Design and Implementation of a high-level multi-language .NET Debugger

Dennis Strein
Omnicore Software, Werderstr. 87,
76137 Karlsruhe, Germany
MSI, Software Technology Group,
Växjö University, Sweden
strein@omnicore.com

Hans Kratz
Omnicore Software, Werderstr. 87,
76137 Karlsruhe, Germany
kratz@omnicore.com

ABSTRACT

The Microsoft .NET Common Language Runtime (CLR) provides a low-level debugging application programmers interface (API), which can be used to implement traditional source code debuggers but can also be useful to implement other dynamic program introspection tools. This paper describes our experience in using this API for the implementation of a high-level debugger. The API is difficult to use from a technical point of view because it is implemented as a set of Component Object Model (COM) interfaces instead of a managed .NET API. Nevertheless, it is possible to implement a debugger in managed C# code using COM-interop. We describe our experience in taking this approach. We define a high-level debugging API and implement it in the C# language using COM-interop to access the low-level debugging API. Furthermore, we describe the integration of this high-level API in the multi-language development environment X-develop to enable source code debugging of .NET languages. This paper can be useful for anybody who wants to take the same approach to implement debuggers or other tools for dynamic program introspection.

Keywords

Debugger, CLR, multi-language, C#, COM-interop, Rotor

1. INTRODUCTION

Tracking execution and examining the internal state of a program are important techniques for every developer. They can be used with debuggers to find bugs and unintended behavior. But they can also be used in other sorts of dynamic program introspection tools. A high-level debugger should provide a defined user experience regardless of the underlying technology. The developer who examines a running program cannot be bothered with the low-level intricacies of the underlying debugging API.

The Microsoft .NET Common Language Runtime (CLR) provides a low-level debugging API, to implement such tools. Using this API directly is difficult. First the API is not easy to use from a technical point of view, because it is implemented as a set of COM interfaces instead of a managed API. Thus, it cannot directly be used in managed C# [Hei04a] code. Also the low-level debugging API has no notions of high-level programming languages or debugging functionality. This has to be implemented using low-level features.

This paper describes how these problems can be solved. We describe our experience in defining a high-level debugger API and implementing it in managed C# code using COM-interop to access the low-level CLR debugging API. Furthermore, we describe the integration of this high-level API in the multi-language development environment X-develop [Omn04a] to enable debugging of .NET languages.

The paper is structured as follows: Section 2 gives an overview of our architecture. Section 3 describes the supporting CLR debugging technologies. Section 4 explains how to use COM-interop to create a managed wrapper for the low-level API. Section 5 describes at this API and how to implement high-level debugging features like breakpoints, stepping and variable introspection. Section 6 outlines the integration of these high-level features into the multi-language development environment X-develop to create a full-fledged interactive debugger. Section 7 discusses related work. Finally, we summarize this paper in Section 8.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

.NET Technologies'2005 *conference proceedings,*
ISBN 80-86943-01-1
Copyright UNION Agency – Science Press, Plzen, Czech Republic

2. ARCHITECTURE

This section gives an overview of the architecture of our debugger.

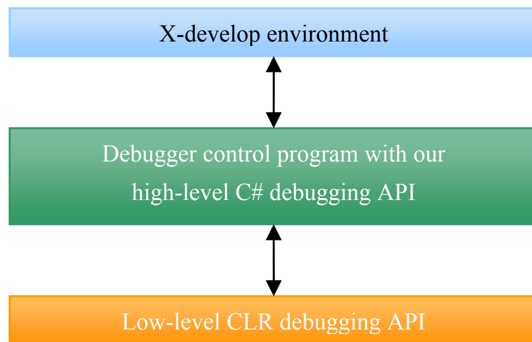

Figure 1. Architecture of our debugger

Design goals

The architecture should fulfill multiple design goals:

1. The main goal is to provide a user interface which enables all features necessary for conventional debugging from within the IDE. This functionality should follow the user's expectations. The developer should be able to set breakpoints in the source code at which execution of the whole program is suspended automatically. Once suspended the developer can switch between different threads of execution, display the current stack trace and step through the source code. When the developer uses single-stepping all threads of execution must be resumed to avoid deadlock situations. The developer also needs to be able to suspend execution at any time to inspect what the program is doing at that time. The interface must be powerful enough to allow a complete examination of the program state.

2. We want to integrate the debugger into the multi-language development environment X-develop [Omn04a]. X-develop supports C#, J# and Visual Basic and has an open API to extend it for new languages. Thus, it is important that the debugger also supports multiple languages.

3. We want to implement a high-level debugger API which provides a clean interface to the IDE hiding all the runtime-specific peculiarities of the low-level debugging API.

4. The debugger should be integrated in a way that provides maximum independence from the IDE. Even if the debugger interface ceases to function, the IDE should not be affected.

System Architecture

Figure 1 shows our architecture. On top, there is the X-develop environment that communicates with the debugger control program written in C#. This control program uses a high-level API which provides the desired high-level debugging functionality. The implementation of this API is based on the low-level CLR debugging services. The implementation is described in detail in section 4.

3. SUPPORTING TECHNOLOGIES

This section gives an overview of the CLR debugging services and other supporting technologies.

CLR Debugging Services

The Common Language Runtime provides low-level debugging services for runtime control and program introspection. Additionally, it defines a set of notifications for specific events that may occur during the execution of a program. The CLR debugging services are implemented as a set of COM interfaces. The program being debugged runs in its own Win32 host process. In the same process there is a special helper thread that communicates with the debugging services.

Symbol Manager

The CLR and the CLR debugging services know nothing about high-level programming languages. It knows only of the intermediate language (IL).

However, there is a mechanism for mapping source code to IL code and vice versa. The compilers for the various .NET languages store the mapping information in a separate program database file (PDB). This mapping information can be used for mapping between lines in the source code and positions in the IL code or for mapping between variable names and their respective addresses. The component that allows access to this information is called the symbol manager. The symbol manager API is part of the .NET core libraries. It can be found in the namespace `System.Diagnostics.SymbolStore`.

Additionally to the PDB files the executable files themselves contain information describing method names and signatures, class names, etc. This information is called metadata. It can also be used for source-to-IL mapping. For example it is possible to determine all the fields of a given class using metadata. The metadata API is a COM API like the debugger API.

One key benefit of using the compiler provided mapping information is that this information can be accessed uniformly for all programming languages. Thus, it provides support for multi-language debugging without any further work per

programming language. The work is done in the compilers.

For basic debugging functionality this information is sufficient. For more advanced applications a more detailed mapping might be desired. This would require additional static analysis of the source code in the compiler. Examples are expression-level stepping in debuggers or back-in-time debuggers.

4. USING COM INTEROP TO ACCESS THE CLR DEBUGGING SERVICES

Although the CLR debugging API is a classic COM API it is possible to implement a debugger in managed code using COM-interop. The advantage of this approach is that we can use C# (or any other managed .NET language) to implement the debugger. In this section, we give a step-by-step description how to achieve this.

About COM-interop

COM-interop is a technology to use classic COM APIs from managed code. This is done by creating managed wrapper classes representing the COM interfaces. This wrapper classes can than be called like normal managed classes. When calling a COM method from C# code, the CLR will internally marshal the arguments and return values to/from the COM object. Creating an instance of a COM class can be done in C# by simply creating a new instance using the `new` keyword. Internally, this causes a call to the native method `CoCreateInstance`.

Wrapping the debugger COM API

The preferred way to create wrapper classes is to use the tool `TlbImp.exe` that is included in Microsoft's .NET framework software development kit (SDK). This tool reads a COM type library definition file (TLB) and converts it to a managed dynamic link library (DLL) containing the wrapper classes. The file `cordebug.tlb` that is part of Microsoft's .NET framework SDK contains the definition of the debugger API. To create a wrapper assembly for this file we initially use `TlbImp.exe` to create a wrapper DLL. However, in this special case the DLL will not be complete. On the one hand there are classes missing that cannot be automatically converted by `TlbImp.exe`, on the other hand even some definitions in `cordebug.tlb` are not complete. To solve this problem we disassemble the wrapper DLL using `ILdasm.exe`. This tool is an intermediate language disassembler and is also part of the SDK. The result is an editable assembler version of the DLL. We can now add the missing classes by hand and adapt incomplete method signatures. Afterwards we use the SDK assembler `ILasm.exe` to create a DLL once again from the assembly file.

The classes in our wrapper DLL can now be used from C# code to create a high-level debugger API. We describe the classes in detail in section 5.

Our approach works with .NET 1.1 and .NET 2.0 depending on which version we want to target. The Rotor Shared Source CLR [Mic02a] implements the `ICorDebug` COM interface as well and can be used in place of a MS .NET framework.

Wrapping the metadata COM API

It is also possible to write a wrapper class in C#. Since the required metadata API is quite small we choose this approach. We only have to write a wrapper for the `IMetadataImport` interface. A C++ header file containing the definition can be found in the file `cor.h`, which is part of the SDK. A wrapper in C# consists of a single C# interface, which contains the same methods as defined in cor.h. This interface has to be marked with the `ComImport` attribute as well as the correct `Guid` attribute. The Guid of the `IMetadataImport` interface can be found in the file `cor.h`. Now we can use the C# wrapper class to access the metadata of assemblies.

5. IMPLEMENTATION OF A HIGH-LEVEL API

This section describes how to use the low-level API to implement a high-level debugging API, which is suited for integration into a development environment. Our high-level API allows to run programs, set breakpoints in source code, step single lines, introspect variables defined in the source code and to browse the fields of objects. The low-level API on the other hand provides access to the runtime and is not limited to our particular use case. In the following sections we will describe in detail on how to implement specific features affiliated with debugging. Figure 2 shows the architecture of our debugger and the high-level debugging API implementation.

Initializing the debugger

The first thing the debugger has to do is to create an instance of the `ICorDebug` interface. This is done in a completely different way in .NET 1.1 compared to .NET 2.0.

COM-activation is used in .NET 1.1. COM-activation is done in C# by simply creating an object of the COM wrapper class. In our case, `new CorDebugClass()` will create the correct class, which is an instance of the `ICorDebug` interface.

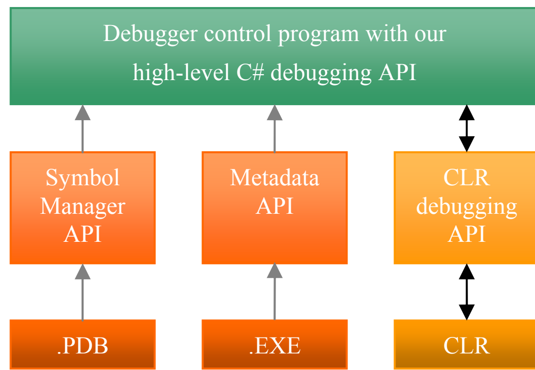

Figure 2. Implementation Overview

However, this causes problems. If one has a .NET 1.1 debugger debugging a .NET 1.1 program, the .NET 1.1 implementation of `ICorDebug` will be used. As soon as .NET 2.0 is installed, that scenario is automatically updated to use the .NET 2.0 implementation of `ICorDebug`. Now, if the .NET 2.0 implementation is slightly different than the 1.1 implementation, installing the 2.0 version breaks the 1.1 debugging scenario.

Thus, since version 2.0, the debugger has to create the `ICorDebug` object using the method `CreateDebuggingInterfaceFromVersion`. This method takes the desired .NET version as an argument. The method is defined in `mscoree.dll`, which is part of the .NET framework. In C# this method can be called by defining an extern method.

With the Rotor Shared Source CLR [Mic02a] the `ICorDebug` object can be obtained in the same way as for the .NET 1.1 framework. But before this method can be used the `mscordbi.dll` of Rotor has to be registered as a COM server using the `regsvr32` tool. Unfortunately this is just the scenario which the second method was designed to avoid: Once the Rotor `mscordbi.dll` is registered the MS .NET 1.1 framework `ICorDebug` object can no longer be created using COM-activation.

The `ICorDebug` object is the entry point to all debugging services. The debugger has to call the `Initialize` method of the `ICorDebug` object before doing anything else.

Handling events

The CLR will notify the debugger whenever certain events occur. To make this possible the debugger has to provide an implementation of the `ICorDebugManagedCallback` interface. This interface has to be registered using the `SetManagedHandler` method of the `ICorDebug` object. The registered implementation will only receive events that occur when debugging managed code. There is also a `ICorDebugUnmanagedCallback` interface that can be used for debugging unmanaged code.

Whenever an event is raised the affected process will be suspended. This allows the debugger to handle these events in an appropriate way. Afterwards the affected process has to be resumed. The process is passed as an `ICorDebugProcess` object to the corresponding interface method. The debugger has to call the `Continue` method of this object to resume execution. This has to be done for all events even if they do not require special handling. Otherwise the execution will not continue.

There is one event that needs special treatment. That is the `CreateAppDomain` event. It is called when the CLR application domain of the process is created. The method will receive an `ICorDebugAppDomain` object representing the application domain. In order to receive further events it is necessary to call the `Attach` method of this object.

We will describe some other relevant events in the following the sections as well.

Creating a process

The `ICorDebug` interface provides the method `CreateProcess` to create a process to debug. This method takes essentially the same arguments as the common Win32 method with the same name. The `CreateProcess` method returns an `ICorDebugProcess` object representing the process. The process will be created asynchronously after the call and the `CreateProcess` method of the `ICorDebugManagedCallback` interface is called by the debugger once the process has actually been created. As with all events the process will be suspended after this event.

Suspending and resuming the process

Suspending and resuming program execution is a common debugger feature. A process can be suspended by calling the `Stop` method of the `ICorDebugProcess` object. This method takes an integer timeout parameter that should be set to some high value. Otherwise crashes of the CLR can occur.

To resume execution we use the `Continue` method of the `ICorDebugProcess` object.

Mapping between source and IL code

The next features are more difficult to implement than the previous ones. The reason for this is that we now need to map between source code and IL code. The CLR debugging API itself has no notion of source code. Instead, the mapping has to be done by the debugger. Section 2 describes how symbol information is generated by the compilers. We will now show how to access this information.

First, the `IMetadataImport` interface can be used to access metadata of a given module. For a given module represented by an ICorDebugModule object, we can get an `IMetadataImport` object by calling the `GetMetaDataInterface` method.

The `ISymbolReader` interface can be used to access mapping information form PDB files. The way to create an `ISymbolReader` object differs between .NET 1.1 and .NET 2.0.

In .NET 1.1 the debugger has to create a `SymBinder` object. This class is defined in `ISymWrapper.dll`, which consequently has to be referenced by the debugger. The `GetReader` method of the `SymBinder` object returns the desired `ISymbolReader` object.

In .NET 2.0 the `GetReaderForFile` method of the `SymbolBinder` interface that is part of the core library can be used.

For the core debugging features described in this paper the information provided by the metadata and symbol manager APIs is sufficient. The following sections show particular use cases.

Setting breakpoints

Breakpoints are set in certain positions in source files. With the CLR debugging API however, a breakpoint can only be set on a specific point in the intermediate language (IL) level. Hence, we have to implement the mapping between source code and intermediate code. To do this, we use symbolic information as described in the last section.

5.1.1 Source-to-IL mapping

To set a breakpoint with the debugging API, the IL position for a given position (line) in a source file is required. To achieve this, the debugger proceeds as follows: first it iterates all loaded modules, respectively the `ICorDebugModule` objects. For each module the debugger creates an `ISymbolReader` object to access source-to-IL mapping information as described in the previous section. Then we call the `GetDocuments` method to obtain all source files in the module. If the breakpoint source file is found in the module we can use the `GetMethodFromDocumentPosition` to obtain the method at the breakpoint position represented by an `ISymbolMethod` object. The `GetFunctionFromToken` method will then return an `ICorDebugFunction` object representing this method in the debugging API.

The next step is to map the line in the source code to the corresponding IL instruction. To do this we can once again use compiler generated information, so called sequence points. The sequence points of a method specify for each statement in the source code where it can be found in the IL code. Thus, the desired IL instruction can be found by iterating each sequence point and comparing its line number with the breakpoint line number.

The sequence points are delivered by the `GetSequencePoints` method of the `ISymbolMethod` object.

5.1.2 Setting the breakpoint

Once the source-to-IL mapping is done setting the actual breakpoint is possible. First the debugger calls the `GetILCode` method of the `ICorDebugFunction` object, which returns an `ICorDebugCode` object, representing the methods IL code. Then we call the `CreateBreakpoint` method of this object with the IL position as an argument. The breakpoint is now set and the debugged process will suspend once it is hit.

5.1.3 Handling breakpoint events

As soon as the execution of any thread in the CLR passes the breakpoint the whole process will be suspended and the `Breakpoint` event of the `ICorDebugManagedCallback` will be raised. This event contains an `ICorDebugThread` object representing the thread that has passed the breakpoint. We handle this event by raising an event in the debugger GUI. The GUI now has to show the affected thread, the source position it has stopped at, allow stepping the code and support introspection of variables and object contents. The implementation of these features is described in the next sections.

Accessing the stack trace

To show the current execution point when the debugger is suspended we need to access the stack trace with current IL positions of the affected thread. We then map this IL position to a position in a source file using sequence points.

A stack trace of a CLR thread is separated into a series of so called chains. Each chain contains a series of frames. We can use the `EnumerateChains` and `EnumerateFrames`

methods to access those. The result is a series of `ICorDebugFrame` objects. Each frame object contains the current IL position. To map these positions to source positions we use symbol information and sequence points as described in the breakpoint section.

Stepping source code

When the debugger has been suspended at a given line in the source code, it offers the possibility to step over the next line in the source code, i.e. executing just this line. Additionally, a step-in feature will step into the next method called by the stepped line. Finally, a step-out feature will execute the rest of the current method and will stop after the call to this method.

To implement stepping we proceed as follows: a call to the `GetActiveFrame` method of the current `ICorDebugThread` object will returns an `ICorDebugFrame` object. Now the debugger has to create a stepper object by calling the `CreateStepper` method, which returns an `ICorDebugStepper` object. The desired stepping behavior can be achieved by configuring this object.

5.1.4 Step-over

We use the `StepRange` method of the `ICorDebugStepper` object to specify the IL instructions we want to step over. In fact, this method takes the IL instructions that should not be stepped as an argument. To calculate those, the debugger once again uses the sequence points of the current method as described in the breakpoint section. The sequence points contain the information which IL instructions represent the source code line to be stepped.

5.1.5 Step-in

Step-in can be implement just like step-over with the difference of passing an additional argument to the `StepRange` method.

5.1.6 Step-out

Step-out does not require source-to-IL mapping. Instead we can just use the `StepOut` method of the `ICorDebugStepper` object.

5.1.7 Other stepping behavior

The debugging API is flexible enough to configure more stepping features than those described here. However, its main limitation is the lack of an appropriate source-to-IL mapping. For example if we want to step single expressions instead of statements, the provided mapping information is not sufficient. In this situation additional static source code analysis is required.

Accessing local variables

The debugger should show all variables defined at the current position, and their value. To do this, we first resolve the defined local variable names in the source code using the compiler generated source-to-IL mapping. This mapping will also give us the address of each variable, which can be then used to determine its value.

5.1.8 Resolving declared variables

To determine all declared variables at the current position the debugger first has to retrieve an `ISymbolMethod` object representing the current method. The variables are grouped into scopes in which they are defined. The root scope of the method is returned as an `ISymbolScope` object by the `RootScope` property of the `ISymbolMethod` object. The subscopes of a scope are returned by the `GetChildren` method. The variables of a scope are returned by the `GetLocals` method. The debugger will use these methods and search for declared variables. The `ISymbolScope` objects contain the start and end position in the source file. This allows to determine the declared variables at a given source position.

5.1.9 Accessing the value

To access the value of a local variable of an `ICorDebugFrame` object the debugger calls the `GetLocalVariable` method. This method takes the address of the local variable and returns an `ICorDebugValue` object representing the value.

5.1.10 Rendering values

`ICorDebugValue` is the base of a hierarchy of interfaces representing different kinds of values. For primitive values the `GetValue` method will return a pointer to the bytes representing the actual value. Note that in C# the use of unsafe code and the unsafe keyword are necessary to access this value. The next section describes how to access the content of values representing object references.

Accessing object contents

If a value is a reference to an object, we want to access the fields of this object with their values. Doing this recursively allows to access the complete program state.

A value of an object is represented by an `ICorDebugObjectValue` object. The `GetFieldValue` method of this object will return the value. The field is identified by an integer token. Again we have to use the source-to-IL mapping information to determine the declared fields with their token. This is done by using the `EnumFields`

and GetFieldProps methods of the IMetadataImport interface.

Conclusion

The previous sections described how to use the low-level CLR debugger API to implement features of a high-level debugger. We make them available via a high-level debugger API – each feature is provided by a particular method. The low-level API is not limited to this use case though. It can also be used to implement other tools for dynamic program introspection. There are also more features in the low-level API than those described here. For example it is possible to suspend and resume individual threads, or modify data in the debugged program. This is required to implement further debugging features or for other applications.

6. INTEGRATION IN X-DEVELOP

This section outlines the integration of the debugger functionality with the development environment X-develop.

Communication protocol

In order to achieve maximum separation between IDE and debugger, the debugger interface and the debugger control program run in different processes and communicate using sockets. This architecture also enables easy implementation of remote debugging later on. There are three types of packets used for communication between IDE and debugger control program: command packets, reply packets and event packets. After startup of the debugger control program the IDE sends command and request packets to the debugger control program which in turn. Those command and request packets are modeled around the use cases identified in the previous section. When the debugger receives a command packet it carries out the requested action without sending a reply. When the IDE requests information from the debugger control program, a reply packet is generated containing the result or an error flag if the information could not be obtained. When a breakpoint is hit or execution is suspended after a step operation, the debugger control program sends an event packet back to the IDE.

GUI

The GUI provides user access to the debugging functions. X-develop displays the source code of the debugged program in its editor and allows setting of breakpoints in particular lines.

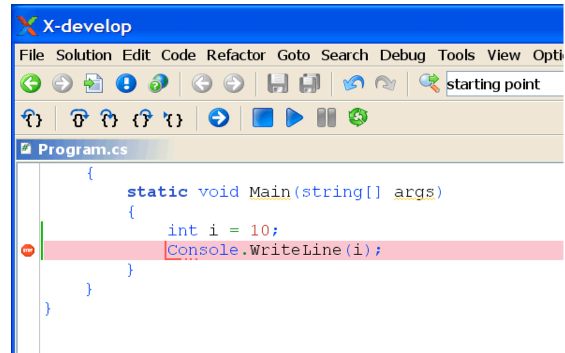

Figure 3. Breakpoints

The “Run-in- Debugger” function will start the debugger control program and set breakpoints by sending the appropriate command packets. Once a breakpoint has been hit, socket communication is used to obtain the stack trace and associated source position to show where the debugged program has stopped. Figure 3 shows this scenario.

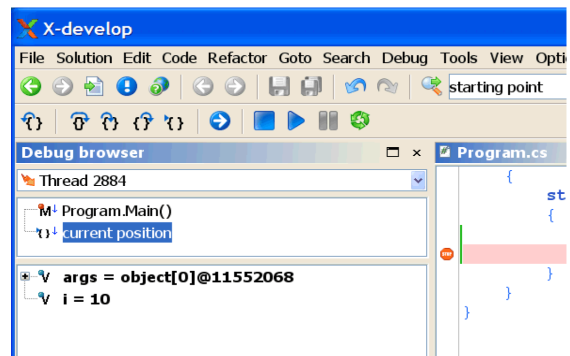

Figure 4. Variables

The user can continue program execution at any time using the Continue function. It is also possible to step through the program using the presented stepping functions. Additionally, all variables declared at the current position will be shown together with their value in a tree widget – see Figure 4. If the value is an object reference it may be further expanded to see the fields of the object and their respective values.

Experience

The integration in X-develop allows testing the performance and stability of the debugger. Our experience was positive:

1. Except for initial hurdles with COM-interop the implementation was straightforward.

2. Real-world stability of the debugger implementation was good. All functions work as intended. Debugging multi-threaded applications works as well as simple single-threaded applications.

3. Debugger responsiveness is excellent. We measured the “stepping speed”. This is the time between pressing the step button, execution of the step inside the debugger and the callback event with the new position. The measured time was always between 50 and 500 milliseconds. This is sufficiently fast for a responsive user experience.

7. RELATED WORK

The CLR debugging API is explained in detail in the documentation accompanying the .NET SDK. While being a comprehensive guide to the low-level API, it lacks information on how to put together a working debugger. Neither examples nor a tutorial are included.

Jon Shute published a series of articles on how to write a debugger with .NET using the CLR debugging API [Shu04a]. Unfortunately, the articles only cover a few details and uses example code written in C++.

The .NET SDK contains the source code of CorDbg - a C++ command line debugger using the CLR debugging COM API directly. It has no high-level API abstraction nor is it written in managed code.

Microsoft .NET 2.0 provides the source code of a command line debugger (Mdbg) that is also written in managed C# code. This tool also uses COM-interop to access the native debugging API. However, this tool does not include any documentation how the integration of the COM classes is performed. It only works with the 2.0 framework and it does not provide a high-level API abstraction. Furthermore, our architecture can easily be extended to support remote debugging and it offers a stronger separation between the debugger and the debuggee.

8. CONCLUSION AND FUTURE WORK

We have described the design and implementation of a high-level multi-language debugger for the .NET CLR. One advantage of our approach is that it allows to use managed C# (or any other .NET language) to implement the debugger. This can be useful for everybody who wants to take the same approach to implement debuggers or other tools for dynamic programming introspection.

We integrated the debugger in the development environment X-develop, but it is not limited to this particular use case.

The implementation of the high-level debugging API for Mono using `the Mono.Debugger` low-level API is underway.

The CLR debugging services provide rich access to the state of executed programs. The main limitation is the lack of additional source-to-IL mapping information. The information generated by the compilers for the various .NET languages is sufficient to implement the basic functionality. But for more advanced applications, additional static source code analysis is required. A good example for such an application is a back-in-time debugger [Kra04a] [Omn04b]. Such a debugger allows stepping backwards by replaying the previously recorded program execution.

9. REFERENCES

[Hei04a] A. Hejlsberg, S. Wiltamuth, P. Golde. The C# Programming Language. Addison-Wesley, 2004.

[Kra04a] Hans Kratz. Implementierung eines Debuggers mit Rückwärtsschrittfunktion. Diplomarbeit. 2004. In german.

[Mic02a] Microsoft. Shared source common language infrastructure. Published on the web at http://msdn.microsoft.com/net/sscli, 2002.

[Omn04a] Omnicore Software. X-develop. Published on the web at http://www.x-develop.com, 2004.

[Omn04b] Omnicore Software. CodeGuide. Published on the web at http://www.omnicore.com, 2004.

[Shu04a] Jon Shute. Ramblings about .NET and debuggers. Published as a web page at http://blogs.chimpswithkeyboards.com/jonshute/, 2004.