Technische Universität München
Institut für Informatik
Software & Systems Engineering
Prof. Dr. Dr. h.c. M. Broy

Istanbul Teknik Universitesi
Bilgisayar Mühendisligi Bölümü

Bachelorarbeit

Evaluierung und Vergleich der CLI-Implementierungen Microsoft .NET, SSCLI (Rotor) und Mono

Eingereicht von: *Georg Kalus*
kalus@in.tum.de

Aufgabensteller: *Prof. Dr. Dr. h.c. Manfred Broy*
Betreuer: *Dr. Feza Buzluca (ITÜ)*
Marco Kuhrmann (TUM)

Technische Universität München
Institut für Informatik
Software & Systems Engineering
Prof. Dr. Dr. h.c. M. Broy

Istanbul Teknik Universitesi
Bilgisayar Mühendisligi Bölümü

Bachelorarbeit

Evaluierung und Vergleich der CLI-Implementierungen Microsoft .NET, SSCLI (Rotor) und Mono

Eingereicht von: *Georg Kalus*
kalus@in.tum.de

Begonnen am: *15. November 2004*
Abgegeben am: *31. Januar 2005*

Aufgabensteller: *Prof. Dr. Dr. h.c. Manfred Broy*
Betreuer: *Dr. Feza Buzluca (ITÜ)*
Marco Kuhrmann (TUM)

Erklärung der Selbstständigkeit

Hiermit erkläre ich , dass

1. wörtlich aus anderer Quelle übernommene Teile der Arbeit deutlich als Zitat gekennzeichnet und mit Quellenangabe versehen sind,
2. sinngemäß aus anderer Quelle übernommene Teile der Arbeit mit Quellenangabe versehen sind,
3. alle übrigen Teile der Arbeit, insbesondere die zu entwickelnde Software, ausschließlich von mir ohne inhaltliche oder wörtliche Übernahme von Teilen anderer Werke erarbeitet wurden.

München, den 31. Januar 2005

Georg Kalus

Danksagung

Ganz besonders möchte ich meinen Betreuern Marco Kuhrmann und Dr. Feza Buzluca danken. Marco Kuhrmann möchte ich für das Thema selbst danken und für sein großes Engagement in der nicht einfachen Betreuungssituation über mehrere hundert Kilometer. Trotz dieser Umstände hat er mich stets in die richtige Richtung gelenkt. Dr. Feza Buzluca möchte ich besonders für seine prompte Bereitschaft danken, die Betreuung für diese Arbeit in Istanbul zu übernehmen.

Bedanken möchte ich mich auch bei meinem Vater Dr. Christian Kalus für das Aufspüren einiger sprachlicher und logischer Unstimmigkeiten. Der selbe Dank gilt meinem Bruder Simon Kalus, der mit geschultem Auge die gesamte Arbeit durchgegangen ist und zahlreiche sprachliche Fehler aufgedeckt hat.

Sehr dankbar bin ich auch meinen beiden WG-Mitbewohnern Peer und Robert für unsere abendlichen Tee-Runden und die gegenseitige Motivation. Diese Gespräche und ihre Freundschaft waren der Lohn für die oft anstrengenden Tage.

Zuletzt möchte ich mich bei Ender bedanken. Sie war ein willkommener und liebreizender Grund, die Arbeit auch mal Wochenende ruhen zu lassen.

Georg Kalus München, 31. Januar 2005

Zusammenfassung

Der ECMA CLI Standard hat in den letzten Jahren eine beträchtliche Populatität als Entwicklungsframework erlangt. Neben den Microsoft-Implementierungen hat sich insbesondere Mono als eine sehr ausgereifte Implementierung dieses Standards hervorgetan. In dieser Arbeit sollen die Implementierungen Microsoft .NET , SSCLI (Rotor) und Mono miteinander verglichen und gegeneinander positioniert werden. Dabei liegt der Schwerpunkt auf den Standardelementen und ihren Ausprägungen. Eigenheiten der Implementierungen – sofern nicht Teil des Standards – werden zwar erwähnt, sind aber nicht Hauptgegenstand dieser Arbeit. Die gewonnenen Erkenntnisse werden anhand einiger beispielhafter Anwendungen verifiziert.

Abstract

In the past years, the ECMA CLI standard has gained tremendous popularity as a development framework. Apart from the Microsoft-implementations, especially Mono has been successful as a very mature implementation of the standard. This thesis will compare and relate the three implementations .NET , SSCLI (Rotor) and Mono. The main focus will be the different realizations of standard elements. Non-standard elements unique to one implementation will be mentioned but are not the primary subject of this thesis. The results will be verified with a few exemplary applications.

Inhalt

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einleitung

Im Jahr 2000 hat Microsoft mit .NET eine Software-Umgebung vorgestellt, die nach Aussage des Unternehmens [WEB04f] „Informationen, Menschen, Systeme und Geräte“ verbindet. Hinter dieser etwas nebulösen Beschreibung verbirgt sich eine Softwareinfrastruktur, die sich in erster Linie an Softwareentwickler richtet. Diese Umgebung soll das Entwickeln – insbesondere von Web-basierten Anwendungen – vereinfachen. Microsoft adressiert mit dieser Technologie nicht nur aus der Vergangenheit bekannte Probleme in der Softwareentwicklung, wie zum Beispiel die Dll Hölle (siehe [DTG03]), es führt auch eine komplett neue, von Grund auf objektorientierte Klassenbibliothek – welche unter anderem die MFC ablöst – ein. Vor allem aber bietet diese Technologie ein gemeinsames Fundament für Komponenten, welche in den unterschiedlichsten Programmiersprachen geschrieben sein können. Das Herz dieser Infrastruktur bildet das sogenannte .NET Framework.

1.1 Das .NET Framework

Das .NET Framework bildet ein gemeinsames Bett für momentan über zwanzig Programmiersprachen. Es besteht aus der sogenannten CLI (engl. „Common Language Infrastructure“, manchmal auch CLR – „Common Language Runtime“.) und einer Reihe von Basisbibliotheken, der sogenannten BCL.

Unabhängig voneinander entstandene und aus beliebigen Quellen stammende Komponenten können im Rahmen der CLI sicher und transparent eine gemeinsame Anwendung bilden. Möglich wird das durch die sogenannte „Managed Execution“. Compiler erzeugen nicht mehr direkt Maschinencode sondern eine Zwischensprache, die sogenannte CIL („Common Intermediate Language“). Angereichert wird dieser Zwischencode durch sogenannte Metadaten. Das Format der Metadaten ist extrem flexibel und erweiterbar gehalten. Zusammen mit den Metadaten hat die Common Intermediate Language genügend Sprachmittel, um beinahe beliebige höhere Programmiersprachen repräsentieren zu können. Nicht mehr das Betriebsystem selbst, sondern die Common Language Infrastructure führt diesen Zwischencode aus. Sie erzeugt erst zur Laufzeit tatsächliche Maschinenbefehle. Die Ausführung in der Runtime bietet zahlreiche Vorteile. Die größten sind:

Portabilität Um eine Anwendung auf einer anderen Plattform auszuführen muß nicht mehr die Anwendung selbst, sondern nur die Laufzeitumgebung portiert werden. Die Anwendungen sind auf verschiedenen Plattformen – zumindest theoretisch – binärkompatibel. Das heißt, eine Anwendung kann auf einer Plattform entwickelt und kompiliert worden sein, und auf einer anderen Plattform ausgeführt werden, vorausgesetzt die Common Language Infrastructure existiert auch auf der Zielplattform.

Sicherheit Durch die Ausführung des Anwendungscodes in der Runtime und nicht im Betriebssystem direkt, wird die Sicherheit erhöht. Die Runtime hat jederzeit volle Kontrolle über die in ihr ausgeführten Anwendungen. Der Schaden, den nicht

oder falsch funktionierende Komponenten – zum Beispiel durch illegale Speicherzugriffe – verusachen können, wird durch diese Kontrolle und strikte Überwachung begrenzt. Da jede Komponente durch Zwischencode – angereichert mit Metadaten, also Typinformationen – repräsentiert wird, stehen der Laufzeitumgebung außerdem mehr Möglichkeiten zur Verfügung, Sicherheit tatsächlich zu gewährleisten, also etwa Typsicherheit konsequent einzufordern. Ebenso fällt zum Beispiel Reflection als Nebenprodukt mit ab. Bisher mußte dies durch Hilfsmittel und künstliche Erweiterungen, wie zum Beispiel die C++ RTTI („Runtime Type Information"), realisiert werden.

Dynamik Die Verschiebung der Erzeugung von tatsächlichem Maschinencode bis zum letztmöglichen Moment bietet zahlreiche Perspektiven zur Optimierung des erzeugten Maschinencodes: Zum einen müssen nur noch diejenigen Teile der Anwendung übersetzt werden, die auch tatsächlich benutzt werden. Darüberhinaus können Laufzeitinformationen, wie zum Beispiel Feldgrößen vom Laufzeitsystem unmittelbar zur Optimierung herangezogen werden.

Jede Komponente ist in der Laufzeitumgebung durch die Metadaten komplett selbstbeschreibend. Damit kann diese dem Anwendungsprogrammierer viele Aufgaben abnehmen. Solche Aufgaben sind zum Beispiel Garbage Collection, Codezugriffsbeschränkungen, Serialisierung von Typen usw.

Der größte Nachteil der Managed Execution in der Laufzeitumgebung ist ein potentieller Geschwindigkeitkeitsnachteil gegenüber herkömmlicher, statischer Kompilierung und der Ausführung direkt im Betriebsystem. Vorgänge wie Garbage Collection, Gewährleistung von Typsicherheit, Laufzeitkompilierung usw. sind alle zeitaufwendig.

1.2 Einführendes Beispiel

Um die Bedeutung und Fähigkeiten der CLI zu verdeutlichen, sollen die Möglichkeiten der Sprachinteroperabilität an einem kleinen, abstrakten Beispiel demonstriert werden.

Der in Listing 1.1 dargestellte Code realisiert eine einfache „Hello World" Anwendung in C++ .

```
// C++ Beispiel

# using <mscorlib.dll>

using namespace System;

void main()
{
   Console::WriteLine("Hallo␣Welt␣aus␣C++");
}
```

Listing 1.1: Hello World in C++

Die selbe Anwendung ist in Listing 1.2 in C# implementiert.

```
// C# Beispiel

using System;

class HelloWorldCSharp
```

```
{
   public static void Main()
   {
      Console.WriteLine("Hallo␣Welt␣aus␣C#");
   }
}
```

Listing 1.2: Hello World in C#

Das dritte Beispiel in Listing 1.3 zeigt die einfache „Hello World“ Anwendung in der Sprache Visual Basic .

```
'VB Beispiel

imports System

public Module ALL

   public sub main()

      Console.WriteLine("Hallo␣Welt␣aus␣VB")

   end sub

end Module
```

Listing 1.3: Hello World in Visual Basic

Auch wenn diese Beispiele sehr primitiv sind, lassen sich zwei interessante Beobachtungen machen:

- Alle Beispiele benutzen ein und die selbe Methode `System.Console.WriteLine()` aus den CLI Basisbibliotheken.
- Bis auf Feinheiten werden alle Beispiele nach dem Kompilieren zu dem selben Resultat in der CIL führen. Dieses Resultat wird erst zur Laufzeit vom Just-In-Time Compiler in tatsächliche Maschinenbefehle umgesetzt.

Eine einfache „Hello World“ Komponente in der CIL Sprache ist in Listing 1.4 dargestellt.

```
// IL Beispiel

.assembly extern mscorlib {}

.assembly BeispielAssembly{}

.class public HelloWorldIL
{

   .method public hidebysig static void sayhello() cil managed
   {
      .entrypoint

      ldstr "Hallo Welt aus IL"

      call void[mscorlib] [m1] System.Console::WriteLine(class System.String)

      ret
   }
```

```
   .method public hidebysig specialname rtspecialname instance  void .ctor()
       il managed
   {
      call instance void[mscorlib]System.Object::.ctor()

      ret
   }
}
```

Listing 1.4: Hello World in CIL

Diese Komponente kann nun wiederum aus einer beliebigen anderen CLI Programmiersprache instanziiert und verwendet werden. Listing 1.5 zeigt das für die C# Programmiersprache und Listing 1.6 für die Visual Basic Programmiersprache.

```
// Benutzung der IL Hello World Komponente aus C#

using System;

class CallILCode
{
   public static void Main()
   {
      try
      {
         HelloWorldIL.sayhello();
      }
      catch(Exception e)
      {
         Console.WriteLine(e);
      }
   }
}
```

Listing 1.5: Instanziierung aus C#

In Listing 1.5 wird die statische Methode `sayhello()` der Komponente `HelloWorldIL` völlig transparent benutzt, als sei diese Komponente ein C# Typ.

```
' Benutzung der IL Hello World Komponente aus Visual Basic

imports HelloWorldIL

public Module All
   public Class VbCode
      Inherits HelloWorldIL
   end Class

   public Sub Main()
      HelloWorldIL.sayhello()
   end Sub
end Module
```

Listing 1.6: Instanziierung aus Visual Basic

Auch in Listing 1.6 wird aus der Visual Basic Methode `Main()` die in CIL geschriebene Komponente `HelloWorldIL` benutzt, um die Bildschirmausgabe zu machen. Darüberhinaus wird hier auch noch von dieser Komponente abgeleitet.

Natürlich sind diese Mechanismen nicht nur für Komponenten, welche in der CIL Sprache geschrieben wurden, möglich. Die sprachenübergreifende Instanziierung, Benutzung und Ableitung ist zwischen allen auf die CLI abzielenden Sprachen möglich. Eine

in Visual Basic geschriebene Komponente kann von einer C# Komponente ableiten und in einer Delphi Komponente benutzt werden.

In Abbildung 1.1 sind die sprachübergreifenden Fähigkeiten und die Arbeitsweise der CLI noch einmal dargestellt.

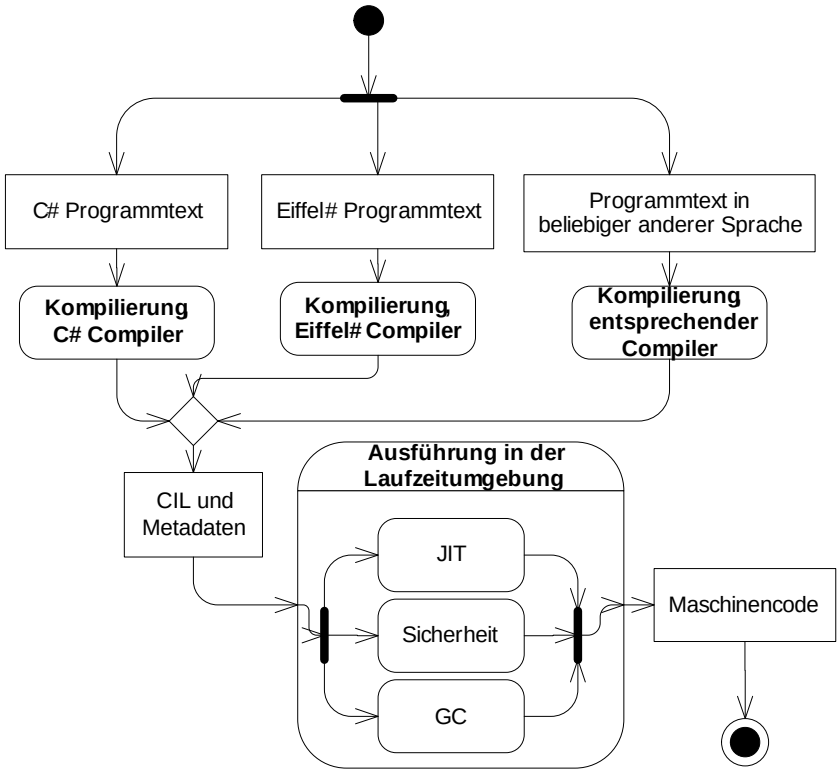

Abbildung 1.1: Arbeitsweise der CLI

1.3 Der ECMA CLI Standard

Im August 2000 hat Microsoft – mit mehreren Industriepartnern, u. a. Intel und Hewlett Packard – einen Großteil des .NET Frameworks zusammen mit der C# Sprachspezifikation bei der Standardisierungsorganisation ECMA eingereicht. Im Dezember 2001 wurden diese Standards von der ECMA bei der ISO eingereicht. Das Ergebnis dieses Prozesses sind der CLI-Standard ECMA-335 und der C# Sprachstandard ECMA-334, bzw. die Standards ISO/IEC 23270:2003 und 23271:2003. Die aktuelle Fassung der Standard-Dokumente ist bei der ECMA erhältlich [WEB04g].

Die ECMA

Die ECMA [WEB04g], oder „European Computer Manufacturer Association“ existiert seit 1961 mit Sitz in Genf. Sie hat bereits über 350 Standards verabschiedet. Sie ist sehr eng mit der ISO [WEB04i] – der „International Standards Organisation“ – verbunden.

Mit der Erhebung zum Standard hat Microsoft einerseits die alleinige Kontrolle über die CLI-Spezifikation aus der Hand gegeben und andererseits alternativen Implementierungen die Tür geöffnet.

1.4 Die Shared Source CLI

Um alternative Implementierungen zu fördern, eine funktionierende Implementierung des Standards zu präsentieren und vor allem den Portabilitätsbeweis zu erbringen hat Microsoft – in einem für das Unternehmen ungewöhnlichen Schritt – neben der kommerziellen .NET Implementierung eine kostenlose, im Quellcode verfügbare Implementierung des ECMA CLI-Standards veröffentlicht. Diese Version ist unter dem Namen SSCLI (engl. „Shared Source CLI“) und unter dem Microsoft-internen Codenamen „Rotor“ bekannt geworden und kann bei Microsoft [WEB02a] heruntergeladen werden. Während die kommerzielle .NET Version zahlreiche, nicht im Standard festgelegte Erweiterungen, wie zum Beispiel ADO.NET, ASP.NET und Windows.Forms enthält, umfasst die SSCLI nur den Standard. Große Teile der SSCLI sind direkt aus der kommerziellen Version übernommen. Gewisse Teile, wie zum Beispiel die Garbage Collection, sind aus politischen Gründen oder um das Verständnis zu erleichtern neu geschrieben worden. Andere Teile, wie beispielsweise die Laufzeitkompilierung, wurden aus technischen Gründen modifiziert bzw. neu geschrieben, um z. B. die Portierung auf andere Plattformen zu vereinfachen.

Die SSCLI ist zwar portabel und läuft zum Beispiel unter FreeBSD und MacOS X, die Lizenzbedingungen sind aber recht restriktiv. Die kostenlose Nutzung beschränkt sich im Wesentlichen auf Forschung und Lehre. Da sie von Microsoft auch ausdrücklich als Demonstrationsobjekt verstanden wird, kommt die SSCLI für den professionellen Einsatz nur bedingt in Frage. Diese Tatsache hat alternative Implementierungen des .NET Frameworks auf den Plan gerufen.

1.5 Mono

Erklärtes Ziel des Standardisierungsprozesses war und ist die Förderung alternativer, von Microsoft unabhängiger Implementierungen des .NET Frameworks.

Zwei Open Source Initativen haben sich an die Implementierung des CLI-Standards gewagt: Das DotGNU-Projekt mit Portable.NET und das Unternehmen Ximian mit dem Mono-Projekt.

DotGNU's Portable.NET

Die Portable.NET genannte Implementierung des DotGNU-Projektes [WEB04e] ist noch nicht besonders weit vorangeschritten. Sie verfügt noch nicht über einen vollständigen C# -Compiler, hat noch keinen Just-In-Time Compiler sondern bisher nur einen CIL-Interpreter und auch die im Standard spezifizierten Klassenbibliotheken sind bisher nur in Ansätzen implementiert. Zum 9.12.2004 war Portable.NET in der Version 0.6.10 verfügbar. Diese Implementierung wird daher bei den folgenden Betrachtungen dieser Arbeit nicht weiter berücksichtigt.

Die am weitesten ausgereifte alternative Implementierung des CLI-Standards kommt aus dem Hause Ximian und trägt den Namen Mono. Diese Firma wurde im Sommer 2004 von Novell aufgekauft. Das Mono-Projekt verfügt daher über wesentlich bessere Ausstattung, als das Portable.NET Projekt der DotGNU Initiative. Neben den zahlreichen Entwicklern der Open-Source-Gemeinde, arbeiten an dem Projekt etwa 5 festangestellte Entwickler. Insgesamt arbeiten an Mono momentan circa 130 größtenteils freiwillige Entwickler ([WEB04b]). Im Sommer 2004 wurde offiziell der Mono 1.0 Release

veröffentlicht. Neben dem reinen CLI-Standard enthält Mono auch noch Implementierungen der ASP.NET und der ADO.NET API. Darüberhinaus enthält es zahlreiche spezifischen Erweiterungen, die vor allem GUI-Anwendungen und Datenbankanbindung betreffen. Mono enthält eine rudimentäre Implementierung von Windows.Forms und Anbindungen an zahlreiche andere Gui-Toolkits, wie zum Beispiel Gtk und Qt.

2 Motivation

Mit knapp vier Jahren ist der CLI-Standard verhältnismäßig jung. Die Microsoft SSCLI wurde im Mai 2002 erstmals der Öffentlichkeit vorgestellt. Der erste offizielle Release von Mono ist noch kein halbes Jahr alt (er wurde im Sommer 2004 veröffentlicht). Dementsprechend dünn gesät sind bisher unabhängige Vergleiche dieser Implementierungen. Da auf dem CLI-Standard basierende Plattformen aber mit hoher Wahrscheinlichkeit eine gewichtige Stellung in der Softwarelandschaft von morgen haben werden, ist eine genauere Untersuchung der bisher existierenden Implementierungen dringend nötig.

Umso wichtiger wird ein Vergleich aufgrund des Aufkaufs des Mono-Herstellers Ximian durch die Firma Novell. Das Mono-Projekt hat durch diesen Schritt einen gewaltigen Schub erfahren. Edd Dumbill, Coauthor des Buches "Mono: A Developer's Notebook"([EN04]), prophezeit gar eine radikale Änderung der Linux-Entwicklungslandschaft ([WEB04r]):

> „Linux-Entwicklung wird in Zukunft sehr viel anders aussehen, wenn Mono seine Ziele erreicht."

Die Chancen dafür stehen nicht schlecht. Unter anderem setzt Novell Mono intern für die Entwicklung von Produkten wie iFolder und ZENworks ein und die Berliner Firma Völcker Informatik AG benutzt Mono, um eine große Zahl von Serveranwendungen für die Stadt München zu portieren ([WEB04q]). Damit wird Mono in Zukunft auf den etwa 350 Servern der Stadt München laufen.

Schon jetzt wird Mono zusammen mit Linux als Host für zahlreiche ASP.NET Webanwendungen benutzt. Beispiele dafür sind u. a. [WEB04l] und [WEB04t].

Wie in der Einleitung schon angedeutet und in Abbildung 2.1 veranschaulicht, bieten besonders .NET und Mono zahlreiche, nicht im Standard erfasste Erweiterungen und eigene Komponenten an.

Diese Komponenten – zum Beispiel die Realisierungen von ASP.NET – werden in dieser Arbeit nicht verglichen. Mit der vorliegenden Arbeit wird nur auf den Standard selbst eingegangen (Kapitel 3). Nach der Beschreibung des Standards werden die drei Implementierungen .NET , SSCLI (Rotor) und Mono vorgestellt, ihre Standardkonformität beleuchtet und ihre spezifischen Erweiterungen erfasst (Kapitel 4). Nach einer Positionierung der Implementierungen gegeneinander werden die gewonnen Erkenntnisse im Kapitel 5 anhand konkreter Fallbeispiele verifiziert. Abgeschlossen wird die Arbeit mit Kapitel 6 durch eine Bewertung und einen Ausblick.

Da der Rahmen dieser Arbeit sehr beschränkt ist, kann der Vergleich der Implementierungen nicht vollständig erfolgen. Vielmehr werden Teilaspekte der Implementierung herausgegriffen und näher untersucht.

Das Thema Performance wird z. B. überhaupt nicht berücksichtigt. Dieses Thema bietet sicher genug Stoff für eine eigene Arbeit. Einige Performance-Messungen und -Vergleiche der CLI-Laufzeitumgebungen finden sich in [WEB04s]. Auch die Fähigkeiten der Sprachinteroperabilität werden nicht behandelt. Ist ein Assembly übersetzt, beinhaltet es schließlich nur noch CIL-Code. Die verwendete höhere Quellsprache ist nicht mehr erkennbar.

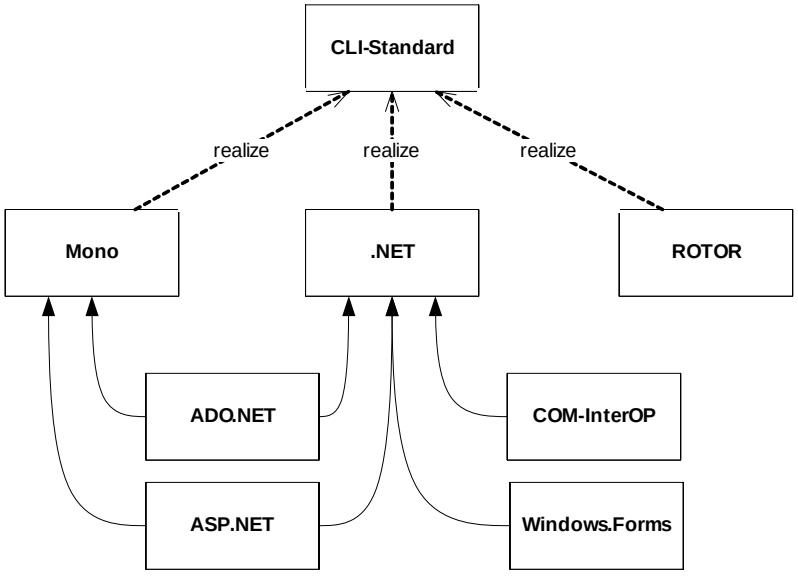

Abbildung 2.1: Der CLI Standard und seine Implementierungen

Eine solche Untersuchung wäre damit in erster Linie eine Untersuchung der CIL-Code erzeugenden Compiler.

Auch die praktische Untersuchung der verschiedenen Ausprägungen erfolgt nicht systematisch und umfassend, sondern stichprobenartig. Die mehr oder weniger willkürliche Wahl von Anwendungsbeispielen erlaubt aber die Gewinnung von für den Alltag relevanten und interessanten Erkenntnissen hinsichtlich der Verträglichkeit und Interoperabilität von .NET , der SSCLI und Mono.