Towards a CLI Assembly Format for Embedded Systems

Bernhard Rabe
Hasso-Platter-Institute, University of Potsdam
P.O. Box 90 04 60
14440 Potsdam, GERMANY
bernhard.rabe@hpi.uni-potsdam.de

Keywords: CLI, assembly format,memory footprint, embedded systems

Abstract

Bytecode-based Middleware and virtual execution environments have become popular in development for embedded Systems. The ECMA/ISO Common Language Infrastructure (CLI) specifies a bytecode-based execution environment (Common Language Runtime) and a comprehensive class library. As Microsoft's CLI implementation the .NET Compact Framework was built for high-end mobile devices. It would be reasonable subset of this technology to low-end embedded systems as well. Often CLI applications use a small subset of the CLI class library, but the whole memory footprint is basically determined by the class library. The loose coupling of CLI applications and utilized library features requires dynamic linking before execution. The process of dynamic linking causes long startup times and high memory consumption in addition to the high memory footprint of the CLI class library. For application in low-end embedded systems the memory footprint must be minimal. To overcome memory requirements of the class library, an minimal application format that includes all essential class library features is reasonable. Self-contained CLI assemblies as an approach for size-optimized deployment format for low-end embedded systems, are presented in this paper.

1. Introduction

Embedded systems have become almost ubiquitous in the last decade. High-level programming languages and bytecode-based execution environment have become popular in development of desktop systems. The *Common Language Infrastructure* (CLI) [8] as implemented in the .NET Framework [12] has been a popular platform for creating component-based applications, because of:

- Platform independence of bytecode-based executables
- Fine granular security restrictions
- Revisable code
- Component model

It would be beneficial if CLI applications could be executed on low-end embedded systems that are not covered by existing CLI implementation. .NET developers could then reuse their code for these systems instead of reimplementing their applications from the ground up using C or C++.

Embedded systems differ from desktop systems in various aspects: hardware resources are often limited in terms of memory size, processing power or power supply; software capabilities, faulty programs can crash the system, because memory protection is not available; capabilities for developer interaction, for debugging, or communication bandwidth are often limited. CLI technology is seamlessly integrated in Rapid Application Development tools as Microsoft's Visual Studio suite for desktop development just as for embedded development. Compiler and tools are available for multiple programming languages e.g. C#, C++ .NET, or Delphi. The CLI would offer the developer of the embedded systems the same advantages that they get on desktop systems.

Due to the predictable nature of the sandbox-mode execution of CLI instructions, programming errors never result in system crashes, but cause exceptions to be thrown. This allows for a simpler postmortem analysis of a fault. Due to the support for rapid prototyping, simulators for the target can be created more easily. Ideally, much of the code would only use standard library functions of the CLI, so that simulators are only necessary for the target-specific hardware. But the

CLI as implemented in the Microsoft .NET Framework, the Microsoft Compact Framework [11], or the Mono Project [16] does not meet the requirements of limited resources of low-end embedded systems.

The memory footprint of an executable assembly is calculated by the assembly itself, custom libraries used, the *Base Class Library* (BCL) and the *Common Language Runtime* (CLR). The first three items were focused on in this paper and an approach to reduce the memory footprint of an executable assembly in such a way that unused library features are not required to be present at runtime, is proposed.

This can be achieved by assembling an assembly with its used library features into a self-contained assembly. The self-contained assembly will contain only required library features and will become smaller than the combined libraries. Furthermore the number of referenced assemblies which are required to be loaded is reduced to the self-contained assembly itself.

This work is based on the PERWAPI [7] library, which is extended to the needs of creating self-contained assemblies.

The rest of this paper is structured as follows: Section 2 briefly reviews the Common Language Infrastructure. In Section 3 the mechanism of executing CIL-code is discussed in detail. Next, self-contained assemblies as approach for optimized memory footprints and predictable behavior in are presented in section 4. Section 5 gives an overview of related work followed by conclusions and future plans.

2. Common Language Infrastructure

The CLI standard specifies the executable format, a virtual runtime environment (Virtual Execution System (VES)) and a set of libraries as implemented in the Microsoft .NET Framework, *Shared Source Common Language Infrastructure* (SSCLI) [10], or in the Mono project. An VES is commonly named *Common Language Runtime* (CLR).

CLI executables, called assemblies are encoded in the *Common Intermediate Language*(CIL) instruction set. An assembly is the deployment unit of the CLI and may consist of multiple files (modules). Assemblies are embedded in Portable Executable (PE)-files, which is the binary format for Windows executables. An assembly is loose coupled with the BCL and other assemblies in a similar way to native applications and shared libraries.

CIL is a stream of bytecodes similar to processor instructions. Most opcodes are one byte long, a few 2 bytes long and may have a optional parameter (up to 8 bytes long). Every method consists of a header, a body and a possible a footer. To evaluate opcodes a stack is used. Bytecodes are located in the method body.

2.1. Metadata

Assemblies are equipped with metadata about references, types, methods... Metadata are organized in named streams and tables. Table rows may have references into other table rows. The following tables are mainly involved while executing an CIL code:

- *Assembly:* Assembly defined in the PE file.
- *AssemblyRef:* Assemblies required for execution.
- *TypeRef:* Used types defined in external assemblies. Every type in this table refers its resolution scope that is located in the *AssemblyRef*-table for the relevant cases.
- *TypeDef:* Contains all types that are defined within an assembly.
- *Method:* All methods that are declared by types in *TypeDef*-table. Every row in the Method-table is owned by one and only one row in the *TypeDef*-table.
- *MemberRef:* All methods or fields of external defined types that are accessed within the assembly. There is merely a 'forward-pointer' from each row in the *TypeRef*-table.

References in table rows are tokens to other table rows, to an heap entry or to an *Relative Virtual Address* (RVA) within the assembly. The size of assembly is mainly given by the size of the metadata. Costa and Rohou [5] show that metadata size varies from 40 percent up to 80 percent of the whole assembly size. The metadata split 70 percent to 30 percent into constant pool (heaps) and tables. Most of the contents in the #String heap are not needed by the CLR. For example variable names, property names are only used by the reflection API.

2.2. Version compatibility

To overcome problems resulting from different versions of dynamic libraries on Windows systems [2] the CLI introduced an version management that builds up on version numbers and public keys. An assembly version has four parts a major, a minor, a build and a revision number. A version may be specified with an attribute, but a version number is not mandatory. If a version attribute is not specified, a default version number of zeros is set. To make a assembly reference distinct the assembly must have strong name. Strong names

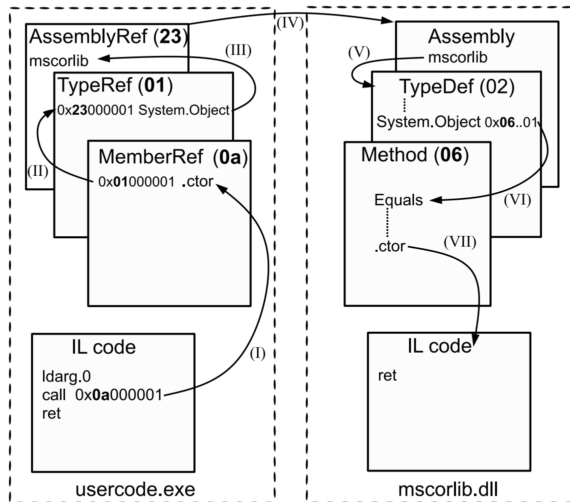

Figure 1. Resolving of an external method

guarantee name uniqueness by relying on unique key pair. All shared assemblies that reside in the GAC must have a strong name. The BCL provided by actual CLI implementations have all the same standard public key that does not require a private key to sign. This is done to provide vendor independent execution of assemblies. That means an assembly which has references to the BCL (mscorlib.dll) may behave differently with different BCL implementations.

3. Execution of .NET assemblies

Executing an method within an assembly the CLR needs to load all assemblies features referenced within that method. In fact the CLR loads all assemblies referenced by the assembly, even though they might not be needed most of the time the application is executed. Merging multiple modules into one reduces the number of modules which were loaded. Clearly this applies only if it is feasible e.g. source available. In terms of the CPU, assembly loads have fusion binding and CLR assembly-loading overhead in addition to the LoadLibrary call, so fewer modules mean less CPU time. In terms of memory usage, having fewer assemblies also means that the CLR will have less state to maintain. Creating the executable image requires dynamic linking of CIL code. This task is different for assembly internal and assembly external references. Figure 1 shows how CIL code of an external referenced method is located.

(I) An CIL operation (call) has a token operand that points to an *MemberRef*-table row.

(II) The *MemberRef*-table row contains the method name and a token into the *TypeRef*-table.

(III) Namepspace, type name and a token for the associated *AssemblyRef*-table row are included in the *TypeRef*-table.

(IV) The *AssemblyRef*-table row provides the name, the version number and the public key token of the related assembly.

(V) Within the referenced assembly the CLR looks into the *TypeDef*-table for the requested type. This has to be done by a linear search with string and signature comparison until the matching row is found.

(VI) The linear search for the matching method in the *Method*-table is optimized in the way that the start of the relevant rows is given by the *Method*-table token from the associated *TypeDef*-table row.

(VII) An Method-table row contains the RVA to the method definition within the PE-file.

This task is necessary for every external method. In comparison with an external method an single lookup in the *Method*-table is required to obtain the RVA of an method within the assembly. Recapitulating it has been reflected that loose coupling of assemblies and consequential external references cause the following properties:

- MEMORY CONSUMPTION: Each extern assembly must be loaded and metadata tables have to build up.

- PROCESSING POWER: Multiple indirections, linear search, string and signature comparison during reference resolving cause additional CPU Time in contrast with internal references.

- MEMORY FOOTPRINT: Combination of functionality into a single assembly (mscorlib.dll) cause a high memory footprint if a single type is referenced.

- REVISABLE CODE: CIL within an assembly can be inspected for validity. Extern assemblies especially the BCL may implemented differently and makes it impossible to predict the behavior of CIL code.

The above disadvantages can be minimized if all extern referenced functionality is assembled into a single assembly. This harms the loose coupling, but it allows lower memory footprints and analysis of CIL code within the assembly.

4. Self-Contained CLI assemblies

Dynamic linking has introduced loose coupling of applications and libraries. The process of linking application code and library functionality is done at loadtime alternatively at runtime. Update and maintenance of were simplified, but the level of compatibility receded. CLI assemblies also loose coupled with other assemblies. Generally applications do not use all provided library features that induces an higher memory footprint than necessary for execution. The self-contained assembly approach lifts up the problems with unused features of referenced assemblies.

The smallest available multi-purpose CLI implementation is the .NET Compact Framework, that has a memory footprint about 2 megabytes and is designed for high-end PDA and smartphone devices with 10th of megabyte memory. To run CLI on embedded systems equipped with less memory the self-contained assembly approach could be practicable.

An self-contained assembly has a minimal memory footprint, a predictable behavior based on CIL-code, and a reduced startup time.

The memory footprint of an assembly is calculated by the CLR, the relevant library assemblies and the assembly itself. In general every assembly uses BCL features (e.g. System.Object). The BCL is represented as *mscorlib.dll* [6]. But the mscorlib.dll provides additional features, which are not used by most assemblies. Independently from the amount of mscorlib.dll features used by an assembly, the memory footprint for the BCL is fixed. Self-contained assemblies do not need additional library assemblies and form together with the CLR the minimal footprint for an execution environment. This feature targets mainly memory restricted systems.

Prediction of execution behavior for self-contained assemblies is possible, because all executable CIL code is within the assembly. A static behavior evaluation can be done before runtime and allows for example prediction of memory consumption.

CIL code execution will be delayed by dynamic linking of assemblies at load time. The time is needed for loading assemblies and resolving references. Self-contained assemblies do not require additional assemblies and dynamic linking does not appear, therefore the startup time is shortened.

Figure 2 shows top down an C# program, the compiled CIL-code and the CIL-code of the self-contained alternative. The C# program declares an *Main*-method, that creates an instance of type Object and returns the value one.

Beneath the compiled CIL-code of the *Main*-method is indicated. This code was produced by the Microsoft .NET Framework C# compiler[1] with optimization enabled. The local variable `obj` disappeared, because it is not used furthermore. The instance of type `Object` is created with the `System.Object::.ctor()` call from the mscorlib assembly, indicated by the `[mscorlib]` prefix. Then the constant value one is returned.

The third figure illustrates the resulting CIL code if the second is assembled with the System.Object type into a self-contained assembly. The `System.Object::.ctor()` call does not leave the assembly scope. The rest of the program behaves the same. The CIL code illustrations were produced by the .NET Framework Ildasm tool and do not appear during the linking process.

The CIL-code cutouts differ also in the .maxstack value, because the Microsoft C# compiler generates an Fat-method header and the PERWAPI library an Tiny-

```
  ...
public static int Main(string[] args){
   Object obj=new Object();
   return 1;
}
  ...
```

```
  ...
.method public hidebysig static int32 Main(
    string[] args) cil managed
{
  .entrypoint
  .maxstack  1
  newobj     instance void [mscorlib]System.
      Object::.ctor()
  pop
  ldc.i4.1
  ret
}
  ...
```

```
  ...
.method public hidebysig static int32 Main(
    string[] args) cil managed
{
  .entrypoint
  .maxstack  8
  newobj     instance void System.Object::.
      ctor()
  pop
  ldc.i4.1
  ret
}
  ...
```

Figure 2. C# code, compiled CIL-code , self-contained CIL-code

[1] .NET Framework v1.1.4322: csc /optimize+ ...

method header. None of the requirements for an Fat-header as described in section 2 are satisfied, so the 1 byte Tiny header is an better alternative for size optimization.

The example demonstrates the possibilities of self-contained assemblies. The startup time was reduced (theoretically), because the mscorlib assembly won't loaded and reference resolving was shortened. Fewer loaded assemblies effect a scaled down memory consumption and the memory footprint.

4.1. Creating Self-contained Assemblies

Self-contained assemblies do not have any extern references. This means a CLR should able to execute an self-contained assembly without loading the BCL or other managed assemblies. In contrast to statically linked native binaries, the CLI abstracts from the operating system and underlying hardware. This fact makes it feasible to build a BCL independent CLI assembly.

To get a self-contained assembly, the relevant assemblies must be disengaged from type references to external assemblies. This work can be done by processing IL textual descriptions or by using an assembly manipulation library.

In the Micro.NET project we use the library approach, because the ILDASM approach requires a lot of text substitution and depends on available CIL framework tools. The Reflection API of the .NET Framework does not supports access to CIL code and writing of modifications. Microsoft's new compiler framework Phoenix allows assembly modifications within a compiler run. After evaluation of capabilities of different assembly manipulation frameworks, the Micro.NET linker finally bases on PERWAPI [7] developed at the Queensland University of Technology. PERWAPI provides an abstract representation of the PE-file embodied as object oriented structure. The library is implemented in C# and is available for free. PERWAPI was extended to support the creation of self-contained assemblies. Figure 3 shows the creation of self-contained assemblies with the Micro.NET linker tool and an optional configuration. The assembly on the left side references the BCL (mscorlib) and may have references to multiple custom libraries. The PERWAPI-based linker tool resolves references controlled by an optional configuration file. The configuration allows the instrumentation of the assembling process inside the linker tool. The source for an type to import could be set or types that should kept as references.

Every type defined in an assembly must be reviewed for the following list of elements:

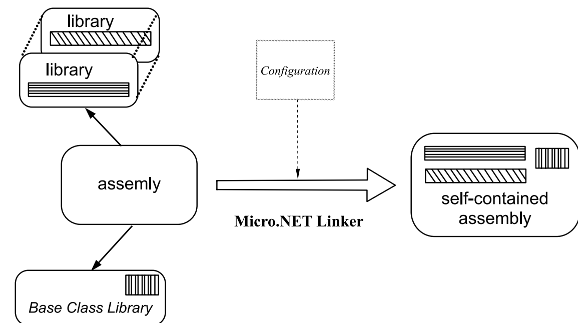

Figure 3. Creation of self-containd assemblies

Custom Attributes A Custom Attribute points to an type constructor method and contains optional constructor values. Attributes can occur at assembly, type, and method level.

Type A Type has an parent type except `System.Object` and may implement a number of interfaces. Methods describe operations that may be performed on that type. Fields are named subtypes of an type.

Interface Interfaces are special types that do not have a super type and contain no CIL code. Interface may extend other interfaces.

Method A Method is a named operation and is characterized by its parameter types. Besides the parameter types also the return type and possible Custom Attributes have to set to the resolved type. Local variables are unnamed subtypes within a method resolution scope. CIL code may have a type, method or field parameter. Exception clauses are defined by a code range and the type of the exception.

Event Event are handles like fields of an type.

CIL code The following types of IL codes must be reviewed for references to types, methods or fields references:

Type Op.: castclass, newarr, initobj, ...

Method Op.: call, calli, callvirt, newobj, ...

Field Op.: ldfld, ldflda, stfld, stflda, ...

The challenge of assembling self-contained assemblies is to verify types for references and generate a consistent PE-file. Currently self-contained assemblies address CLI v1.1 features only. There are further size optimizations practicable. To reduce the size of the constant

pool, some kind of type descriptions can be shorten or eliminated. Description of custom types is not required by the CLR, except special names e.g. type constructor.

4.2. CLR implementation issues

The CLI describes possibilities for optimized CLR implementations. This section discusses these optimizations in terms of portability of self-contained assemblies among different CLR.

The CLR is responsible for resolving references to assemblies and loading types. References to external types are available in textual representation. A referenced type can have references to the same assembly or the external assemblies. The CLI suggests to resolve all references before start the execution or just-in-time compilation. Therefore all related assemblies must be loaded to create a consistent memory image.

For optimization issues the CLI introduced build in types e.g. `bool, char, object, string, ...`, which does not induce type references as long no type specific operation were performed. In contrast to Java the CLI provides an internal mapping of primitive type to their wrapper types. The CLR knows the mapping of primitive types to their wrapper types e.g. `object` ≡ `System.Object`. The mapping of primitive types to BCL types, inside the CLR, is realized with string compare, because type references are given in textual representation. For types implemented inside a self-contained assembly this mapping is possible further on. The CLI supports multiple ways to implement type methods [9] inside the BCL:*cil* means be implemented in CIL code.; *internalcall* flag indicates that the method is internal to the runtime and must be called in a special way.; *runtime*-method's implementation is provided by the runtime itself.; *pinvokeimpl*-method's have a unmanaged implementation and will be called through the platform invocation mechanism P/Invoke.

A *cil* implemented method can be executed by any CLR. An *internalcall* method is not portable among CLR implementations. This flag can occur in the BCL or in additional features provided by the CLR. An *runtime* supplied implementation is also CLR dependent. The *pinvokeimpl* flags indicates the CLR provided mechanism (P/Invoke) to call native code.

Figure 4 shows three different implementations of the System.Object::Equals(object) method.

The Microsoft .NET Framework uses the *internalcall* manner to perform the comparison. This implies the existence of an dispatch table for *internalcalls*.

Mono's provided implementation is based on CIL code, which makes the implementation portable.

In the Compact Framework BCL `System.Object::Equals(object)` is implemented

```
Microsoft .NET v1.1.4322
.method public hidebysig newslot virtual
    instance bool Equals(object obj) cil
    managed internalcall {}
```

```
Mono v1.1.13.2
.method public hidebysig newslot virtual
    instance bool Equals(object obj) cil
    managed
{
  .maxstack 8
  IL_0000: ldarg.0
  IL_0001: ldarg.1
  IL_0002: ceq
  IL_0004: ret
}
```

```
Compact Framework v1.0.500
.method public hidebysig newslot virtual
    instance bool  Equals(object obj) cil
    managed
{
  .maxstack  8
  IL_0000:  ldarg.0
  IL_0001:  ldarg.1
  IL_0002:  call      bool System.PInvoke.EE
      ::Object_Equals(object,object)
  IL_0007:  ret
}
.method public hidebysig static pinvokeimpl("
    mscoree" as "#17" winapi) bool
    Object_Equals(object obj1, object obj2)
    cil managed preservesig {}
```

Figure 4. Implementation of System.Object::Equals(object) in .NET, Mono and Compact Framework

with a additional call through the P/Invoke mechanism.

The current version of self-contained assemblies implementation is portable among different CLR as long as no implementation specifics are used. Whom can benefit from self-contained features as long as is executed with the CLR that provided the BCL implementation. To extend CLI technology to embedded devices an CLR implementation and a lightweight Kernel profile complaint BCL and a lightweight CLR must be provided.

5. Related Work

There are several approaches to optimize Java class files to meet the requirements of small embedded devices. The optimizations are often done on a per class basis.

Rayside et al. [14] propose a modified Java class file format with significant space reduction with little or no runtime penalty.

Clausen et al. [4] use macros for multiple occurrences of code fragments and a extended JVM with macro support.

The JamaicaVM [1] developed by aicas GmbH includes a builder tool for integrating Java bytecode and an corresponding Virtual Machine implementation into a single executable application binary. Bytecode is embedded as C-Array definition and linked with the JamaicaVM library.

TinyVM [15] is a firmware replacement for the Lego™ Mindstorm™ RCX hardware. The firmware can execute (interpret) Java programs that are compacted and considerable before deploy time.

The Lego.NET [13] project has developed a GCC front-end which translates CIL code into native machine code of the Lego™ Mindstorm™ RCX processor.

Microsoft's .NET Compact Framework is a subset of the .NET platform for mobile and embedded devices. The Compact Framework class libraries occupy at least 2 Megabyte of memory. The assembly format and execution environment differ only in trifles from the desktop version.

AppForge, Inc. offers with Crossfire [3] a product for multi-platform applications for mobile and wireless devices based on .NET. The CIL bytecode is transfered into a custom executable format that is executed by a platform specific Crossfire-Client software.

6. Conclusion and Future Work

In this paper the self-contained assembly approach is proposed to reduce memory consumption and startup time while executing CLI assemblies. CLI assemblies are loose coupled with other assemblies (shared class libraries, custom libraries). The creation of self-contained assemblies is done at type level with the customized PERWAPI assembly manipulation library. The compaction of assemblies bases on referenced types and requires no source code, nor compiler support. Self-contained assemblies are size optimized in terms of assembly footprint and memory consumption while execution. Furthermore the impact of an executed self-contained assembly is identical assuming that the CLR is CLI-complaint and no CIL-code is executed outside the assembly. The customized PERWAPI library allows adaptive compaction at type level, that means certain types can remain as references. It has to be investigated to what extent the abstraction of CLR internals from the BCL implementation could be realized in a CLI-compliant way. Self-contained assemblies could offer useful features for embedded systems development. The proposed assembly format could extend the availability of .NET to low-end embedded systems. CLR implementation issues have show that a lightweight Kernel profile based CLR must provided to target low-end embedded devices. Self-contained assemblies are an step forward to extend CLI to memory restricted embedded devices.

References

[1] aicas GmbH. JamaicaVM. Available at aicas.com, 2006.

[2] R. Anderson. The end of dll hell. Available at msdn.microsoft.com/library/en-us/dnsetup/html/dlldanger1.asp, 2000.

[3] AppForge, Inc. Crossfire homepage. Available at www.appforge.com/products/enterprise/crossfire, 2006.

[4] L. R. Clausen, U. P. Schultz, C. Consel, and G. Muller. Java bytecode compression for low-end embedded systems. *ACM Transactions on Programming Languages and Systems*, 22(3):471–489, 2000.

[5] R. Costa and E. Rohou. Comparing the size of .net applications with native code. In *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 99–104, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-161-9.

[6] Ecma International. Standard ECMA-335, Common Language Infrastructure (CLI). Available at www.ecma-international.org/publications/standards/Ecma-335.htm, 2002.

[7] J. Gough and D. Corney. PERWAPI- A PE File Reader/Writer. Available at www.plas.fit.qut.edu.au/perwapi, 2005.

[8] International Standards Organisation. Information technology – Common Language Infrastructure ISO/IEC 23271:2003(E) First edition, 2003.

[9] S. Lidin. *Inside Microsoft .NET IL Assembler.* Microsft Press, 2002.

[10] Microsoft Corporation. Shared source common language infrastructure. Available at msdn.microsoft.com/net/sscli, 2002.

[11] Microsoft Corporation. .NET Compact Framework. Available at msdn.microsoft.com/netframework/programming/netcf, 2005.

[12] Microsoft Corporation. .NET Framework. Available at msdn.microsoft.com/netframework, 2005.

[13] Operating Systems and Middleware Group. Lego.NET Website. Available at www.dcl.hpi.uni-potsdam.de/research/lego.NET, 2005.

[14] D. Rayside, E. Mamas, and E. Hons. Compact java binaries for embedded systems. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 9. IBM Press, 1999.

[15] J. H. Solorzano. TinyVM Homepage. [Online:] tinyvm.sf.net, 2006.

[16] The Mono Project. Homepage. Available at www.mono-project.com, 2006.