access control. In this section we will show that the current implementation of access control in C# is insufficient in some scenarios, while Eiffel's implementation would be sufficient. Consider the C# code fragment:

Listing 1. C# code fragment with insufficient access control.

```
class Book
{
    public string GetTitle() { ... }

    public double GetPrice() { ... }
    public void SetPrice(double price) { ... }
    public BookReaderStream Read() { ... }
}
```

The Book class shown in Listing 1 can return the title and the price of the book it represents, it can give a stream which is responsible to retrieve the book's content and it exposes a method which can set the price of the book. These methods have to be public because we want them to be reachable from the outside world.

We can ask the following questions:

1. Should everybody have the right to set the price of the book?

2. Should everybody have the right to read the book?

The answer of these questions is clearly no. Only somebody from the bookstore can set the price of the book and only the reader of the book can read the book.

If we were using the Eiffel programming language we could easily distinguish which class members are accessible by different callers.

4. Implementation

In the previous section we described a scenario where the current access control features of C# are insufficient. In this section we describe the aims and conditions we would like to reach and match in the context of access control in C#.

The most important aim is to implement a mechanism that can restrict the access of public methods in the same way as Eiffel's selective export. The compiler cannot be modified because we would like to use the authentic Microsoft C# compiler which cannot be altered. Because we cannot modify the compiler then access control checking can be done only in runtime like in Ruby. We would like to reach our aim with minimal syntactic overhead. Because the solution should be easy to read and understand; therefore we have chosen attributes and inheritance. (Attributes are standard language elements of C# which can annotate some static information about different language elements like classes, methods, etc.) When an unauthorized access is encountered an exception should be thrown. Consider the following example:

Listing 2. C# code fragment with runtime access control.

```
class Book : RuntimeAccessControlBase
{
    [AllowedCallerClass(typeof(Reader))]
    [AllowedCallerClass(typeof(BookStore))]
    public string GetTitle() { ... }

    [AllowedCallerClass(typeof(Reader))]
    [AllowedCallerClass(typeof(BookStore))]
    public double GetPrice() { ... }

    [AllowedCallerClass(typeof(BookStore))]
    public void SetPrice(double price) { ... }

    [AllowedCallerClass(typeof(Reader))]
    public BookReaderStream Read() { ... }
}
```

In Listing 2 attributes are used that indicate which caller types can access the particular methods, and the class is inherited from the RuntimeAccessControlBase class. We have not modified the compiler and do the job with minimal syntactic overhead. The attributes only declare which caller types can access the methods but cannot check; therefore a custom call interception mechanism should be incorporated into the system. This way a custom implementation could check if the caller is in the list of allowed callers specified by the attributes.

4.1. High-level implementation

Because the attributes only declare which caller types are able to reach the particular methods, the *RuntimeAccessControlBase* class has to have some special behaviour where the access control can be implemented.

Listing 3. The implementation of RuntimeAccessControlBase class.

```
[Intercept]
public class RuntimeAccessControlBase : ContextBoundObject
{
}
```

Consider the implementation of the *RuntimeAccessControlBase* class in Listing 3: As it can be seen the *RuntimeAccessControlBase* class is inherited from *ContextBoundObject* and has the Intercept attribute. *ContextBoundObject* [1, 5] is a system class which resides in the System namespace of the Microsoft .NET Base Class Library and it is responsible to provide a dedicated context to every

object which inherits from *ContextBoundObject*. The context is created during the activation of the context-bound objects, and destroyed when the object becomes garbage. Usage rules can be added to these objects by specifying an attribute inherited from *ContextAttribute*. In our case this attribute is called *Intercept* which is implemented by the *InterceptAttribute* class. These usage rules are enforced when method calls are intercepted by the .NET Common Language Runtime.

Far behind the *Intercept* attribute there is the implementation that is responsible to check that the caller is from the list of allowed callers that the *AllowedCallerClass* attributes specify.

4.2. Low-level implementation

As we have mentioned previously the *InterceptAttribute* class is inherited from the *ContextAttribute* [5] class and a new context is created during the activation of the context-bound object. The default constructor of *InterceptAttribute* has to call the constructor of *ContextAttribute* which has one string type parameter and pass a unique identifier (in our case "Intercept"). Here we have to override the *GetPropertiesForNewContext* method which is called at activation time and has one *IConstructionCallMessage* type parameter called *ctorMsg*. The *ctorMsg* object has a *ContextProperties* collection, and a new *InterceptProperty* class instance is added to this collection. The *InterceptAttribute* class can be seen in Listing 4.

Listing 4. Implementation details of InterceptAttribute attribute.

```
[AttributeUsage(AttributeTargets.Class)]
class InterceptAttribute : ContextAttribute
{
   public InterceptAttribute()
       : base(InterceptProperty.IDENTIFIER)
   {
   }

   public override void GetPropertiesForNewContext(IConstructionCallMessage ctorMsg)
   {
       ctorMsg.ContextProperties.Add(new InterceptProperty());
   }
}
```

The *InterceptProperty* implements the *IContextProperty* and the *IContributeObjectSink* interfaces which enforce us to add some methods and properties to the class like *Name*, *IsNewContextOk* and *GetObjectSink*. The *Name* property should return the same unique identifier specified before ("Intercept"), the *IsNewContextOk* should return true. The *GetObjectSink* has two parameters:

1. A MarshalByRefObject [1, 5] called obj which specifies a remote reference to the original object.

2. An IMessageSink called nextSink which specifies the next message sink.

The method also returns an IMessageSink. In our case we return a new InterceptSink type class instance which accepts the nextSink property in its constructor (Listing 5).

Listing 5. GetObjectSink method.

```
public IMessageSink GetObjectSink(MarshalByRefObject obj, IMessageSink nextSink)
{
    return new InterceptSink(nextSink);
}
```

Every method call is represented by a message and returns a message also which represents the return value. The *InterceptSink* class implements the *IMessageSink* interface.

The most important method we have to implement in this class is the *SyncProcessMessage*, which accepts an *IMessage* parameter (represents the method call) and also returns an *IMessage* (represents the return value). The method enforces the runtime access control rules described by the *AllowedCallerClass* attributes by calling *CheckSelectiveVisibility*, and calls the next sink (Listing 6).

Listing 6. SyncProcessMessage method.

```
public IMessage SyncProcessMessage(IMessage msg)
{
    CheckSelectiveVisibility((msg as IMethodCallMessage).MethodBase);

    return _nextSink.SyncProcessMessage(msg);
}
```

The *CheckSelectiveVisibility* method simply loops through the attributes of the currently intercepted method stored in the *MethodBase* member of msg and checks if the direct caller is specified by any of these attributes. If so then it allows to continue the program otherwise throws an *InvalidCallerException* exception.

5. Performance results

We created a simple class with empty methods and enabled the runtime access control checking method described in this article. We used empty methods to be able to measure the pure performance of our solution. It performed about 4000 calls on a 2.6 Ghz Pentium 4 computer using the Microsoft .NET Framework 2.0. We can ask the question if this is eligible or not. The answer is that it depends on the use case. If the methods are simple class member variable accessors then our solution is not eligible. However if the methods perform some database, file or network access then the performance is eligible because an average database, file or network operation can take much more time than 1/4000 seconds.

6. Further work

We have shown a new runtime access control checking method for the C# language which supports any .NET-language because the cross-language nature of the .NET Framework.

In the current implementation only single caller class type checking is implemented but we can extend it to support a class type and every type that is inherited from the specified ones.

Properties are special parameterless methods in the C# language that are generally responsible for getting and setting a single class member variable; therefore they are similar to Java's getter/setter methods. It is important to add read/write access support to our runtime access control solution to fully support the get/set properties.

Classes can be placed in different roles at runtime or by specifying a custom interface at compile time to be able to add role based security features to our solution.

We can analyze the performance issues of our solution and fix it or maybe find another implementation way.

References

[1] Albert, I. (et. al. ed.), A .NET Framework és programozása, *Szak*, (2004).

[2] Meyer, B., Eiffel, The Language, *Prentice Hall*, (1991).

[3] Stroustrup, B., A C++ Programozási nyelv, Hungarian translation (Z. Porkoláb et. al. ed.), Kiskapu, (2001).

[4] Thomas, D., Fowler, C., Hunt, A., Programming Ruby, The Pragmatic Programmer's Guide, Second Edition, *Addison Wesley Longman*, (2001).

[5] Lövy, J., Programming .NET Components, *O'Reilly*, (2003).

[6] Nyékyné, G. J. (et al. ed.), Programozási nyelvek, *Kiskapu*, (2003).

[7] Nyékyné, G. J. (et al. ed.), Java 2 útikalauz programozóknak 1.3, ELTE TTK Hallgatói Alapítvány, Budapest, Hungary, (2001).

Krisztián Pócza, Mihály Biczó, Zoltán Porkoláb
Dept. of Programming Languages and Compilers
Fac. of Informatics, Eötvös Loránd University
Pázmány Péter sétány 1/c.
H-1117 Budapest
Hungary