Hochschule für Technik und Wirtschaft Dresden (FH)

Fachbereich Informatik / Mathematik

Studiengang Informatik

# DIPLOMARBEIT

**zur Erlangung des akademischen Grades**

**Diplominformatiker (FH)**

Thema: **Untersuchungen zur Einbettung von Sprachelementen der prozessorientierten Simulation in C# unter .NET**

eingereicht von: Torsten Wondrak

eingereicht am: 04.11.2005

Betreuer: Prof. Dr.-Ing Wilfried Nestler

Hochschule für Technik und Wirtschaft Dresden (FH)

Fachbereich Informatik / Mathematik

Friedrich-List-Platz 1, D-01069 Dresden

# Eidesstattliche Erklärung

Hiermit versichere ich, Torsten Wondrak, geboren am 13.09.1979 in Meissen, dass die vorliegende Diplomarbeit von mir eigenständig unter Verwendung der angegebenen Literatur und ohne Mithilfe anderer Personen angefertigt wurde. Direkt oder indirekt übernommene Gedanken Dritter sind als solche gekennzeichnet.

Dresden, 04.11.2005

Torsten Wondrak

# Danksagung

An dieser Stelle möchte ich jenen Personen Dank sagen, die mich fachlich und menschlich bei meiner Diplomarbeit unterstützt haben.

Ein großer Dank gebührt meinem Betreuer Herrn Prof. Dr.-Ing Wilfried Nestler, der sowohl mit seiner stetigen Bereitschaft mich zu unterstützen, als auch durch viele nützliche Ratschläge und Anregungen, mir sehr geholfen hat.

Für die aktive Unterstützung bei der Durchsicht meines Manuskriptes danke ich Herrn Manuel Tanner und Herrn Thomas Matzke.

Für die Ermöglichung meines Studiums und die interessierte Anteilnahme möchte ich meinen Eltern und Großeltern danken.

Allen anderen, die mir zur Seite gestanden haben und die sich hier nicht persönlich wieder finden, sei hiermit auch mein Dank ausgesprochen.

# Inhaltsverzeichnis

# Abkürzungsverzeichnis

| | |
|---|---|
| C# | CSharp, .NET Programmiersprache |
| CIL | Common Intermediate Language |
| CLR | Common Language Runtime |
| CLS | Common Language Specification |
| CTS | Common Type System |
| FCL | Framework Class Library |
| GC | Garbage Collector |
| GDI | Graphics Device Interface |
| IL | Intermediate Language |
| JIT | Just-in-Time |
| JVM | Java Virtual Machine |
| MSIL | Microsoft Intermediate Language |
| Opcode | Operationscode |
| VES | Virtual Execution System |

# Glossar

***. NET-Compiler*** – Complier einer .NET-Sprache erzeugen keinen prozessorspezifischen Maschinencode, sondern einen plattformunabhängigen Zwischencode.

***.NET Framework*** – Das .NET Framework ist eine Softwareentwicklungsplattform der Firma Microsoft. Es ist der Hauptbestandteil des .NET-Konzeptes.

***Common Intermediate Language*** – siehe Intermediate Language

***Common Language Runtime*** – Laufzeitumgebung des .NET Frameworks.

***Common Language Specification*** – Regelwerk, welches spezifiziert wie die Umsetzung der Intermediate Language erfolgen muss.

***Common Type System*** – Typkonzept des .NET Frameworks.

***Coroutine*** – Coroutinen sind Methoden, die die Kontrolle an eine beliebige andere Coroutine oder an das Hauptprogramm weitergeben können. Die Abarbeitung muss dabei nicht am Ende der Coroutine angelangt sein. Erhält die Coroutine die Kontrolle zurück wird an der Unterbrechungsstelle fortgesetzt. Die Werte aller lokalen Variablen bleiben über die Unterbrechung hinaus erhalten.

***Debugger*** – Tool zum Auffinden, Diagnostizieren und Eliminieren von Fehlern in Software.

***Delegate*** – ist ein Referenztyp, der dazu dient, eine Methode mit einer bestimmten Signatur und einem bestimmten Rückgabewert zu kapseln. Jede zu diesem Delegate passende Methode kann gekapselt werden. (In C++ sind diese Anforderungen Zeiger auf Funktionen). Delegates werden insbesondere bei der Ereignisbehandlung und bei asynchronen Methodenaufrufen verwendet.

***Framework Class Library*** – Einheitliche Klassenbibliothek für alle .NET-fähigen Programmiersprachen.

***Garbage Collector*** – Automatische Speicherverwaltung der Common Language Runtime. Der Garbage Collector arbeitet im Hintergrund und räumt den Speicher auf.

***Intermediate Language*** – Zwischencode der vom Compiler erzeugt wird. Code in IL ist verwalteter Code (Managed Code).

***Just-in-Time-Compiler*** – erzeugt aus Managed Code zur Lauftzeit Native Code (Maschinencode). Er ist Teil der Common Language Runtime.

***Managed Code*** – Programmcode, der im .NET Framework abläuft, wird als Managed Code bezeichnet.

***Maschinencode*** – wird erst zur Laufzeit aus dem IL-Code erzeugt. Maschinencode wird auch als nicht-verwalteter (unmanaged) Code oder Native Code bezeichnet.

***Metadaten*** – Informationen über Typen. Im .NET Framework sind Metadaten Pflichtbestandteil einer jeden Komponente.

***Prototyp*** – Ein Prototyp ist ein, für die jeweiligen Zwecke, voll funktionsfähiges Versuchsmodell eines geplanten Produktes. In der Softwareentwicklung werden drei Arten des Prototypings unterschieden: experimentelles, evolutionäres und exploratives Prototyping.

***Stack*** – Stapelspeicher oder Kellerspeicher. Ein Stapel kann eine beliebige Menge von Objekten aufnehmen und gibt diese entgegengesetzt zur Reihenfolge der Aufnahme wieder zurück (Last In-First Out-Prinzip, kurz LIFO).

***Unmanaged Code*** – nicht-verwalteter Code (Maschinencode)

***Verifizierung*** – oder Verifikation. In der Informatik und Softwaretechnik versteht man unter Verifizierung den mathematischen Beweis, dass ein Programm (also eine konkrete Implementierung) der vorgegebenen Spezifikation entspricht.

# 1 Einleitung

Die Computersimulation ist heutzutage ein bedeutendes Instrument zur Analyse komplexer Systeme. Sie findet Anwendung in den verschiedensten Fachgebieten – von den Natur- und Ingenieurwissenschaften über die Wirtschafts- und Sozialwissenschaften bis hin zur Medizin.

Der Kernpunkt einer Simulation sind stets Experimente am Modell, um Rückschlüsse auf das Verhalten des Originalsystems führen zu können.

Die Simulation kommt zum Einsatz, da Experimente mit dem Original aus verschiedenen Gründen[1] oft nicht möglich sind. Aber auch bestehende Prozesse können analysiert und durch Veränderung verschiedener Parameter optimiert werden.

Nach der mathematischen Struktur des verwendeten Modells wird zwischen kontinuierlicher und diskreter Simulation unterschieden. Bei der diskreten Simulation unterscheidet man weiterhin zwischen prozessorientierter und ereignisorientierter Simulation.

[1] Experimente mit dem Original dauern zu lange, sind zu teuer, sind physikalisch unmöglich, zerstören das Original, Original existiert noch gar nicht etc.

# 2 Die von Philosophie MODSIM III

MODSIM III ist eine modulare Programmiersprache, welche objektorientierte Programmierung und diskrete prozessorientierte Simulation ermöglicht. Die Syntax von MODSIM ähnelt stark der eines Modula-2 Programms.

Die objektorientierte Simulation in MODSIM hat folgende Eigenschaften:

- Objekte können mehrere zeitparallele Aktivitäten ausführen, die Simulationszeit verbrauchen
- Aktivitäten operieren autonom, können sich synchronisieren, können unterbrochen werden, konkurrieren um Ressourcen
- Eine Aktivität ist ein Ereignis, gesteuert durch eine Objektinstanz unter Nutzung einer TELL- oder WAITFOR-Methode
- Eine Instanz kann mehrere TELL- oder WAITFOR-Methoden ausführen, jede dieser Methoden auch mehrmals
- Aktivitäten können zum gleichen Modellzeitpunkt oder in der Zukunft aufgerufen werden
- Simulationszeit kann nur in TELL-Methoden verbraucht werden
- Objekte besitzen unterschiedliche Lebensdauer
- Objekte tauschen Nachrichten aus
- Prozessorientierte Simulation wird realisiert; vereinfacht große Modelle, weil gemeinsames Verhalten im Objekttyp und individuelle Daten in der jeweiligen Instanz enthalten sind
- Große Modelle werden logisch gemäß der Strukturierung des Originals unterteilt, vereinfacht die Modellentwicklung

Die TELL- bzw. WAITFOR-Methoden und die WAIT-Anweisungen stellen in MODSIM wichtige Mittel zur prozessorientierten Simulation dar. Durch sie werden die Aktivitäten eines Objektes beeinflusst und gesteuert.

Hier die wichtigsten Eigenschaften der TELL- und WAITFOR-Methoden:

- TELL-Aufrufe sind nur für TELL-Methoden zulässig
- TELL-Methoden müssen keine Simulationszeit verbrauchen, können aber in der Zukunft gestartet werden
- Jede WAIT-Anweisung in einer TELL- oder WAITFOR-Methode wird als eine Aktivität des Elternobjektes betrachtet
- Wenn eine WAIT-Anweisung betreten wird, dann suspendiert die TELL- oder WAITFOR-Methode ihre Ausführung
- Nach Ablauf der WAIT-Anweisung nimmt die suspendierte Methode die Ausführung wieder auf (Objekt hat Aktivität ausgeführt)
- ON INTERRUPT - Statements, dienen als Alternative falls WAIT unterbrochen wurde, die Codeausführung wird dann an dieser Stelle fortgesetzt
- TELL-Methoden sind nur mit IN-Parametern und nicht als Funktionsmethoden nutzbar
- WAITFOR-Methoden sind zusätzlich mit OUT und INOUT-Parametern auch als Funktionen formulierbar

Die WAIT-Anweisung ist in MODSIM III im allgemeinem wie folgt aufgebaut:

```
WAIT   reason
       statementsequence
       [ON INTERRUPT
       statementsequence]
END WAIT;
```

Hierbei gibt es zwei Arten – WAIT DURATION und WAIT FOR, wobei die erste Anweisung zum Warten über eine bestimmte Zeit dient, z.B. *WAIT DURATION 10.0* wartet 10 Zeiteinheiten. Bei WAIT FOR wird eine zweite Aktivität gestartet,

und die rufende Aktivität suspendiert ihre Ausführung. Nach Ende der gestarteten Aktivität nimmt die rufende Aktivität ihre Ausführung wieder auf.

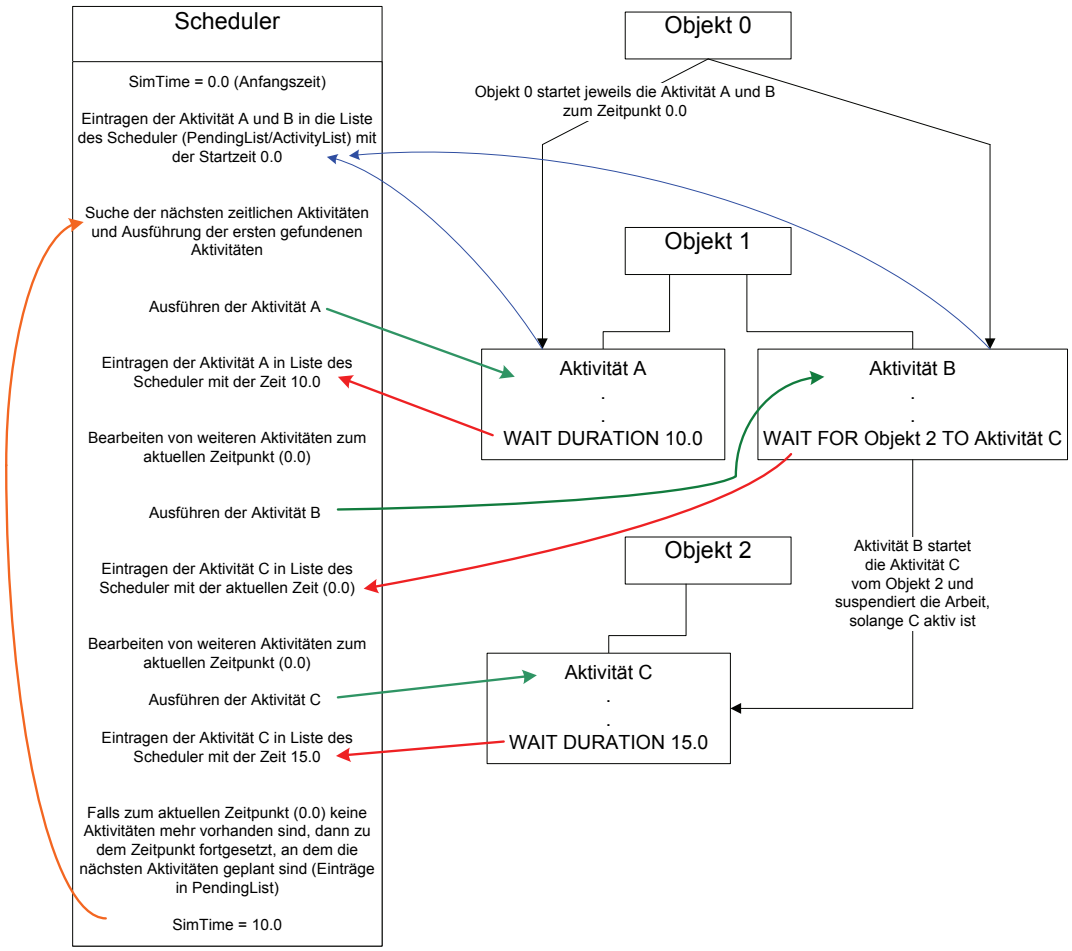


**Abbildung 2.1: Allgemeiner Ablauf der prozessorientierten Simulation in MODSIM III**

# 3 Aufgabenstellung und Motivation

Die Diplomarbeit hat zur Aufgabe einen Lösungsweg zu finden, um eine prozessorientierte Simulation im .NET Framework möglich zu machen.

Hierbei sollen folgende Aspekte beachtet werden:

- die Programmiersprache MODSIM III dient hierbei als Vorlage zur Umsetzung der notwendigen Sprachelemente
- es soll eine nicht-threadbasierte Umsetzung erfolgen, d.h. parallele Prozesse sollen nicht über Threads dargestellt werden

Ein weiterer Bestandteil der Diplomarbeit ist die prototypische Implementierung des gefundenen Lösungsweges.

Die Einbettung in die .NET-Umgebung bringt viele Vorteile mit sich. So stehen, neben den zu implementierenden Elementen, alle Möglichkeiten des .NET Framework, wie z.B. Datenbankanbindung und Webservices, zur Verfügung. Weiterhin könnte durch die Verwendung von Remoting verteilte Simulation betrieben werden. Durch das GDI+ von .NET ist die Anbindung einer grafischen Oberfläche kein Problem.

.

# 4 Lösungsansatz

Der wichtigste Aspekt ist die nicht-threadbasierte Umsetzung, welche durch die Verwendung von Coroutinen erfolgen kann. Coroutinen können die Kontrolle explizit an eine beliebige andere Coroutine oder an das Hauptprogramm weitergeben. Dabei muss aber die Abarbeitung der Coroutine nicht am Ende angelangt sein. Erhält die Coroutine die Kontrolle zurück, so wird die Codeausführung genau an der Unterbrechungsstelle fortgesetzt. Die Werte aller lokalen Variablen bleiben über die Unterbrechung hinaus erhalten. Dieses Konzept wird in dieser Art auch in MODSIM verwendet. Im .NET Framework gibt es (noch) keine Implementierungen von Coroutinen. Daher muss hier ein Lösungsansatz gefunden werden, um dieses Konzept umzusetzen.

Ein möglicher Ansatz wäre die Verwendung von Sprüngen mittels *goto*, um innerhalb einer Methode an die entsprechende Stelle zu gelangen. Allerdings besteht hierbei das Problem, dass mit Hilfe von *goto* nicht in Schleifen gesprungen werden kann. Außerdem wäre dabei die Lesbarkeit des Quellcodes eher schlecht als recht und die Implementierung läge hier ganz und gar beim Nutzer selbst.

Ein weiterer Ansatz wäre die Verwendung der Intermediate Language, denn hier sind Sprünge innerhalb des Codes ohne Probleme möglich. Jedoch soll nicht der Nutzer in IL programmieren, sondern nur in C# die notwendigen Implementierungen vornehmen.

Hieraus ergibt sich folgender Lösungsansatz: Dem Nutzer wird eine Bibliothek mit den Sprachelementen zur prozessorientierten Simulation zur Verfügung gestellt. Nach der erfolgreichen Kompilierung des vom Nutzer erstellten Quellcodes, wird das erstellte Assembly mit Hilfe der IL mit allen notwendigen Elementen erweitert.

# 5 Das .NET Framework

Das Herzstück der .NET-Plattform ist das .NET Framework. Es ist ein Modell zum Erstellen, Bereitstellen und Ausführen von Anwendungen.

Das .NET-Framework setzt auf dem Betriebssystem auf, das eine beliebige Windows[2]-Variante sein kann und besteht aus mehreren Komponenten. Gegenwärtig sind dies folgende:

- die vier offiziellen Sprachen C#, VB.NET, Managed C++ und JScript .NET
- die Common Language Runtime (CLR), eine objektorientierte Plattform für die Windows- und Web-Entwicklung, die von allen diesen Sprachen genutzt wird
- mehrere zusammenhängende Klassenbibliotheken, die kollektiv als Framework Class Library (FCL) bezeichnet werden [B3]

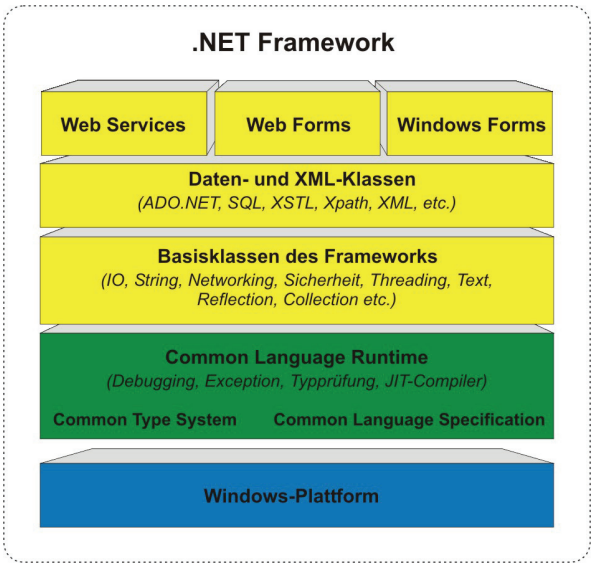


**Abbildung 5.1: .NET Framework Architektur [B3]**

[2] Wegen der Architektur der CLR könnte das Betriebssystem auch eine Unix-Variante oder irgendein anderes Betriebssystem sein

Die wesentlichen Bausteine der .NET-Technologie bilden die Common Language Runtime, die Spezifikation der Datentypen (Common Type System, kurz CTS) und eine Reihe von Definitionen für Sprachenintegration (Common Language Specification, kurz CLS).

Microsoft hat für die Portabilität von Anwendungen eine prozessorunabhängige Zwischensprache – Microsoft Intermediate Language (MSIL) – entwickelt. Diese ist von der zugrunde liegenden Architektur unabhängig. Der Quellcode wird mittels sprachspezifischer Compiler in die Zwischensprache übersetzt und anschließend unter Aufsicht der CLR verarbeitet und ausgeführt. Auf die MSIL wird näher im 6. Kapitel eingegangen.

## 5.1 Common Language Runtime

Die wichtigste Komponente des .NET Frameworks ist die CLR: Sie stellt die Umgebung zur Verfügung, in der die Programme ausgeführt werden. [B3]

Die CLR ist somit die Laufzeitumgebung des .NET Frameworks und ist vergleichbar mit der virtuellen Maschine von Java (JVM).

Sie besteht aus zwei Hauptbestandteilen – der Laufzeitausführungsengine (mscoree.dll) und der Basisklassenbibliothek. Die Laufzeitengine hat mehrere Aufgaben. Zunächst sucht und aktiviert sie die Objekte, indem Assemblies aufgelöst und Metadaten gelesen werden. Dann werden die Objekte auf ihre Sicherheit überprüft, im Speicher angeordnet und ausgeführt. Bereinigt werden die Objekte mittels Garbage Collection. Kapitel 6.5.1 gibt nähere Informationen zur Ausführungsumgebung von .NET.

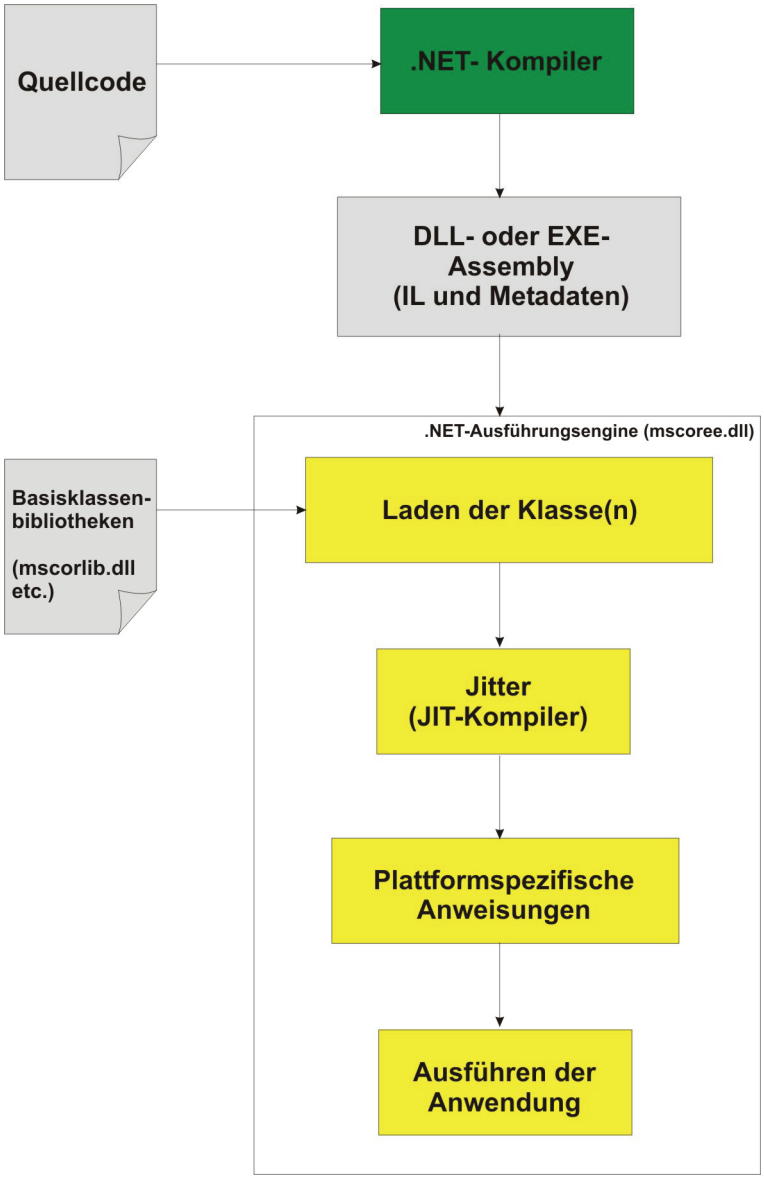


**Abbildung 5.2: Funktionsweise der .NET-Ausführungsengine [B4]**

## 5.2 Common Type System

Das Common Type System ist ein allgemeines Typensystem, das die Regeln für die Vorgehensweise der CLR beim Deklarieren, Verwenden und Verwalten von Typen vorgibt. Das CTS gewährleistet somit eine sprachenübergreifende Integration und die Typensicherheit. Das bedeutet, dass es möglich ist quer durch mehrere Programmiersprachen von Klassen zu erben, Exceptions

abzufangen und Polymorphismus zu nutzen. Nach dem CTS müssen sich alle .NET-Komponenten richten. So ist z.B. in .NET alles und jedes ein Objekt einer bestimmten Klasse, die von der Wurzelklasse *System.Object* abgeleitet ist. Das CTS unterstützt das allgemeine Konzept von Klassen, Interfaces, Delegates, Referenztypen und Werttypen.

## 5.3 Common Language Specification

Bei der Common Language Specification handelt es sich um eine Reihe von Richtlinien. Sie beschreiben detailliert die minimale und vollständige Menge von Funktionen, die ein .NET-fähiger Compiler für die Erzeugung von Code unterstützen muss. Die CLS gewährleistet somit die Verwendung des .NET Frameworks und der CLR. Weiterhin ist die Interaktion mit Komponenten, die in anderen .NET-Sprachen entwickelt wurden, gesichert.

# 6 Microsoft Intermediate Language

Dieses Kapitel durchleuchtet die Intermediate Language näher und gibt eine Übersicht vorhandener Tools zur Arbeit mit Microsofts Zwischensprache. Weiterhin wird näher auf den, vom Compiler erzeugten Code eingegangen und ein Einstieg in die Programmierung mit der Intermediate Language gegeben.

In der Literatur findet man auch die Bezeichnung Common Intermediate Language (CIL). Ob Microsoft Intermediate Language (MSIL), CIL oder nur kurz Intermediate Language (IL), all das meint das Gleiche.

Die IL ist die gemeinsame Basis aller Compiler für .NET-Sprachen und die einzige Sprache, die von der CLR direkt verarbeitet werden kann.

## 6.1 Assemblies – Die .NET-Binärdateien

Wie schon erwähnt erzeugen .NET-Compiler keinen Maschinencode, sondern es werden Assemblies erzeugt, die von der CLR verarbeitet werden.

Ein Assembly besteht aus einem oder mehreren Modulen. Das Manifest ist Bestandteil der Metadaten und beschreibt die Identität von Assembly und Modul sowie deren Interaktion. Es enthält Informationen über Identität, Inhalt, Abhängigkeiten, Zugriffsrechte und spezifische Eigenschaften des Assembly. Bei einem Assembly, das aus mehreren Modulen besteht, besitzt jedes Modul sein eigenes Manifest. Das ausgezeichnete primäre Modul enthält das Manifest über das komplette Assembly. Jedes Assembly enthält genau eines dieser primären Module.

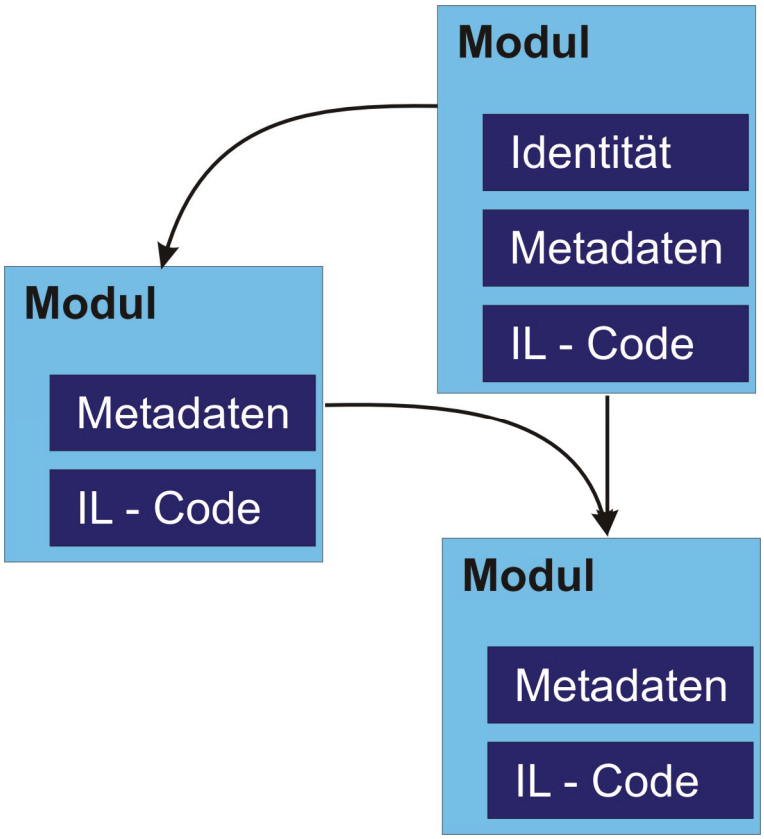


**Abbildung 6.1: Innerhalb einer Assembly trägt nur ein Modul - das so genannte primäre Modul - die Identität der Assembly.**

Die Metadaten werden von der CLR benötigt, um Verweise aufzulösen, eigene Typen zu exponieren und eine Versionskontrolle durchzuführen.

Betrachtet man die Arten von Assemblies, kann neben einer Unterscheidung in ausführbare Assemblies und Bibliotheken auf einer anderen logischen Ebene zwischen statischen und dynamischen Assemblies unterschieden werden. Dies bedeutet, dass sie sowohl statisch zum Kompilierungszeitpunkt als auch dynamisch während der Laufzeit generiert werden können. [M4]

Neben den Metadaten enthalten die Assemblies den auszuführenden Code in Form von IL-Code.

## 6.2 ILDASM – Der Intermediate Language Disassembler

Mit dem Intermediate Language Disassembler-Dienstprogramm ist es möglich jedes .NET-Assembly (EXE oder DLL) zu laden und dessen Inhalt (einschließlich Manifest, IL-Befehlssatz und Metadaten) zu untersuchen. Hierbei gibt es zwei Möglichkeiten, einmal mit Hilfe der grafischen Benutzeroberfläche (Abbildung 6.2) oder aber auch über die Kommandozeile. Wird die *ildasm.exe* ganz ohne Parameter gestartet, zeigt sich die grafische Benutzeroberfläche.

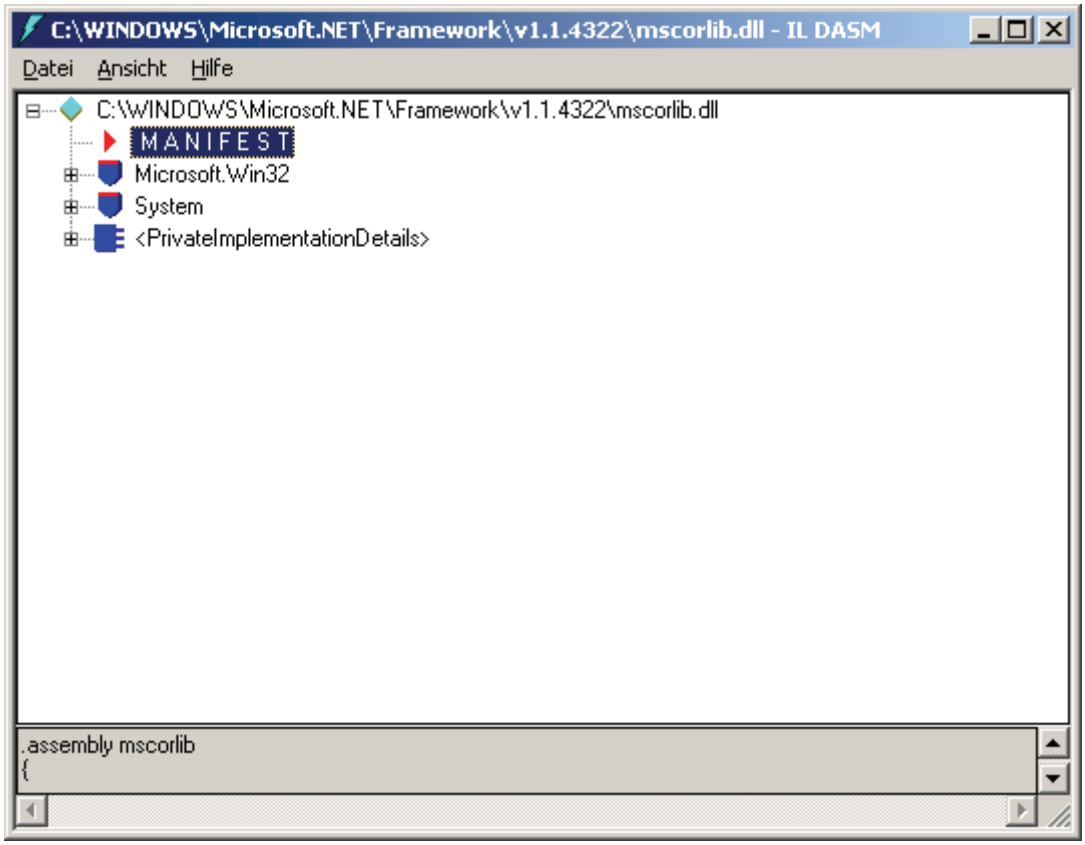


**Abbildung 6.2: Die grafische Benutzeroberfläche des Tools ildasm.exe**

Um den Inhalt eines Assembly mit Hilfe des Disassemblers in einer Datei abzulegen bietet sich folgender Kommandozeilen-Aufruf an:

```
ildasm /out:beispiel.il beispiel.exe
```

Hierbei wird das Assembly *beispiel.exe* vom Disassembler verarbeitet und das Ergebnis in einer IL-Datei abgespeichert.

## 6.3 ILASM – Der Intermediate Language Assembler

Das Pendant zum Disassembler ist das Kommandozeilen-Tool *ilasm.exe*. Dieses Tool generiert aus IL-Code ein Assembly inklusive aller notwendigen Metadaten.

Der Aufruf sieht wie folgt aus:

```
ilasm /exe beispiel.il /out:beispiel.exe /res:beispiel.res
```

Die Kommandozeilenoptionen des IL-Assembler und Disassembler sind im Anhang F zu finden.

## 6.4 Grundlegender Aufbau eines Assembly

### 6.4.1 „Hello World“ in IL

Die folgenden zwei Listings zeigen das „Hello-World“-Programm in C# und im dazugehörigen IL-Code.

**Listing 6.1: „Hello-World“ in C#**

```
using System;

namespace HelloWorld
{
   class Class1
   {
            private static string hellostring;

      [STAThread]
      static void Main(string[] args)
      {
         hellostring="Hallo Welt!";
         Console.WriteLine(hellostring);
      }
   }
}
```

**Listing 6.2: „Hello World“ im IL-Code (ausschnittsweise)**

```
//  Microsoft (R) .NET Framework IL Disassembler.  Version 1.1.4322.573
//  Copyright (C) Microsoft Corporation 1998-2002. All rights reserved.

.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                         // .z\V.4..
  .ver 1:0:3300:0
}
.assembly HelloWorld
{
  .custom instance void
   [mscorlib]System.Reflection.AssemblyCopyrightAttribute::.ctor(string)
   = ( 01 00 00 00 00 )
  ...
  .hash algorithm 0x00008004
  .ver 1:0:1901:16390
}
.module HelloWorld.exe
...
// ============== CLASS STRUCTURE DECLARATION ==================
.namespace HelloWorld
{
  .class private auto ansi beforefieldinit Class1
         extends [mscorlib]System.Object
  {
  } // end of class Class1

} // end of namespace HelloWorld

// =============== GLOBAL FIELDS AND METHODS ===================
// =============== CLASS MEMBERS DECLARATION ===================
.namespace HelloWorld
{
  .class private auto ansi beforefieldinit Class1
         extends [mscorlib]System.Object
  {
    .field private static string hellostring
    .method private hidebysig static void
            Main(string[] args) cil managed
    {
      .entrypoint
      .custom instance void [mscorlib]System.STAThreadAttribute::.ctor()
       = ( 01 00 00 00 )
      // Codegröße       11 (0xb)
      .maxstack  1
      IL_0000:  ldstr      "Hallo Welt!"
      IL_0005:  stsfld     string HelloWorld.Class1::hellostring
      IL_000a:  ldsfld     string HelloWorld.Class1::hellostring
      IL_000f:  call       void [mscorlib]System.Console::WriteLine(string)
      IL_0014:  ret
    } // end of method Class1::Main

    .method public hidebysig specialname rtspecialname
            instance void  .ctor() cil managed
    {
      // Codegröße       7 (0x7)
      .maxstack  1
      IL_0000:  ldarg.0
      IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
      IL_0006:  ret
    } // end of method Class1::.ctor
  } // end of class Class1
// =============================================================
} // end of namespace HelloWorld
```

In den folgenden Abschnitten wird anhand des „Hello World“-Beispiels näher auf die einzelnen Elemente eines Assemblies eingegangen.

### 6.4.2 Programm-Header

Der Programm-Header besteht aus mehreren Elementen. Zunächst das Metadaten-Item der Assembly-Referenz:

```
.assembly extern mscorlib
```

Hier wird ein Verweis auf eine externe .NET-Assembly definiert. Im „Hello World“-Beispiels ist das lediglich die mscorlib.dll. Die *AssemblyRef*-Deklaration enthält zusätzliche Informationen wie z.B. die Versionsnummer und einen öffentlichen Schlüssel.

Das nächste Element identifiziert das aktuelle Assembly und enthält eine Versionsnummer, einen Hash-Wert und beschreibende Attributwerte.

```
.assembly HelloWorld
```

Diese Identifikation auf das Assembly selbst zeigt, dass es sich um ein eigenständiges Assembly handelt. Andernfalls wäre es ein Modul, welches zu einem anderen Assembly bzw. Anwendung gehören würde.

Das folgende Element definiert das aktuelle Modul.

```
.module HelloWorld.exe
```

Jedes Modul trägt solch eine eigene Kennzeichnung, die immer den vollständigen Namen inklusive Dateiendung, angibt.

### 6.4.3 Namensräume und Klassen

Anschließend folgt eine Definition des verwendeten Namensraum (Namespace) und eine Vorwärtsdeklaration der verwendeten Klassen.

Die *Namespace*-Deklaration ist einfach eine *Container*-Deklaration, wie in den meisten Hochsprachen und verwendet (wie Sprachen im C-Stil) geschweifte Klammern, um den Gültigkeitsbereich und folglich die Member dieses Namespace zu definieren.

```
.namespace HelloWorld
{
} // end of namespace HelloWorld
```

Die *.class*-Direktive zur Deklaration einer Klasse ist da etwas komplexer.

```
.class private auto ansi beforefieldinit Class1
         extends [mscorlib]System.Object
 {
} // end of class Class1
```

Die angegebenen Schlüsselwörter haben folgende Bedeutung:

- *private* – Dieses Schlüsselwort bezeichnet den Zugriffsmodifikator und hat die gleiche Bedeutung wie in den meisten Hochsprachen: Der Typ ist nur innerhalb des Assemblies sichtbar.
- *auto* – Bestimmt den Layout-Stil der Klasse. Hier wird der *Class-Loader* angewiesen, das Speicherlayout automatisch anzulegen. Die möglichen Alternativen sind *sequential* und *explicit*.
- *ansi* – Definiert die Art der String-Konvertierungen, wenn mit unverwaltetem Code gearbeitet wird. Weitere mögliche Werte sind *unicode* und *autochar*.

- *beforefieldinit* – Dieses Schlüsselwort spezifiziert, dass der Aufruf einer statischen Methode den Typ nicht initialisiert. Es sagt der CLR, dass kein Konstruktor des Typs vor dem Aufrufen einer statischen Methode aufzurufen ist.
- *Class1* – Der Name der Klasse.
- *extends* – Das Schlüsselwort für Vererbung. Hierauf folgt eine Liste von Klassen und Schnittstellen, die dieser Typ erweitert.
- *[mscorlib]System.Object* – Dies ist der Basistyp, den der aktuelle Typ erweitert (Vererbung). Es ist natürlich auch möglich einen anderen Typ zu erweitern, aber jeder .NET-kompatible Typ erbt von diesem Basistyp.

Eine komplette Übersicht aller Attribute zur Klassendeklaration befindet sich in der IL Grammatikreferenz im Anhang D.

### 6.4.4 Felder

Felder werden in IL mit der Direktive *.field* deklariert. Im „Hello World“-Beispiel gibt es nur eine Feld-Definition für eine private statische Variable *hellostring* vom Typ *string*.

```
.field private static string hellostring
```

Der Anhang D bietet eine Übersicht zu den Attributen, die bei der Deklaration von Feldern verwendet werden können.

### 6.4.5 Methoden

Die *.method*-Direktive zeigt die Definition einer Methode. Im Beispiel gibt es nur eine Methode, die private statische Methode *Main*, welche keinen Rückgabewert liefert und als Parameter ein Array vom Typ *string* besitzt.

```
.method private hidebysig static void
            Main(string[] args) cil managed
```

Das Schlüsselwort *hidebysig* („hide by signature“) besagt, dass die Methode alle Methoden gleichen Namens und gleicher Signatur von Klassen, die in der Objekthierarchie im Vererbungsbaum weiter oben zu finden sind, versteckt.

Die Implementierungs-Flags *cil* und *managed* zeigen an, dass die Methode CIL-Code enthält und das dieser von der Laufzeitumgebung verwaltet werden kann (kein unsicherer Code).

Die Tabelle 6.1 listet die Schlüsselworte, die die Eigenschaften von Methoden beschreiben, auf.

An die Signatur der Methode schließt sich der Rumpf der Methode an. Dieser ist genau wie bei C# in geschweifte Klammern eingefasst. Sie definieren den Gültigkeitsbereich der Methode.

Im Rumpf der Methode des Beispiels finden sich drei weitere Direktiven – *.entrypoint*, *.custom* und *.maxstack*.

```
.entrypoint
.custom instance void [mscorlib]System.STAThreadAttribute::.ctor()...
.maxstack  1
```

Die Direktive *.entrypoint* weist den Compiler an, diese Methode als Einsprungspunkt des Moduls zu markieren. Beim Start des Moduls wird diese

Methode zuerst aufgerufen. Die Anwendung terminiert, wenn diese Methode zurückkehrt.

**Tabelle 6.1: Schlüsselworte, die die Eigenschaften von Methoden beschreiben [O1]**

| Schlüsselwort | Beschreibung |
|---|---|
| abstract | Abstrakte Klasse - entspricht abstract in C# |
| assembly | Entspricht internal in C# |
| famandassem | Die Methode kann nur von abgeleiteten Klassen innerhalb des Assemblies aufgerufen werden -keine Entsprechung in C# |
| Family | Entspricht protected in C# |
| famorassem | Entspricht protected internal in C# |
| Final | Entspricht sealed in C# |
| hidebysig | Die Methode verdeckt gleichnamige Methoden aus Basisklassen nur dann, wenn auch die Signatur übereinstimmt. Ist dieses Flag nicht angegeben, reicht der Name zum Verdecken. |
| memberaccessmask | Legt fest, ob Reflection auf unsichtbare Elemente möglich ist |
| newslot | Die Methode belegt grundsätzlich einen neuen Slot in der vtable |
| pinvokeimpl | Die Methode ist extern und wird via Pinvoke aufgerufen |
| private | Entspricht dem Schlüsselwort private in C# |
| privatescope | Die Methode kann nicht referenziert werden (bezieht sich auf den &-Operator in C++) |
| public | Wie public in C# |
| reuseslot | Die Methode kann einen existierenden Slot in der vtable verwenden |
| rtspecialname | Zeigt an, dass die Common Language Runtime die Namensauflösung übernimmt |
| specialname | Spezielle Methode; in diesem Fall zeigt der Name an, in welcher Weise die Methode speziell ist; .ctor steht zum Beispiel für Konstruktor |
| static | entspricht dem Schlüsselwort static in C# |
| unmanagedexport | Die Methode wird über einen "thunk" von unmanaged Code aufgerufen |
| virtual | Entspricht dem Schlüsselwort virtual in C# |
| vtablelayoutmask | Erlaubt es, Eigenschaften der Vtable zu spezifizieren |

Zum Hinzufügen von benutzerdefinierten Metadaten (Attributinformationen) dient die Direktive *.custom*. In diesem Fall handelt es sich um eine Instanz eines *STAThreadAttribute* mit einem leeren Konstruktoraufruf ( *.ctor()* ).

Mit *.maxstack* bestimmt der Compiler die Größe des Auswertungs-Stack, der für die Ausführung der Methode erforderlich ist. Die Größe wird in abstrakten Elementen und nicht in Bytes gezählt, weil dieser Wert zur Laufzeit nicht benötigt wird, sondern nur dazu da ist, die Verifizierbarkeit für den Compiler von CIL zu systemeigenem Code bereitzustellen. [M1]

Im Anschluss kommt der „eigentliche“ Code des Programms, welcher hier aus folgenden Zeilen besteht:

```
IL_0000:  ldstr      "Hallo Welt!"
IL_0005:  stsfld     string RoundTripWithILASM.Class1::hellostring
IL_000a:  ldsfld     string RoundTripWithILASM.Class1::hellostring
IL_000f:  call       void [mscorlib]System.Console::WriteLine(string)
IL_0014:  ret
```

Beim Aufruf von *ldstr* wird der angegebene String auf den Stack geladen. Anschließend wird dieser Wert mit *stdfld* vom Stack geholt und im angegeben statischen Feld gespeichert. Mit *ldsfld* wird der Wert des statischen Feldes auf den Stack gelegt. Die *call*-Instruktion ruft die nicht virtuelle Methode *WriteLine* der Klasse *Console* auf, die in der Assembly *mscorlib* zu finden ist. Hier wird beim Aufruf der aktuelle String-Wert vom Stack übergeben. [M1]

Der Befehl *ret* zeigt an, dass das Ende der Methode erreicht ist. Hierbei wird der aktuell auf dem Stack befindliche Wert zurückgegeben, wenn die Methode einen Rückgabewert hat.

Auf diese und weitere Befehle wird im Kapitel 6.5, Programmieren mit IL, näher eingegangen.

Eine Übersicht zu allen Attributen und Flags zur Definition von Methoden befindet sich im Anhang D.

### 6.4.6 Datentypen in IL

In den Namensräumen der .NET-Klassenbibliothek befinden sich mehr als 7000 Typen. Eine bestimmte Gruppe dieser Typen sind die so genannten primitiven Datentypen. Tabelle 6.2 listet diese, inklusive einer Beschreibung und der korrespondierenden Elemente in IL, auf.

**Tabelle 6.2: Primitive Datentypen von .NET - Enthalten im Namensraum System in der Assembly mscorlib [M5]**

| CTS-Typ | IL-Notation | Beschreibung |
|---|---|---|
| Void | void | Nichts, leer |
| Boolean | bool | Ein einzelnes Byte, Werte: true = 1, false = 0 |
| Char | char | Ein aus 2 Byte bestehender unsigned int repräsentiert ein Unicode-Zeichen |
| SByte | int8 | 1 Byte langer Integerwert, entspricht char in C/C++ |
| Byte | unsigned int8 | 1 Byte langer Integerwert, ohne Vorzeichen |
| Int16 | int16 | 2 Byte langer Integerwert |
| UInt16 | unsigned int16 | 2 Byte langer Integerwert, ohne Vorzeichen |
| Int32 | int32 | 4 Byte langer Integerwert |
| UInt32 | unsigned int32 | 4 Byte langer Integerwert, ohne Vorzeichen |
| Int64 | int64 | 8 Byte langer Integerwert |
| UInt64 | unsigned int 64 | 8 Byte langer Integerwert, ohne Vorzeichen |
| Single | float32 | 4 Byte lange Fließkommazahl |
| Double | float64 | 8 Byte lange Fließkommazahl |
| TypedReference | typedref | Eine Referenz inklusive entsprechender Informationen über den referenzierten Typ |
| IntPtr | native int | Ein Zeiger, dessen Größe von der jeweiligen Plattform abhängig ist |
| UIntPtr | native unsigned int | Siehe IntPtr |

Eine weitere Untergruppe dieser primitiven Typen ist der Datenzeiger. Hierbei gibt es zwei Typen – den Zeiger auf einen Managed Typ (managed pointer) und

den Zeiger auf einen Unmanaged Typ (unmanaged pointer). Tabelle 6.3 zeigt die jeweilige Notation.

Die Repräsentation von Klassen ist eine weitere Gruppe von Datentypen. Tabelle 6.4 gibt einen Überblick.

**Tabelle 6.3: Datenzeiger von .NET in IL [M5]**

| Name | IL-Notation | Beschreibung |
|---|---|---|
| PTR | <type>* | Unmanaged pointer, Zeiger auf Typ <type> |
| BYREF | <type>& | Managed pointer, Referenz auf <type> |

**Tabelle 6.4: Repräsentation von Klassen [M5]**

| CTS-Typ | IL-Notation | Beschreibung |
|---|---|---|
| [viele] | valuetype <class_ref> | Wertetyp |
| [viele] | class <class_ref> | Klasse oder Interface |
| String | String | [mscorlib]System.String |
| Object | Object | [mscorlib]System.Object |

## 6.5 Programmieren in IL

### 6.5.1 Virtual Execution System

Das Virtual Execution System (VES) setzt das CTS durch und führt verwalteten Code aus. Prinzipiell sind verwaltete Daten nichts weiter als Daten, die über die automatische Garbage Collection geladen und entladen werden. Verwalteter Code ist der Code, der auf diese Daten zugreifen kann. [M1]

#### 6.5.1.1 Funktionsweise des VES

Das VES muss in der Lage sein, verwalteten Code auf vielen möglichen Plattformen auszuführen. Deshalb ist es unmöglich, das VES wie eine Ausführungsumgebung aufzubauen, die für eine konkrete Prozessorarchitektur, beispielsweise die Intel-Architektur, ausgelegt ist. Es sind somit keine Annahmen über verfügbare Register, Cache oder Operationen auf dem Zielsystem vorhanden. Alle diese Betrachtungen werden in eine, für alle Plattformen gemeinsame Form oder in ein theoretisches Format abstrahiert, das sich für die konkrete Zielplattform durch den JIT-Compiler kompilieren lässt, wenn der Code ausgeführt wird.

#### 6.5.1.2 Methodenzustand

Das Ausführungsmodul führt alle Steuerungs-Threads der Anwendung mit verwaltetem Code und die Speicherverwaltung in einem gemeinsam genutzten Speicheraum aus. Die automatische Speicherverwaltung kümmert sich um den verwalteten Heap, auf dem jede aufgerufene Methode ihre Informationen ablegt. Das VES erzeugt bei jedem Methodenaufruf einen neuen Speicherblock für den Methodenzustand. Wenn der letzte Methodenzustand (für den Einstiegspunkt einer Anwendung) freigegeben wird, terminiert die Anwendung.

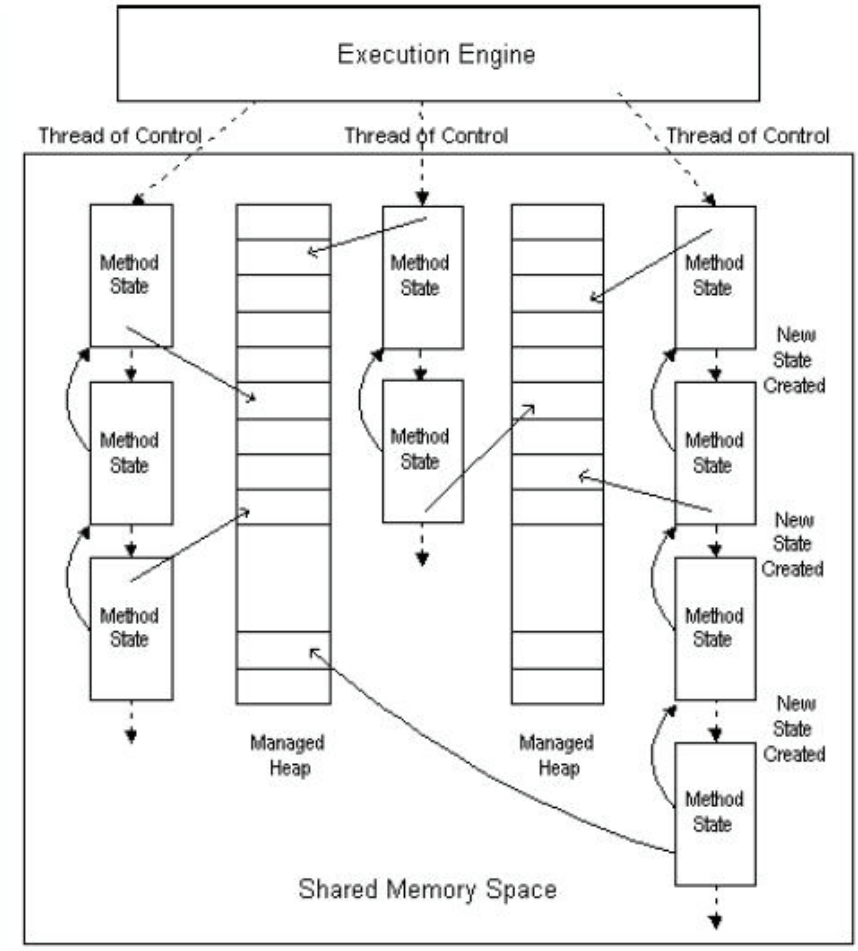


**Abbildung 6.3: Zustandsmodell: Steuerungs-Threads legen Methodenzustände auf dem verwalteten Heap ab [M1]**

Ist eine Methode fertig abgearbeitet, übergibt das Ausführungsmodul die Steuerung an den Methodenzustand zurück, der die abgearbeitete Methode aufgerufen hat. [M1]

Ein Methodenzustand besteht immer aus mehreren Teilen:

- Befehlszeiger (Instruction Pointer) – zeigt auf die nächste auszuführende Anweisung
- Methodenbeschreibung (Method Info Handle) – speichert geschützte Informationen über die Methodensignatur
- Auswertungs-Stack (Evaluation Stack)

- Eingabeparameter (Incoming Arguments) – die beim Aufruf dieser Methode zu verwendenden Argumente
- Lokale Variablen – ein nullbasiertes Array aller lokalen Objekte
- Lokale Zuordnungen (Local Allocations) – für die dynamische Zuordnung von lokalen Objekten
- Sicherheitsbeschreibung (Security Description)
- Rückgabezustand (Return State Handle) – stellt den Methodenzustand auf den Zustand vom Aufrufer wieder her (auch als dynamische Verknüpfung bezeichnet)

#### 6.5.1.3 Auswertungs-Stack

Jeder Methodenzustand besitzt einen zugehörigen Auswertungs-Stack, auf den die meisten Anweisungen zurückgreifen, um Argumente für Aufrufe abzuholen und Ergebnisse von Aufrufen abzulegen. Der Auswertungs-Stack besteht nur aus Objekten. Es spielt keine Rolle, ob eine Ganzzahl, ein String oder ein benutzerdefiniertes Objekt auf den Stack gelegt wird, die virtuelle Umgebung registriert nur, wie viele Einträge auf dem Stack gespeichert sind und in welcher Reihenfolge sie auf dem Stack liegen. Der Auswertungs-Stack darf an jedem möglichen Austrittspunkt einer Methode nur den Rückgabewert enthalten (oder keinen Wert, wenn Methode *void* zurückgibt).

### 6.5.2 Lokale Variablen

Lokale Variablen werden mit der *.locals init* - Anweisung am Anfang der Methode deklariert. Die Angabe von mehreren lokalen Variablen erfolgt in einer durch Komma getrennten und in Klammern geschlossenen Liste.

```
.locals init (int32 var1, string var2)
```

In der IL ist es nicht zwingend erforderlich, dass die Variablen einen Namen haben müssen, da die Variablen nach ihrem Index adressiert werden. Weiterhin wären also noch folgende Schreibweisen möglich:

```
.locals init ([0] int32 var1, [1] string var2)
```

```
.locals init (int32, string)
```

Der ILASM-Compiler übersetzt die Variablen stets in Indizes, so dass nach dem erneuten Disassembling folgender Ausdruck im IL-Code zu finden ist:

```
.locals init (int32 V_0, string V_1)
```

Die *ldloc* und *stloc* - Anweisungen ermöglichen die Arbeit mit den lokalen Variablen. *ldloc* lädt den Wert der angegebenen Variable auf den Stack. *stloc* speichert den obersten Wert, der auf dem Stack liegt in der angegebenen Variable ab.

```
.locals init ([0] int32 var1, [1] int32 var2)
```

```
ldloc 0
```

```
stloc 1
```

Hier wird der Wert der Variable *var1* der Variable *var2* zugewiesen.

Anstatt der Verwendung der Indizes kann natürlich auch der Name der Variablen verwendet werden, also *ldloc var1* bzw. *stloc var2*.

Bei der Verwendung des Index der Variablen gibt es verschiedene Varianten. Werden lokale Variablen mit einer Indexposition zwischen 0 und 3 benutzt, so kann man hier die Kurzschreibweise *ldloc.x* bzw. *stloc.x* (wobei x = 0,1,2 oder 3) verwenden. Liegt der Index zwischen 4 und 255, so findet die „short“ Variante von *ldloc* bzw. *stloc* Anwendung.

```
ldloc.s 5
```

```
stloc.s 6
```

Um einer Variablen einen bestimmten Wert zuzuweisen, kann dieser mittels entsprechender Opcodes auf den Stack geladen und dann mit *stloc* in der Variablen gespeichert werden.

Mit Hilfe des Opcodes *ldc* (kurz für „load constant“) können spezielle Werte auf den Stack geladen werden. Der Typ des Wertes wird dabei, durch einen Punkt getrennt, mit *ldc* verwendet. Für einen Integer-Wert (*int32*) wäre das z.B. i4. Anschließend folgt der konkrete Wert, wobei hier es auch wieder eine Kurzschreibweise für Werte zwischen -1 und 8 gibt.

```
ldc.r4 20 //lädt den Integer-Wert 20 auf den Stack
ldc.r4.2  //lädt den Integer-Wert 2 auf den Stack
```

Zur Konvertierung eines Typs in einen anderen dient die *conv*-Anweisung. Die Verwendung ähnelt der *ldc*-Anweisung, wobei bei *conv* der Typ angegeben wird, in den der auf dem Stack befindliche Wert konvertiert werden soll.

```
conv.i4//Konvertiert den Wert in einen Int32-Wert
```

Eine Übersicht zu den Opcodes gibt die IL Instruktionen-Referenz im Anhang E.

### 6.5.3 Methodenargumente

Die *ldarg*-Anweisung dient zur Arbeit mit Methodenargumenten. Die Verwendung dieses Opcode ähnelt *ldloc*. Es gibt auch hier die Kurzschreibweise *ldarg.x* sowie die „short“ Variante *ldarg.s*.

```
.method private hidebysig instance void
            StoreAValue(int32 ValueToStore) cil managed
{
      .maxstack  1
      .locals init (int32 aLocalInt)
      ldarg.1
      stloc.0
}
```

Hier wird der Wert des Argumentes der Methode *StoreAValue*, der lokalen Variable *aLocalInt* zugewiesen.

Dabei ist zu beachten, dass die Art der Methode (*static* oder *instance*) einen Einfluss auf den Index von *ldarg* hat. Der erste Index, sprich *ldarg.0* ist immer die Referenz des aktuellen Objektes.

Eine Übersicht zu den Opcodes zeigt der Anhang E.

### 6.5.4 Felder

Felder werden mit der *.field*-Direktive innerhalb einer Klasse deklariert. Der Zugriff auf Felder erfolgt mittels der Opcodes *ldfld* und *stfld* bzw. *ldsfld* und *stsfld*, wobei die letzten beiden zum Laden bzw. Speichern von statischen Feldern verwendet werden. Der Zugriff auf Felder erfordert auch das Laden der aktuellen Referenz mittels *ldarg.0*.

```
.class public beforefieldinit Class1
      extends [mscorlib]System.Object
{
   .field private int32 field1;
   .field private static int32 field2;
   .method public hidebysig instance
     void UseFields() cil managed
      {
     .maxstack 2
     ldarg.0
     ldfld int32 ILTest.Class1::field1
     stsfld int32 ILTest.Class1::field2
     ret
      }
}
```

### 6.5.5 Properties

Properties werden in IL auf zwei Methoden und eine Eigenschaft abgebildet. Listing 6.3 zeigt die Darstellung eines Property in IL.

**Listing 6.3: Property in IL**

```
.class private auto ansi beforefieldinit Class1
        extends [mscorlib]System.Object
  {
    .field private int32 x
    .method public hidebysig specialname
            instance int32  get_X() cil managed
    {
      ...
    } // end of method Class1::get_X

    .method public hidebysig specialname
            instance void  set_X(int32 'value') cil managed
    {
      ...
    } // end of method Class1::set_X



    .property instance int32 X()
    {
      .get instance int32 ConsoleApplication1.Class1::get_X()
      .set instance void ConsoleApplication1.Class1::set_X(int32)
    } // end of property Class1::X
  }
```

### 6.5.6 Operatoren

#### 6.5.6.1 Arithmetische Operatoren

Die arithmetischen Operatoren dienen zur Addition, Subtraktion, Multiplikation und Division zweier, sich auf dem Stack befindlichen Werte. Die Opcodes der Operatoren sind *add*, *sub*, *mul* und *div*. Diesen muss ein Laden der zu verarbeitenden Werte auf den Stack voran gehen.

Das folgende Beispiel zeigt die Addition zweier Methodenargumente.

```
.method public hidebysig instance void
            Add(int32 x, int32 y) cil managed
{
      .maxstack  2
      .locals init ([0] int32 z)
      ldarg.1
      ldarg.2
      add
      stloc.0
      ret
}
```

Weitere arithmetische Operatoren sind *neg* für die Negation eines Wertes und *rem* um den Restbetrag einer Division zu erhalten.

#### 6.5.6.2 Logische Operatoren

Für logische Operationen stehen die Opcodes *and*, *or*, *not* und *xor* zur Verfügung. Um Bitverschiebungen ausführen zu können gibt es einmal für die Linksverschiebung den Opcode *shl* und für die Rechtsverschiebung *shr*.

#### 6.5.6.3 Vergleichsoperatoren

Der Vergleich zweier Werte erfolgt mittels *ceq*. Hierbei lädt *ceq* den Wert 1 auf den Stack, wenn die Werte gleich sind und 0 wenn nicht.

Die Opcodes *cgt* und *clt* vergleichen immer den obersten Wert, der auf dem Stack liegt mit dem nächst tieferen. Bei *cgt* erfolgt eine Prüfung, ob der tiefere Wert auf dem Stack größer ist als der oberste (bei *clt* entsprechend kleiner als).

### 6.5.7 Arbeiten mit Objekten

Der Opcode *newobj* dient zum Erzeugen einer Instanz eines Typs. Das folgende Listing zeigt das Anlegen einer Instanz einer Klasse.

**Listing 6.4: Erzeugen einer Instanz einer Klasse**

```
.namespace ConsoleApplication1
{
  .class private auto ansi beforefieldinit Class2
         extends [mscorlib]System.Object
  {
    .method public hidebysig specialname rtspecialname
           instance void  .ctor() cil managed
    {
      .maxstack  1
      ldarg.0
      call       instance void [mscorlib]System.Object::.ctor()
      ret
    }

  }

  .class private auto ansi beforefieldinit Class1
         extends [mscorlib]System.Object
  {
    .method private hidebysig static void
           Main(string[] args) cil managed
    {
      .entrypoint
      ...
      .maxstack  1
      .locals init ([0] class ConsoleApplication1.Class2 cl2)
      // Erzeugen einer Instanz der Klasse Class2
      newobj     instance void ConsoleApplication1.Class2::.ctor()
      stloc.0
      call       string [mscorlib]System.Console::ReadLine()
      pop
      ret
    }
    ...
  }
}
```

Beim Erzeugen der Instanz wird der Konstruktor der Klasse aufgerufen, in diesem Fall der Default-Konstruktor (*.ctor()*). Eine genauere Beschreibung der Methodenaufrufe erfolgt im nächsten Kapitel. Der *newobj*-Aufruf bewirkt auch, dass ein Verweis der erzeugten Instanz auf den Stack gelegt wird und somit weiterverarbeitet werden kann. Im Fall des Beispiels wird das Objekt in einer lokalen Variable abgespeichert.

### 6.5.8 Methodenaufrufe

In IL können Methoden direkt oder indirekt aufgerufen werden. Die Signatur einer Methode selbst enthält Informationen darüber, ob sie eine statische oder eine Instanzmethode ist. Deshalb muss beim Aufruf diesbezüglich keine Unterscheidung getroffen werden. Methodensignaturen enthalten jedoch keine Informationen darüber, ob die Methode virtuell ist. Deshalb ist hierbei eine Unterscheidung beim Aufruf zwingend notwendig. [M6]

Direkte Methodenaufrufe

Bei den direkten Methodenaufrufen wird zunächst einmal unterschieden zwischen Sprüngen (*jump*) und einfachen Aufrufen (*call*). Bei einem Sprung wird die aufrufende Methode abgebrochen und nach Ausführen der aufgerufenen Methode kehrt der Ablauf auch nicht mehr hierhin zurück. Anders ist das Verhalten bei einem einfachen Aufruf per *call*, bei dem nach Ausführung ein Rücksprung zur aufrufenden Methode erfolgt. Beim Aufruf einer virtuellen Methode ist *callvirt* zu verwenden. [M6]

```
call    instance void NameSpace.MainClass::CallMe1()

callvirt    instance void NameSpace.MainClass::CallMe2()
```

Indirekte Methodenaufrufe

In IL können Methoden indirekt aufgerufen werden. Hierbei erfolgt der Aufruf per Funktionszeiger (auf den Stack). Dies ermöglicht ein dynamisches Verhalten, bei dem Methoden, genauer deren Funktionszeiger, beispielsweise als Rückgabewert aus einer anderen Methode übergeben werden können. *ldftn* und *ldvirtftn* laden jeweils den Funktionszeiger für nichtvirtuelle und virtuelle Methoden. [M6]

```
ldftn instance void NameSpace.MainClass::CallMe1()

ldvirtftn instance void NameSpace.MainClass::CallMe2()
```

Indirekte Methodenaufrufe werden z.B. bei der Verwendung von Delegates benutzt.

Neben den erwähnten Möglichkeiten gibt es jedoch noch einen weiteren Weg, Methoden aufzurufen. Dieser wird auch als Tail Call bezeichnet. Dabei wird das zu verwendende Schlüsselwort *tail.* als Präfix vor Aufrufe der Art *call* oder *callvirt* gesetzt.

```
tail. call void NameSpace.MainClass::CallMe()
```

Die Argumente für eine Tail-Call-Methode müssen explizit auf den Stack geladen werden. Im Gegensatz zu einem *jump* ist es nicht erforderlich, dass die Signatur der aufgerufenen Methode exakt derjenigen der aufrufenden entspricht. Lediglich die Rückgabewerte müssen in beiden Fällen kompatibel sein.

### 6.5.9 Flusssteuerung

Zur grundlegenden Flusssteuerung zählen Verzweigungs- bzw. Bedingungsanweisungen und Schleifen. Mit Bedingungsanweisungen, welche den *if/else*-Kombinationen in den meisten Hochsprachen ähnlich sind, lässt sich Code abhängig vom Wert eines Elementes ausführen. Diese Kombinationen enthalten in IL zwei Elemente, Marken und Verzweigungsoperatoren. Die Marken dienen als Sprungziele, d.h. das Programm wird an der Stelle fortgesetzt, die durch die Marke angegeben wird. Hier kann auch ein relativer Wert verwendet werden, der die Sprungdistanz (Offset) spezifiziert. Das bedeutet, wie viele Bytes an dazwischen liegendem Bytecode von der aktuellen Position ausgehend nach vorn oder nach hinten im Programm zu überspringen sind. Werden Marken verwendet, so berechnet der ILASM-Compiler die für den Sprung benötigten Offsets. Das Listing 6.5 zeigt eine *if/else*-Kombination in IL.

**Listing 6.5: if/else-Kombination in IL**

```
IL_0005:  ldloc.0       // Laden der lokalen 1. Variable
IL_0006:  ldc.i4.s   20   // Wert 20 auf den Stack laden
IL_0008:  bge.s      IL_0016 // bge.s holt sich die zwei obersten Werte
                             // vom Stack, und springt zur Marke IL_0016 //
                             wenn das tiefer liegende Element größer //
                             gleich dem obersten ist

IL_000a:  ldstr      "kleiner"
IL_000f:  call       void [mscorlib]System.Console::WriteLine(string)
IL_0014:  br.s       IL_0020 // Sprung zur Marke IL_0020


IL_0016:  ldstr      "groesser"
IL_001b:  call       void [mscorlib]System.Console::WriteLine(string)
IL_0020:  br.s       IL_0030
```

Alle Verzweigungsanweisungen (*branch*) beginnen mit einem b, gefolgt von einer Abkürzung für die Art des jeweiligen Sprungs. Im Beispiel wird die Kurzform mit einem *.s* verwendet, d.h. hier ist das Sprungziel nicht weit entfernt. Das Offset bei der Kurzversion darf nicht größer als ein Byte sein, bei der Langversion stehen vier Byte zur Verfügung.

Alle *branch*-Anweisungen arbeiten in der gleichen Weise: es werden die obersten Werte vom Stack entnommen (wenn etwas verglichen wird), dann wird das Ergebnis des jeweiligen Bedingungstests bestimmt und anschließend wird mit der Codeausführung an der spezifizierten Marke fortgesetzt, wenn der Ausdruck *true* ergibt, andernfalls mit der nächsten Zeile. Die Tabelle 6.5 zeigt eine Liste aller Verzweigungsoperanden.

Schleifen werden in IL ebenfalls mit Hilfe der Verzweigungsoperanden aufgebaut. Listing 6.6 zeigt ein Beispiel für eine Schleife in IL, welche mehrmals den String „looping“ ausgibt. Hierbei wird eine lokale Variable (*ldloc.1*) als Schleifenzähler verwendet. Diese wird vor dem Ende der Schleife immer um eins erhöht (Zeile *IL_002c* – *IL_002f*). Anschließend wird der Wert dieser Variable mit dem Wert 10 (*ldc.i4.10*) verglichen. Ist der Schleifenzähler kleiner als 10, so wird zu der Marke *IL_0022* gesprungen und die Schleife wird erneut ausgeführt.

**Listing 6.6: Eine Schleife in IL**

```
IL_0020:  br.s       IL_0030

IL_0022:  ldstr      "looping"
IL_0027:  call       void [mscorlib]System.Console::WriteLine(string)
IL_002c:  ldloc.1
IL_002d:  ldc.i4.1
IL_002e:  add
IL_002f:  stloc.1
IL_0030:  ldloc.1
IL_0031:  ldc.i4.10
IL_0032:  blt.s      IL_0022
```

**Tabelle 6.5: IL-Verzweigungsoperanden**

| Anweisung | Offset-größe | Vom Stack entfernte Elemente | Beschreibung |
|---|---|---|---|
| br | int32 | - | Verzweige um <int32> Bytes von der aktuellen Position aus |
| br.s | int8 | - | |
| brfalse<br>brnull<br>brzero | int32 | int32 | Verzweige, wenn das oberste Stack-Element gleich 0 ist |
| brfalse.s | int8 | int32 | |
| brtrue<br>brinst | int32 | int32 | Verzweige, wenn das oberste Stack-Element ungleich null (oder eine gültige Adresse einer Objektinstanz) ist |
| brtrue.s<br>brinst.s | int8 | int32 | |
| beq | int32 | *,* | Verzweige, wenn beide Werte gleich sind |
| bge | int32 | *,* | Verzweige, wenn der erste Wert größer oder gleich dem zweiten Wert ist |
| bgt | int32 | *,* | Verzweige, wenn der erste Wert größer oder als der zweite Wert ist |
| ble | int32 | *,* | Verzweige, wenn der erste Wert kleiner oder gleich dem zweiten Wert ist |
| blt | int32 | *,* | Verzweige, wenn der erste Wert kleiner als der zweite Wert ist |
| bne.un | int32 | *,* | Verzweige, wenn die beiden Werte nicht gleich sind (vorzeichenloser Vergleich) |
| bge.un | int32 | *,* | Verzweige, wenn der erste Wert größer oder gleich dem zweiten Wert ist (vorzeichenloser Vergleich) |
| bgt.un | int32 | *,* | Verzweige, wenn der erste Wert größer oder als der zweite Wert ist (vorzeichenloser Vergleich) |
| ble.un | int32 | *,* | Verzweige, wenn der erste Wert kleiner oder gleich dem zweiten Wert ist (vorzeichenloser Vergleich) |
| blt.un | int32 | *,* | Verzweige, wenn der erste Wert kleiner als der zweite Wert ist (vorzeichenloser Vergleich) |
| Short-Varianten | int8 | *,* | |

### 6.5.10 Überwachte Blöcke

Ein überwachter Block beginnt mit der *.try*-Direktive und wird von geschweiften Klammern eingegrenzt. Jeder überwachte Block muss am Ende über einen speziellen Opcode verfügen, der das Verlassen des Blocks spezifiziert – die *leave*-Anweisung. Diese Anweisung bewirkt, dass das Programm den überwachten Block verlässt, ohne eine Ausnahme auszulösen und an der markierten Position fortfährt, wenn kein Fehler aufgetreten ist. Bei einem Fehler setzt die Codeauswertung mit der Zeile fort, die die *catch*-Anweisung enthält. Über diese Anweisung geschieht auch die so genannte Ausnahmefilterung, d.h. es wird spezifiziert welche Art von Ausnahme abgefangen wird. In Listing 6.7 ist der Ausnahmetyp *System.Exception*, die Basisklasse aller Ausnahmen, d.h. hier werden Ausnahmen aller Art abgefangen.

**Listing 6.7: Ein überwachter Block in IL**

```
TryAgain:
IL_004a:  ldstr      "Geben Sie eine Zahl ein: "
IL_004f:  call       void [mscorlib]System.Console::WriteLine(string)
.try
{
        IL_0054:  call       string [mscorlib]System.Console::ReadLine()
        IL_0059:  call       float64 [mscorlib]System.Double::Parse(string)
        IL_005e:  stloc.1
        IL_005f:  leave.s    NoException
}  // end .try
catch [mscorlib]System.Object
{
        IL_0061:  pop
        IL_0062:  ldstr    "Falsches Format, nochmal! "
        IL_0067:  call     void [mscorlib]System.Console::WriteLine(string)
        IL_006c:  leave.s  TryAgain
}

NoException: …
```

### 6.5.11 Debuggen der IL

Das .NET Framework stellt zum Debuggen der IL zwei Tools zur Verfügung – ein Kommandozeilen-Tool (*cordbg.exe*) und ein Tool mit grafischer Oberfläche, den Microsoft CLR-Debugger (*dbgclr.exe*, Abbildung 6.4 ).

Eine wichtige Voraussetzung um den IL-Code debuggen zu können, ist das Erstellen einer so genannten Programmierdatenbank, welche dazu dient die Zusammenhänge zwischen ausführbarem Code und eigentlichem Quellcode festzustellen. Eine solche Datenbank wird durch die Debug-Option des ILASM-Compilers erstellt (pdb-Datei).

```
ilasm /debug test.il /out:test.exe
```

Im Folgenden wird kurz die Funktionsweise des CLR-Debugger beschrieben. Nach dem Starten des Programms muss die zu debuggende IL-Datei geöffnet werden (Datei → Öffnen → Datei, oder STRG+O). Anschließend wird das zu debuggende Programm festgelegt (Debuggen → Zu debuggendes Programm). Abbildung 6.5 zeigt den dazugehörigen Dialog.

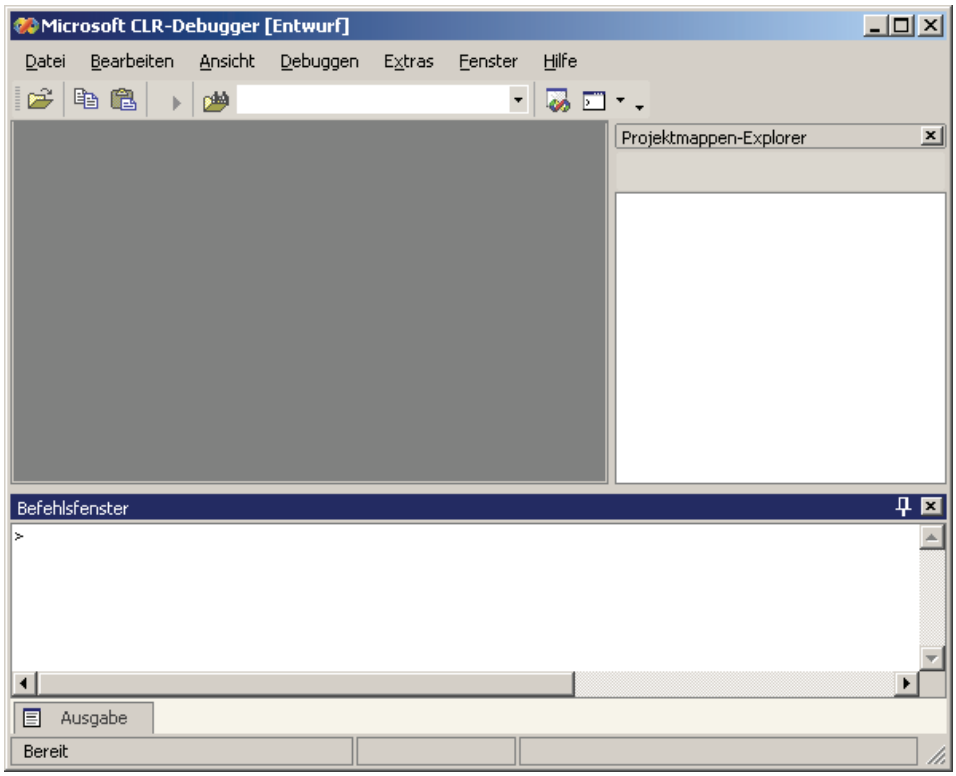


**Abbildung 6.4: Der Microsoft CLR-Debugger**

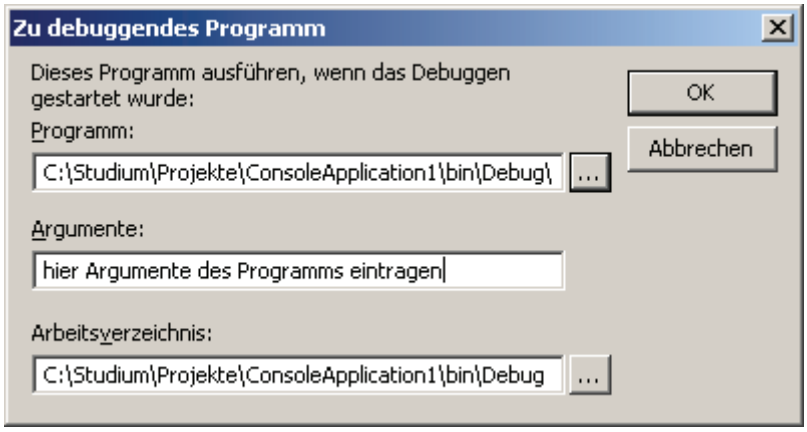


**Abbildung 6.5: Auswahl des zu debuggendem Programms**

Nach dem die Haltepunkte gesetzt wurden, kann das Debuggen gestartet werden (Debuggen → Start, oder F5). Es können natürlich während dieses Vorgangs Variablen überwacht sowie eine Vielzahl von Informationen (z.B. Ausgabe und Aufrufliste) angezeigt werden. Abbildung 6.6 zeigt den CLR-Debugger in Aktion.

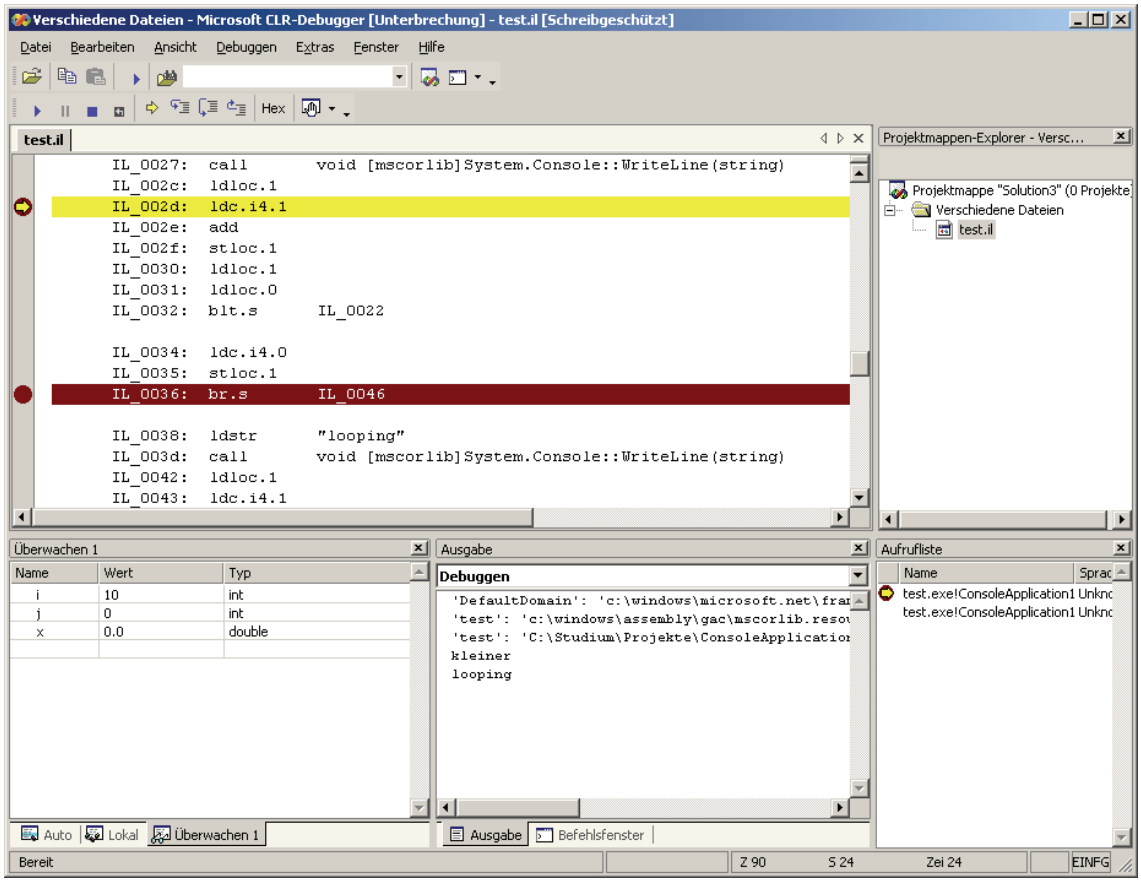


**Abbildung 6.6: Der CLR-Debugger in Aktion - Haltepunkte, Überwachen, Ausgabe, Aufrufliste**

### 6.5.12 Verifizieren des IL-Codes

Mit Hilfe des Tools *PEVerify* kann der erstellte IL-Code verifiziert werden. Diese Möglichkeit trägt ebenfalls, wie das Debuggen, zur Fehlersuche bei. Manchmal entstehen Fehler, die vom ILASM-Compiler nicht als solche erkannt werden und dann während der Laufzeit z.B. eine *InvalidProgramException* verursachen. Die Ursachen solcher Fehler können verschieden sein und manchmal versagt dann hier auch der Debugger[3].

Das Tool PEVerify hilft bei der Diagnose und Suche solcher Fehler. PEVerify stellt sicher, dass ein Assembly den Regeln für die Metadatendefinitionen und der Semantik der Spezifikation der Common Language Infrastructure (CLI)[4] entspricht. Listing 6.8 zeigt ein Beispiel mit einigen Fehlern, die der ILASM-Compiler nicht erkennt.

Wenn der Code sorgfältig überprüft wird, sind einige Fehler schnell zu finden. Beispielsweise ist *Int33* kein gültiger Datentyp in der *mscorlib* (Zeile 17) und *AddArguments* hat kein viertes Argument (Zeile 29, *ldarg.3* sollte *ldarg.2* heissen). Es sind aber auch Fehler vorhanden, die nicht auf den ersten Blick zu finden sind. So ist z.B. in der Funktion *Main()* der Wert von *.maxstack* falsch (Zeile 9, sollte 3 sein). In diesem kurzen Beispiel ist dieser Fehler sicher noch leicht zu finden. Jedoch bei größeren und komplexeren Assemblies wird es schon schwieriger. Gerade dieser Fehler bei *.maxstack* führt dazu, dass sich das Programm nicht debuggen lässt.

Der ILASM-Compiler erstellt das Programm ohne eine Fehlermeldung. Die Ausführung des Programms führt dann zu einer *InvalidProgramException*.

```
Unbehandelte Ausnahme:  System.InvalidProgramException:
Die Common Language Runtime hat ein ungültiges Programm gefunden.
at VerifyTest.Main(String[] args)
```

---

[3] Der Debugger gibt nur an, in welcher Methode die Exception ausgelöst wurde. Jedoch ist ein Debuggen der Methode nicht möglich.

[4] Kapitel 21 aus dem Teil 2 zeigt die Metadaten – Validierungsregeln [D4]. Zu jedem CIL-Opcode im dritten Teil existiert ein Abschnitt über die Verifizierung [D5].

**Listing 6.8: Fehler zum Testen von PEVerify**

```
01 .class public beforefieldinit VerifyTest
02     extends [mscorlib]System.Object
03 {
04     .method private hidebysig static void
05         Main(string[] args) cil managed
06     {
07         .entrypoint
08         .locals init (int32 retVal)
09         .maxstack 1
10         break
11         ldc.i4.0
12         ldc.i4.1
13         ldc.i4.2
14         call void VerifyTest::AddArguments(int32, int32, int32)
15         stloc.0
16         ldloca 0 //  Print the value.
17         call instance string [mscorlib]System.Int33::ToString()
18         call void [mscorlib]System.Console::WriteLine(string)
19         ret
20     }
21
22     .method private hidebysig static int32
23         AddArguments(int32 X, int32 Y, int32 Z)
24     {
25         .maxstack 2
26         ldarg.0
27         ldarg.1
28         add
29         ldarg.3
30         add
31         ret
32     }
33 }
```

PEVerify bringt hierzu folgende Ausgabe:

```
C:\Studium\Projekte\PEVerifyTest>peverify /md /il PEVerifyTest.exe

Microsoft (R) .NET Framework PE Verifier  Version 1.1.4322.573

Copyright (C) Microsoft Corporation 1998-2002. All rights reserved.

[IL]: Error:

[c:\studium\projekte\peverifytest\peverifytest.exe:VerifyTest::Main]

[Offset 0x00000002] [Opcode ldc.i4.1] Stapelueberlauf.

[IL]: Error:

[c:\studium\projekte\peverifytest\peverifytest.exe:VerifyTest::Main]

[HRESULT 0x80004005] - Unbekannter Fehler

[IL]: Error:

[c:\studium\projekte\peverifytest\peverifytest.exe:VerifyTest::AddArguments]
[Offset 0x00000003] [Opcode ldarg.3]

[Argument #0x00000003] Unbekannte Argumentennummer.

[IL]: Error:

[c:\studium\projekte\peverifytest\peverifytest.exe:VerifyTest::AddArguments]

[HRESULT 0x80004005] - Unbekannter Fehler

4 Errors Verifying PEVerifyTest.exe
```

Die Optionen /md und /il sorgen dafür, dass PEVerify die Metadaten und die IL-Instruktionen verifiziert. Wird keine dieser Optionen spezifiziert, dann werden erst die Metadaten überprüft und wenn keine Fehler auftreten, dann die IL-Instruktionen. Beide Optionen sorgen dafür, dass alles überprüft wird, auch wenn die Metadaten Fehler aufzeigen.

In diesem Fall bringt PEVerify vier Fehler. Der erste Fehler zeigt, dass der zweite Ausdruck in *Main()* inkorrekt ist – es wird ein Stapelüberlauf (*Stack overflow*) erzeugt. Der Offsetwert zeigt an, an welcher Stelle der Methode sich der Fehler befindet (Angabe in Byte). In diesem Fall wäre das in der Zeile 12, beim Opcode *ldc.i4.1*. Das ist genau die Stelle, an der der Stapelüberlauf passiert, weil ein weiterer Wert auf den Stack geladen wird und *.maxstack* nur den Wert 1 hat. Hier zeigt sich auch, dass PEVerify nur die Stellen anzeigt, an denen ein Fehler auftritt, aber nicht die Ursache dafür. Jedoch lassen sich anhand der Meldungen von PEVerify die Fehler und die Ursachen schnell finden.

Der dritte Fehler besagt, dass in *AddArguments()* ein nichtkorrekter Index eines Argumentes verwendet wird. Der Fehler befindet sich bei Opcode *ldarg.3* (Zeile 29), welcher natürlich *ldarg.2* sein sollte.

Die weiteren zwei Fehler zeigen keine konkreten Fehler oder Ursachen an. Nachdem die zwei oberen Fehler behoben wurden, zeigt die erneute Ausführung von PEVerify folgende zwei Fehler:

```
C:\Studium\Projekte\PEVerifyTest >peverify /md /il PEVerifyTest.exe
Microsoft (R) .NET Framework PE Verifier  Version 1.1.4322.573
Copyright (C) Microsoft Corporation 1998-2002. All rights reserved.
[IL]:Error:
[c:\studium\projekte\peverifytest\peverifytest.exe : VerifyTest::Main]
[Offset 0x00000004] [Opcode call]
[Token  0x0A000003] Token kann nicht aufgelöst werden.
[IL]:Error:
[c:\studium\projekte\peverifytest\peverifytest.exe : VerifyTest::Main]
[HRESULT 0x80004005] - Unbekannter Fehler
2 Errors Verifying PEVerifyTest.exe
```

Der zweite Fehler bleibt weiterhin unbekannt, doch der Erste hat jetzt etwas mehr Aussagekraft. Wird der Code einer genaueren Prüfung unterzogen, zeigt sich, dass der Rückgabewert von AddArguments() falsch ist (Zeile 14). Die Methode gibt einen Integerwert zurück. Die Zeile sollte demnach so aussehen:

```
call int32 VerifyTest::AddArguments(int32, int32, int32)
```

Eine erneute Kompilierung und Überprüfung mit PEVerify zeigt nun an, dass alle Klassen und Methoden verifiziert wurden.

```
C:\Studium\Projekte\PEVerifyTest >peverify /md /il PEVerifyTest.exe
Microsoft (R) .NET Framework PE Verifier  Version 1.1.4322.573
Copyright (C) Microsoft Corporation 1998-2002. All rights reserved.

All Classes and Methods in PEVerifyTest.exe Verified.
```

Wird PEVerifyTest nun ausgeführt, so öffnet sich die Dialogbox des Just-In-Debugger, da sich in *Main()* noch ein *break* befindet. Das Ignorieren des Debuggers führt zu der Ausgabe „3“, d.h. *Main()* wurde korrekt ausgeführt (0+1+2 = 3).

# 7 Konzept zum Lösungsansatz

Die Bibliothek stellt die API für den Nutzer dar und beinhaltet alle Elemente, die für die prozessorientierte Simulation notwendig sind. Da MODSIM III einen sehr großen Funktionsumfang hat, werden vorerst nur die wichtigsten Elemente umgesetzt – die TELL-Methode, WAIT- und WAITFOR-Aufrufe, INTERRUPT-Aufrufe sowie die Verwendung von Ressourcen mit statistischer Auswertung.

Weiterhin beinhaltet die Bibliothek Klassen und Funktionen, die vom Enhancer zur Erweiterung des vom Nutzer erstellten Codes benötig werden.

Der Enhancer nimmt die notwendigen Erweiterungen im IL-Code vor, um das Prinzip einer Coroutine umzusetzen.

## 7.1 Die SimNet – Bibliothek

### 7.1.1 Der Scheduler

Der Scheduler verwaltet eine Liste der Elemente, welche sequentiell abgearbeitet werden. Die Elemente werden nach Zeit und Priorität in der Liste gespeichert und dementsprechend verarbeitet. Die Simulationszeit wird immer auf den Zeitpunkt gesetzt, an dem die nächsten anstehenden Aktivitäten vorliegen.

Weiterhin enthält der Scheduler Listen zum Verarbeiten der unterbrochenen Aktivitäten (*INTERRUPT*), der wartenden Aktivitäten (*WAITFOR*) und der Aktivitäten, die Ressourcen verwenden.

### 7.1.2 Umsetzung der TELL – Methode

Der Nutzer kann jede Methode als TELL-Methode mittels eines Attributes kennzeichnen. Die Kennzeichnung ist notwendig, damit der Enhancer die TELL-Methoden herausfiltern kann. Diese Methoden müssen bestimmte Parameter enthalten – die Startzeit der Methode, die Priorität der Methode und eine Parameterliste mit den nutzerspezifischen Parametern für diese Methode. Die

Verwendung einer solchen Parameterliste ist für die Umsetzung des Coroutinen-Prinzips nötig, da die Anzahl der vom Nutzer festgelegten Parameter unbestimmt ist und auch diese beim Verlassen einer TELL-Methode (z.B. mittels WAIT) zwischengespeichert werden müssen. Würde hier keine Liste verwendet, sondern die Parameter ganz normal übergeben werden, so wäre die Analyse im Enhancer sehr aufwendig und komplex.

Die TELL-Methode wird mittels eines Delegates umgesetzt, Dieses wird in speziellen Objekten in der Liste im Scheduler gespeichert und eben stellvertretend vom Scheduler bei der Verarbeitung der Liste aufgerufen.

Der Aufruf einer TELL-Methode erfolgt über eine spezielle Methode, welche als Parameter das Delegate, die Startzeit, die Priorität und die Liste der nutzerspezifischen Parameter enthält. In dieser Methode werden alle Parameter in einem Objekt gekapselt und dieses in die Liste des Schedulers eingetragen.

```
Tell(TellMethod tm, double start, double priority, object[] list)
```

### 7.1.3 Umsetzung der WAIT – Aufrufe

Ein WAIT-Aufruf enthält zwei Parameter: die Zeit die gewartet werden soll sowie ein Parameter zum Unterbrechen der Methode. Hierbei wird die Startzeit der aktuellen Aktivität um die übergebene Zeit erhöht und die Liste vom Scheduler neu geordnet, so dass die Aktivität für eine entsprechende Zeit die Arbeit suspendiert.

Da der erste Parameter nicht nur ein einfacher Wert, sondern auch ein Ausdruck oder Methodenaufruf mit Rückgabewert sein kann, muss die WAIT-Anweisung in einem *try/catch*-Block stehen. Das Einfügen der Erweiterungen durch den Enhancer wird somit vereinfacht und beugt Fehler vor.

```
try
{
  Simulation.Wait(5, ref interrupt);
}catch{}
```

### 7.1.4 Umsetzung der WAITFOR – Aufrufe

Ein WAITFOR-Aufruf enthält im Fall der TELL-Methode folgende Paramenter: das Delegate der Methode, die gestartet werden soll, die Startzeit, die Priorität und die Parameterliste dieser Methode sowie einen Parameter zum Unterbrechen der Methode.

```
WaitFor( TellMethod tm, double t, double priority, object[] list,
            ref bool interrupt)
```

Diese Parameter werden in einem neuen Objekt gekapselt und dieses in die Schedulerliste eingetragen. Das aktuelle Objekt wird aus dieser Liste entfernt und solange in der Liste der wartenden Aktivitäten gespeichert, bis die gestartete Aktivität ihre Arbeit beendet hat.

Eine WAITFOR-Anweisung muss ebenfalls in einem *try/catch*-Block stehen.

### 7.1.5 Umsetzung der INTERRUPT – Aufrufe

In MODSIM III gibt es einige Möglichkeiten, um aktive Objekte zu unterbrechen. Im Konzept sind vorerst nur zwei Varianten vorgesehen – das Unterbrechen einer bestimmten Methode eines Objektes sowie das Unterbrechen aller aktiven Methoden eines Objektes.

Die erste Variante beinhaltet zwei Parameter beim Aufruf: das Objekt und den Namen der Methode. Hier wird die erste gefundene Methode mit diesem Namen unterbrochen.

Bei der zweiten Variante wird nur das Objekt übergeben und es werden alle gefunden Methoden dieses Objektes unterbrochen.

In jedem Fall werden gefundene Methoden über einen Member des Objektes, in dem sie im Scheduler gekapselt sind, als unterbrochen markiert und in eine weitere Liste im Scheduler aufgenommen. Diese Liste wird zu jedem Zeitpunkt, nach dem alle Aktivitäten abgearbeitet wurden, im Scheduler durchlaufen. Werden Objekte gefunden, die unterbrochen wurden, so wird die Zeit, an der diese ihre Arbeit wiederaufnehmen sollen, auf die aktuelle Simulationszeit

gesetzt und die Schedulerliste neu geordnet. Dadurch stehen alle unterbrochenen Objekte an erster Stelle und werden sofort verarbeitet.

Damit der Nutzer die Möglichkeiten des Interrupts verwenden kann, muss nach jeder WAIT- oder WAITFOR-Anweisung, nach dem try/catch-Block eine if-Anweisung zur Auswertung des Interrupt-Parameters folgen.

```
try
{
   Simulation.Wait(5, ref interrupt);
}
catch{}
if(interrupt) //Methode wurde unterbrochen
{
   . . .
}
else //keine Unterbrechung
{
      . . .
}
```

### 7.1.6 Umsetzung der Ressourcen

Ressourcen stellen in MODSIM III ein wichtiges Mittel zur Simulation dar. Eine Ressource ist ein Objekttyp mit limitierter Kapazität. Objekte können einen oder mehrere Plätze der Ressource belegen bzw. anfordern. Beispielsweise hat ein Parkplatz eine Kapazität von 100 Stellplätzen, von denen jeder PKW genau einen Stellplatz, jedoch ein LKW acht belegt.

Die Umsetzung der Ressourceobjekte erfolgt durch eine eigene Klasse, welche die notwendigen Funktionen inkl. statistischer Auswertung beinhaltet.

Die Anforderung der Plätze einer Ressource erfolgt mittels einer Methode, welche mit einer WAITFOR-Anweisung aufgerufen wird. Dies ist notwendig, da das anfordernde Objekt solange warten muss bis die Ressourcen zu Verfügung stehen.

### 7.1.7 Zwischenspeichern der lokalen Variablen

Das Speichern und Wiederherstellen der Variablen erfolgt hauptsächlich durch den Enhancer und auf Ebene der Intermediate Language. Zur Speicherung der lokalen Variablen wird die Klasse aus der Bibliothek verwendet, in der auch die Delegates und Parameter der TELL-Methoden gekapselt sind. Die Werte der lokalen Variablen werden dabei in einem Array vom Typ *object* abgelegt. Über spezielle Methoden erfolgt der Zugriff vom Enhancer auf dieses Array, um die Werte der Variablen zu speichern und wiederherzustellen.

## 7.2 Der Enhancer

### 7.2.1 Aufgabe des Enhancers

Der Enhancer hat die Aufgabe, den vom Nutzer erstellten und kompilierten Code zu erweitern. Mit Hilfe des IL-Disassemblers (ildasm.exe) wird das erstellte Assembly in IL-Code umgewandelt. Dieser IL-Code wird vom Enhancer eingelesen, analysiert und an den entsprechenden Stellen mit den notwendigen Elementen erweitert. Anschließend wird der erweiterte IL-Code durch den IL-Assembler wieder kompiliert und kann dann ausgeführt werden.

Die Erweiterungen werden nur in Methoden vorgenommen, die mit einem TELL-Attribut gekennzeichnet sind, da nur TELL-Methoden mittels WAIT oder WAITFOR verlassen werden können.

### 7.2.2 Die Erweiterungen

#### 7.2.2.1 Lokale Variablen

Die lokalen Variablen der TELL-Methoden werden um drei Variablen erweitert, welche für die Auswertung des Labels sowie zum Speichern und Wiederherstellen benötigt werden. Besitzt eine Methode keine lokalen Variablen, so wird die Methode um eine *.locals init*-Anweisung erweitert und die drei Variablen hier eingefügt.

#### 7.2.2.2 Label/Marke zum Wiedereinstieg in die Methode

Am Anfang jeder TELL-Methode wird eine Auswertung des gespeicherten Labels vorgenommen. Je nachdem welchen Wert dieses Label besitzt, wird zu der entsprechenden Marke gesprungen. Die Marken werden hierzu nach jeder WAIT- oder WAITFOR-Anweisung eingefügt und dienen somit zum Wiedereinstieg in die Methode.

#### 7.2.2.3 Speichern und Wiederherstellen der lokalen Variablen

Die lokalen Variablen müssen jedes Mal bevor die Methode verlassen wird zwischengespeichert, und entsprechend bei der Rückkehr in die Methode, vor der eigentlichen Codeausführung, wiederhergestellt werden.

Die Variablen werden in einem Array vom Typ *object* abgelegt und in der entsprechenden Klasse (Typ *SimObj*) des Objektes gespeichert.

Die Codeerweiterung erfolgt vor und nach jeder WAIT- oder WAITFOR-Anweisung.

#### 7.2.2.4 Verlassen der Methode

Zum Verlassen der Methode wird eine *return*-Anweisung unmittelbar nach der WAIT- oder WAITFOR-Anweisung eingefügt.

#### 7.2.2.5 Entfernen des Objektes aus der Schedulerliste

Wenn das Ende einer TELL-Methode erreicht ist, muss das Objekt aus der Schedulerliste gelöscht werden. Hierzu wird eine Erweiterung eingefügt, welche einen Funktionsaufruf einer Methode zum Löschen des Objektes beinhaltet.

## 7.3 Einbinden des Enhancer in das Visual Studio

Der Enhancer ist ein eigenständiges Programm und muss nach jeder Kompilierung des Codes auf der Konsole ausgeführt werden. Um diesen Prozess zu automatisieren bietet sich die Verwendung der Build-Events des Visual Studio an, welche die Ausführung des Enhancer nach erfolgreicher Kompilierung automatisch übernimmt. Diese Buildereignisse sind ab der Version 2003 des Visual Studio integriert. Dabei können Präbuildereignisse und Postbuildereignisse verwendet werden.

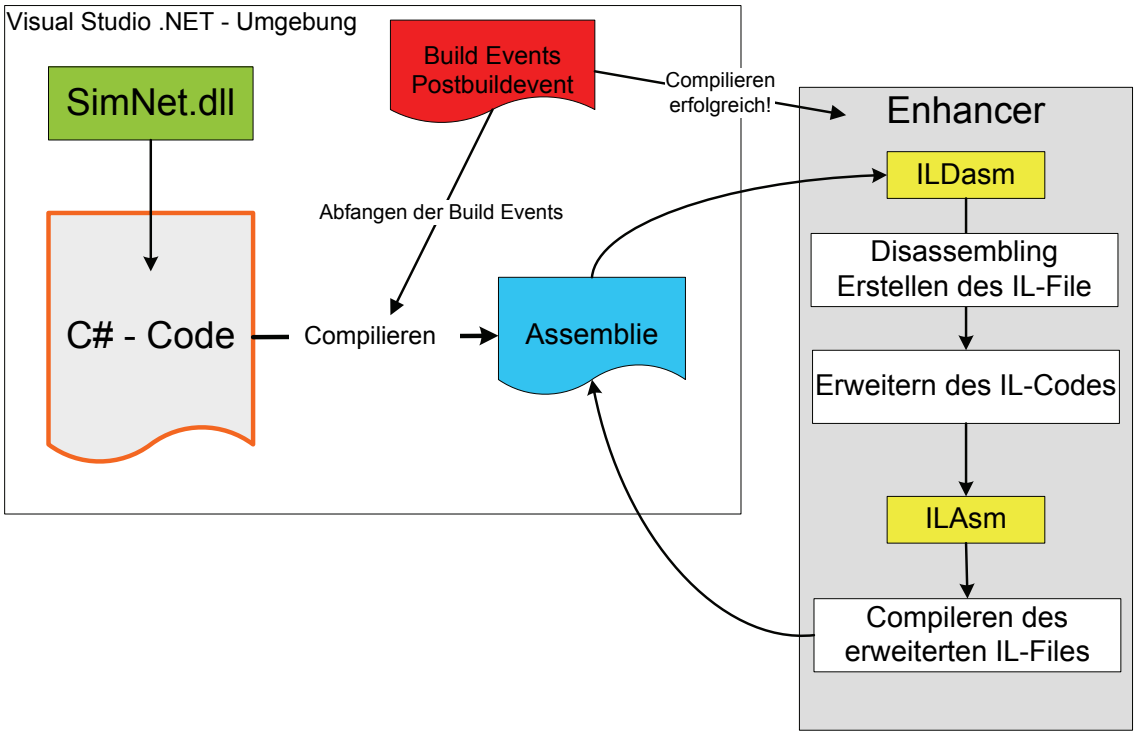


**Abbildung 7.1: Das Konzept in grafischer Darstellung**

# 8 Implementierung des Konzeptes

## 8.1 SimNet.dll – Bibliothek zur prozessorientierten Simulation

### 8.1.1 Klasse Simulation

Die Klasse Simulation stellt die unmittelbare Schnittstelle zum Nutzer dar. Hier werden alle notwendigen Funktionen zur prozessorientierten Simulation deklariert. Die Tabelle 8.1 zeigt eine Übersicht der implementierten Funktionen.

#### 8.1.1.1 Die Tell-Methode

In der Klasse Simulation befindet sich auch das Delegate zur Umsetzung der TELL-Methoden. Dieses Delegate wird vom Scheduler verwendet, um stellvertretend die entsprechenden TELL-Methoden aufzurufen.

```
public delegate void TellMethod( double start, double priority,
                                 params object[] list);
```

In der eigentlichen Methode *Tell()* zum Ausführen einer solchen TELL-Methode wird eine neue Instanz der Klasse *SimObj* angelegt und in ihr die TELL-Methode sowie die zugehörigen Parameter gespeichert. Anschließend wird dieses Objekt zur Verarbeitung in die Schedulerliste aufgenommen.

#### 8.1.1.2 Die Wait-Methode

Hier wird der entsprechende Wert (*futureTime*) der aktuellen Instanz der Klasse *SimObj* neu gesetzt und die Liste vom Scheduler neu geordnet. Dadurch wird erreicht, dass das aktuelle Objekt die entsprechenden Zeiteinheiten wartet.

**Tabelle 8.1: Methoden der Klasse Simulation**

| Name d. Fkt. | Parameter | Beschreibung |
|---|---|---|
| StartSimulation | | Starten der Simulation. |
| StopSimulation | | Stoppen der Simulation. |
| SimTime | | Gibt die aktuelle Simulationszeit zurück. |
| Tell | TellMethod tm<br>double start<br>double priority<br>object[] list | Aufrufen einer TELL-Methode. Übergeben werden dieser Methode das Delegate zur TELL-Methode, der Startzeitpunkt, die Priorität und die Parameterliste. |
| Wait | double t<br>ref bool interrupt | Wait-Anweisung, Warten der übergebenen Zeiteinheiten. Ein weiterer Parameter dient zum Unterbrechen der Methode. |
| WaitFor | TellMethod tm<br>double t<br>double priority<br>object[] list<br>ref bool interrupt | WaitFor-Anweisung für TELL-Methoden. Übergeben wird hier die TELL-Methode, welche gestartet und auf welche gewartet wird, sowie die entsprechenden Parameter dieser TELL-Methode. Ein weiterer Parameter dient zum Unterbrechen der Methode. |
| WaitFor | GiveMethod gm<br>object obj<br>int numberDesired<br>ref bool interrupt | WaitFor-Anweisung für die Give-Methode der Ressourcen. Übergeben wird hier die entsprechende Give-Methode, welche gestartet und auf welche gewartet wird, sowie die entsprechenden Parameter dieser Give-Methode (Objekt, welches Ressourcen anfordert und Anzahl der Ressourcen). Ein weiterer Parameter dient zum Unterbrechen der Methode. |
| Interrupt | object obj<br>string method | Unterbrechen einer bestimmten Methode eines Objektes. Übergeben wird das Objekt und der Name der Methode. Falls mehrere Methoden mit gleichem Namen existieren, wird die Methode unterbochen, welche als erstes in der Liste steht. |
| InterruptAll | object obj | Unterbrechen aller Methoden eines Objektes. |

#### 8.1.1.3 Die WaitFor-Methoden

Die WAITFOR-Anweisungen werden über die Methode *WaitFor()* realisiert. Hier werden ebenfalls Delegates verwendet, wobei für jede Art von Methode, die über eine WAITFOR-Anweisung gestartet wird, ein entsprechendes Delegate vorhanden sein und die *WaitFor*-Methode entsprechend überladen werden muss. In der bisherigen Implementierung können mittels *WaitFor()* TELL-Methoden und die *Give*-Methode (siehe 8.1.5) aufgerufen werden.

In der ersten *WaitFor*-Methode wird eine neue Instanz der Klasse *SimObj* angelegt und in diesem Objekt die TELL-Methode sowie die zugehörigen Parameter gespeichert. Diese Instanz wird dann im aktuellen Objekt eingetragen und zur Verarbeitung in die Schedulerliste aufgenommen, wobei das aktuelle Objekt in einer weiteren Liste im Scheduler (Liste der wartenden Objekte) zwischengespeichert und aus der Schedulerliste entfernt wird.

In der zweiten *WaitFor*-Methode, welche verwendet wird um Ressourcen anzufordern, wird die entsprechende *Give*-Methode der Klasse *ResourceObj* aufgerufen und in Abhängigkeit des Rückgabewertes entsprechend fortgefahren. Wird *false* zurückgegeben, so sind im Moment die angeforderten Ressourcen nicht verfügbar und das aktuelle Objekt muss warten. Hierzu wird im aktuellen Objekt wieder das Objekt, auf das gewartet werden muss eingetragen und das aktuelle Objekt in die Warteliste des Schedulers verschoben.

#### 8.1.1.4 Die Interrupt-Methoden

In den Interrupt-Methoden wird nach dem entsprechenden Objekt (und im Fall von *Interrupt()* auch nach der Methode) in der Schedulerliste gesucht. Wird dieses Objekt gefunden, so wird das Interrupt-Flag gesetzt und das Objekt in die Interrupt-Liste des Schedulers eingefügt.

### 8.1.2 Klasse Scheduler

Der Scheduler verwaltet und organisiert den Ablauf der Simulation. Hier werden insgesamt vier Listen verwaltet:

- die Schedulerliste, beinhaltet alle aktiven Objekte
- die WaitFor-Liste, hier werden die wartenden Objekte gespeichert
- die Interrupt-Liste, alle unterbrochenen Objekte werden hier aufgeführt
- die Ressourcen-Warteliste, beinhaltet die Objekte, die in den Pending-Listen der Ressourcen stehen

In der Schedulerliste werden die Aktivitäten entsprechend ihrer Simulationszeit und Priorität gespeichert und vom Scheduler abgearbeitet. Das Sortieren der Liste erfolgt durch das Bubble-Sort-Verfahren.

Sind alle Aktivitäten eines Zeitpunktes abgearbeitet, so wird vom Scheduler die Liste der wartenden und unterbrochenen Objekte überprüft. Über die Liste der wartenden Objekte wird geprüft, ob die Objekte, auf die gewartet wird noch aktiv sind. Wenn das nicht der Fall ist, wird das wartende Objekt wieder in die Schedulerliste aufgenommen. Befinden sich in der Interrupt-Liste Objekte, so wird nach diesen in der Schedulerliste gesucht und sie werden unterbrochen, d.h. es wird die aktuellen Simulationszeit eingetragen und das Interrupt-Flag entsprechend gesetzt. Damit hier eine Unterbrechung erfolgen kann, muss der Nutzer über eine *if*-Anweisung den Interrupt-Parameter auswerten und dann dementsprechend eine alternative Codeausführung anbieten.

### 8.1.3 Klasse SimObj

Die Klasse *SimObj* dient zum Abspeichern der aktiven Objekte. Instanzen dieser Klasse werden in der Schedulerliste verwaltet. Die folgende Tabelle listet die Member der Klasse auf.

**Tabelle 8.2: Member der Klasse SimObj**

| Typ und Name | Beschreibung |
|---|---|
| object method | In diesem Member wird das Delegate, also die eigentliche TELL-Methode gespeichert. |
| string label | Hier wird aktuelle Sprungmarke abgespeichert. Wird bei Rückkehr in die Funktion abgefragt und dann an entsprechende die Stelle gesprungen. |
| object[] locals | In diesem Array werden die lokalen Variablen abgelegt. |
| object[] par | Zum Speichern der Parameterliste der TELL-Methoden. |
| double priority | Die Priorität der TELL-Methode. |
| double futureTime | Startzeit der TELL-Methode. |
| bool interrupt | Interrupt-Flag. Wird true, wenn Methode unterbrochen wird. |
| object waitfor | Hier wird im Wartefall das Objekt abgespeichert, auf das gewartet werden muss. |

### 8.1.4 Klasse TELLAttribute

Die Klasse *TELLAttribute* beinhaltet die Definition des TELL-Attributes, welches der Nutzer verwenden muss, um eine TELL-Methode als solche zu kennzeichnen.

```
[TELL]
public void Generate(double t, double priority, params object[] p)
{
   . . .
}
```

### 8.1.5 Klasse ResourceObj und ResObj

Die Klasse *ResourceObj* stellt dem Nutzer alle notwendigen Funktionen zur Verwendung von Ressourcen zur Verfügung. Die Tabelle 8.3 zeigt eine Übersicht der wichtigsten Funktionen. In MODSIM III gibt es vier Methoden zum Anfordern von Ressourcen – hier ist vorerst nur eine implementiert, die *Give*-Methode. Mit dieser erfolgt die Anforderung ohne Priorität und zeitliche Einschränkung. Die Funktion *TakeBack()* gibt die Ressourcen wieder frei, durchläuft anschließend die Warteliste mit den weiteren Anfordungen und ruft entsprechend wieder *Give()* auf.

Die Klasse *ResObj* dient zum Zwischenspeichern, der auf Ressourcen wartenden Objekte.

### 8.1.6 Klasse RandomObj

Die Klasse *RandomObj* stellt Funktionen zur Erzeugung von Zufallszahlen zur Verfügung. Hierbei sind die wichtigsten Verteilungen integriert worden. Die Tabelle im Anhang C gibt eine Übersicht zu den theoretischen Verteilungsfunktionen.

**Tabelle 8.3: Funktionen der Klasse ResourceObj**

| Name der Funktion | Parameter | Beschreibung |
|---|---|---|
| **CreateResource** | int numberOfResources | Initialisierung des Ressource-Objektes mit der angegeben Anzahl von Ressourcen. Es wird die Kapazität der Ressource festgelegt. |
| **IncrementResources** | int increaseBy | Vergrößern der Kapazität des Ressource-Objektes um den angegebenen Wert. |
| **Give** | object obj<br>int numberDesired | Stellt die Anzahl von benötigten Ressourcen dem anfragenden Objekt zur Verfügung. Die aufrufende Methode wird solange geblockt, bis die Ressourcen verfügbar sind. Wenn also keine Ressourcen verfügbar sind werden die Anfragen werden in die Warteliste (PendingList) aufgenommen nach dem Prinzip Frist-In First-Out verarbeitet. |
| **TakeBack** | object obj<br>int numberReturned | Hier wird die angegebene Anzahl an Ressourcen wieder freigegeben. |

## 8.2 ILEnhancer – Erweitern der Assemblies

Der Enhancer ist als eigenständiges Programm konzipiert und implementiert. Das Programm kann auf der Konsole oder über die Buildereignisse des Visual Studio ausgeführt werden. Der Enhancer verwendet den IL-Disassembler (*ildasm.exe*) und den ILAsm-Compiler (*ilasm.exe*), um das Assembly zu verarbeiten.

### 8.2.1 Klasse ILEnhancer

Die Klasse *ILEnhancer* besitzt zwei wichtige Methoden. Die Methode *Enhance()* leitet die Erweiterung ein. Zuerst wird hier das Argument, mit dem der Enhancer gestartet wird (Name des Assembly) an die Klasse *ILDasm* übergeben und das Disassembling eingeleitet. Anschließend wird die IL-Datei in der Klasse *ILFile* analysiert und für die Erweiterung vorbereitet. In der Methode *EnhanceMethods()* erfolgt das Erweitern des Assembly. Hier werden die Extensions der Klassen *ILSimNetExtensions* initialisiert und entsprechend in die IL-Datei eingefügt. Im Anschluß erfolgt die Kompilierung des erweiterten IL-Codes durch den ILASM-Compiler.

### 8.2.2 Klasse ILDasm und ILAsm

Zum Ausführen des ILASM-Compilers und des IL-Disassemblers wird die *Process*-Klasse aus dem Namespace *System.Diagnostics* verwendet. Im Konstruktor beider Klassen wird jeweils nach den EXE-Dateien der Programme gesucht.

Der ILASM-Compiler ist Bestandteil des .NET Frameworks und liegt somit auch im Verzeichnis des Frameworks. Um eine versionsunabhängige Pfadangabe zum Verzeichnis des .NET Frameworks zu erhalten, wird in der Registry nach der CLSID einer bestimmten .NET-DLL gesucht.

**HKEY_CLASSES_ROOT\CLSID\05EBA309-0164-11D3-8729-00C04F79ED0D\InprocServer32**

Dieser Key führt zum Pfad der *mscorld.dll*, die den Assembly-Lader von .NET beherbergt. Diese DLL liegt im .NET-Verzeichnis, also dort, wo sich auch die *ilasm.exe* befindet. Wenn die Suche nach dem Registry-Key fehlschlägt, werden zusätzlich noch alle Pfade der Enviroment-Variable PATH durchsucht, so dass auch auf diesem Weg der ILASM-Compiler gefunden werden kann, vorausgesetzt, der Pfad ist in der PATH-Variable eingetragen.

Der IL-Disassembler gehört zum Visual Studio und befindet sich somit im Visual Studio-Verzeichnisbaum. Bei der Installation von Visual Studio wird normalerweise der folgende Registry-Key erzeugt:

**HKEY_CLASSES_ROOT\Applications\ildasm.exe\shell\open\command**

Der dort abgelegte String enthält den Pfad zum IL-Disassembler. Zur Sicherheit ist auch hier die Abfrage über die PATH-Variable implementiert, da es auch VS-Versionen gibt, die den IL-Disassembler nicht enthalten. So kann auch hier der Pfad zum Programm über PATH gesetzt und somit gefunden werden.

Beide Klassen weisen jeweils die Funktion *DoIt()* auf, die das Disassemblieren bzw. das Assemblieren vornehmen. Im Fall der Klasse *ILDasm* erwartet diese Methode zwei Strings als Parameter: Den Namen des Assembly, welches zu disassemblieren ist und den Namen der IL-Datei, in die das Resultat geschrieben wird. Die Funktion selber ist eigentlich nur ein Wrapper, um den Aufruf von *Process.Start,* analog wie in Listing 8.1 zu sehen.

In der Klasse *ILAsm* wird neben dem Namen der zu assemblierenden Datei und dem Namen der DLL bzw. EXE-Datei noch ein boolscher Wert als Parameter benötigt, der angibt, ob die Debug-Information mit erzeugt werden soll. Der ILASM-Compiler kann durch eine Menge an Switches gesteuert werden. Diese Switches werden am Anfang der Funktion *DoIt()* zusammengestellt. Interessant dabei ist die Erzeugung des Switches "/resource". Der IL-Disassembler extrahiert .res-Dateien aus den Assemblies, wenn in den Assemblies Ressourcen existieren. Diese .res-Dateien müssen beim ILASM-Compiler als Parameter angegeben werden, damit das Assembly hinterher auch wieder

funktioniert. Hierzu wird getestet, ob es eine Datei gibt, die wie die DLL bzw. EXE heißt, nur eben die Endung ".res" aufweist. Das Listing 8.1 zeigt die Funktion *DoIt()* der Klasse *ILAsm*.

**Listing 8.1: Die Funktion DoIt() der Klasse ILAsm**

```
public void DoIt(string ilFileName, string dllFileName, bool debug)
{
   if (!File.Exists(ilFileName))
      throw new Exception("Assembling: File not found: " + ilFileName);
   if (!File.Exists(dllFileName))
      throw new Exception("Assembling: File not found" + dllFileName);

       DateTime ct = File.GetCreationTime(dllFileName);
   DateTime at = File.GetLastAccessTime(dllFileName);
   DateTime wt = File.GetLastWriteTime(dllFileName);
   // DLL oder EXE?
   string libMode = "/" + Path.GetExtension(dllFileName).Substring(1).ToUpper();
   string debugMode = debug ? " /DEBUG" : string.Empty;
   string resourceFile = Path.ChangeExtension(dllFileName, ".res");
   string resource = File.Exists(resourceFile) ? " /RESOURCE=" + resourceFile :
                              string.Empty;
   string parameters =  libMode +" "
            + " /DEBUG "
            + ilFileName
            + " /OUTPUT=" + dllFileName
            + resource;
   ProcessStartInfo psi = new ProcessStartInfo(ilAsmPath, parameters);
   psi.CreateNoWindow = true;
   psi.UseShellExecute = false;
   psi.WorkingDirectory = Path.GetDirectoryName(dllFileName);
   psi.RedirectStandardOutput = true;
   psi.RedirectStandardError = true;
   System.Diagnostics.Process proc = System.Diagnostics.Process.Start( psi );
   proc.WaitForExit();

       string stderr = proc.StandardError.ReadToEnd();
   if ( stderr != null && 0 < stderr.Length )
   {
      if(!stderr.StartsWith("// WARNING"))
         throw new Exception ("ILAsm: " + stderr);
   }
   else
   {
      File.SetCreationTime(dllFileName, ct);
      File.SetLastAccessTime(dllFileName, at);
      File.SetLastWriteTime(dllFileName, wt);
   }
}
```

Nach dem Erzeugen des Assembly werden sämtliche Zeitangaben der neu erzeugten Datei auf die Angaben der alten Datei gesetzt. Dies ist notwendig, damit die Build-Logik des Visual Studio nicht durcheinander kommt. Auf diese Weise „denkt“ Visual Studio, die erweiterte DLL sei die original erzeugte DLL.

### 8.2.3 Ein IL-Code-Baum

Eine der zentralen Aufgaben des Enhancers ist es, den IL-Code zu analysieren und in einem Baum aufzubereiten. Die oberste Klasse der Hierarchie ist die Klasse *ILFile*. Sie enthält eine *ArrayList* namens *lines* für alle Zeilen des IL-Codes. Für jede Zeile wird ein Objekt vom Typ *ILLineElement* erzeugt, das dann an die *ArrayList* angehängt wird. Es wird somit Zeile für Zeile gelesen, bis ein IL-Befehl auftaucht, der den Beginn eines Elements kennzeichnet, das in der Hierarchie unter *ILFile* liegt. In der bisherigen Implementierung sind das ausschließlich Methoden, die in IL durch das Schlüsselwort *.method* gekennzeichnet sind. Wird ein solches Element gefunden, so wird ein Objekt vom Typ *ILMethodElement* erzeugt und mit der Analyse dieses Elements fortgefahren. Es wird wieder Zeile für Zeile analysiert und folgende wichtige Punkte abgearbeitet:

- Die Deklaration der Methode wird analysiert und der Name der Methode ausgelesen.
- Über das Schlüsselwort *.custom instance* wird überprüft, ob es sich um eine TELL-Methode handelt.
- Die Zeile mit *.maxstack* wird ausgelesen und zwischengespeichert.
- Anhand des Schlüsselwortes *.locals init* werden die lokalen Variablen der Methode gefunden und anschließend deren Typen und Namen ausgelesen und abgespeichert.
- Alle Zeilen in denen sich ein WAIT- oder WAITFOR-Aufruf befindet werden in einer *ArrayList* abgespeichert.

### 8.2.4 Die IL-Erweiterungen - Klasse ILSimNetExtensions

Der wohl wichtigste Part des Enhancers ist die Erweiterung des IL-Codes. In den vorhandenen IL-Code werden Elemente eingefügt, um das Prinzip der Coroutine umzusetzen. D.h. es muss ein Verlassen einer Methode für eine unbestimmte Zeit möglich sein und bei der Rückkehr in die Methode müssen die lokalen Variablen im selben Zustand, wie beim Verlassen sein.

Die Erweiterung erfolgt in mehreren Schritten: Zuerst wird in dem Array, in dem alle Methoden gespeichert sind (*ILMethodElement*) nach TELL-Methoden gesucht, denn nur diese werden erweitert. Anschließend werden die lokalen Variablen der Methode um drei weitere aufgestockt. Besitzt die Methode keine lokalen Variablen, so wird eine *.locals init*-Direktive eingefügt. Danach wird am Anfang der Methode der Block zur Abfrage der Sprungmarke eingefügt. Hier wird jedes Mal bei der Rückkehr die Marke aus der Klasse *SimObj* geholt und mit den entsprechenden Marken der Methode verglichen und eben an diese Stelle gesprungen. Die Anzahl der Marken, die in einer Methode gesetzt werden, ist abhängig von der Menge der WAIT- bzw. WAITFOR-Anweisungen. Im nächsten Schritt werden vor und nach jeder dieser Anweisungen die Erweiterungen zum Speichern und Wiederherstellen der lokalen Variablen und der Parameterliste eingefügt. Der *try/catch*-Block, den jede WAIT- bzw. WAITFOR-Anweisung umschließen muss, dient hierbei als Anhaltspunkt für das Einfügen der Elemente. Die Elemente zum Speichern werden direkt am Anfang des *try*-Blocks eingefügt und die zum Wiederherstellen außerhalb, direkt nach dem *catch*-Block. An dieser Stelle befindet sich dann auch die Marke zum Wiedereinstieg in die Methode. Im *try*-Block wird noch direkt nach der WAIT- bzw. WAITFOR-Anweisung, die Anweisung zum Verlassen der Methode eingefügt. Sind alle Anweisungen der Methode abgearbeitet, so wird am Ende, direkt vor der Return-Anweisung noch das Element zum Löschen des Objektes aus der Schedulerliste eingefügt.

Das Listing 8.2 zeigt den C#-Code eines Beispielprogramms. In dem Programm werden in der *Main()*-Funktion zwei TELL-Methoden aufgerufen, einmal die Methode *Generate()* und einmal die Methode *Generate2()*. Die zweite TELL-Methode startet zum Zeitpunkt 0.0 und die erste zum Zeitpunkt 10.0. Beide Methoden haben als nutzerspezifischen Parameter einen Double-Wert, welcher mit der Liste *parameter* übergeben wird. Mit dem Aufruf von *Simulation.Start()* wird die Simulation gestartet. In der Schedulerliste befinden sich somit zwei Aktivitäten, wobei die Methode *Generate2()* als erstes zum Zeitpunkt 0.0 ausgeführt wird. In dieser Methode befindet sich nach der Ausgabe der Simulationszeit eine WAIT-Anweisung. Der Aufruf *Simulation.Wait()* bewirkt in diesem Fall das Warten von 10.0 Zeiteinheiten. Hier wird also der Zeitpunkt, an dem diese Methode wieder aktiv wird auf 10.0 gesetzt (aktuelle Zeit 0.0 plus 10.0) und die Liste im Scheduler neu geordnet. Demnach sind zum Zeitpunkt 0.0 keine weiteren Aktivitäten vorhanden und die Simulationszeit wird auf 10.0 gesetzt. Nun wird in die Methode *Generate2()* zurückgekehrt und in dieser mit der Codeausführung fortgefahren. Anschließend wird die Methode *Generate()* ausgeführt. In dieser Methode befindet sich ebenfalls eine WAIT-Anweisung, welche die Methode 10.0 Zeiteinheiten warten lässt. So sind zum Zeitpunkt 10.0 keine weiteren Aktivitäten vorhanden und die Simulationszeit wird auf den nächsten Zeitpunkt gesetzt, an dem wieder Aktivitäten vorliegen. In diesem Fall liegt dieser bei 20.0. Hier wird die Ausführung der Methode *Generate()* fortgesetzt. Es erfolgt eine Auswertung des Interrupt-Parameters, welcher false ist. Somit liegt keine Unterbrechung vor. Jetzt wird eine TELL-Methode aufgerufen, hier *Generate2()* mit einer Startzeit von 5.0 (Wert aus Parameterliste), d.h. diese Methode wird zum Zeitpunkt 25.0 gestartet und ausgeführt.

Das Listing 8.3 zeigt den IL-Code der Methode *Generate()* mit den vom Enhancer eingefügten Elementen. Alle Erweiterungen sind rot markiert.

**Listing 8.2: Beispiel in C#**

```
using System;
using SimNet;
using System.Collections;

namespace ConsoleApplication2
{
   class GeneratorObj
   {
      [TELL]
      public void Generate(double t, double priority, params object[] p)
      {
         bool interrupt=false;
         int x=10;
         double y = (double)p[0];

         Console.WriteLine(   "Methode Generate 1, SimTime = {0}",
                              Simulation.SimTime());
         try
         {
            Simulation.Wait(x, ref interrupt);
         }
         catch{};
         if(interrupt)
         {
            Console.WriteLine(   "Methode Generate 1 wurde unterbrochen, SimTime =
                                 {0}", Simulation.SimTime());
         }
         else
         {
         Simulation.Tell(new TellMethod(this.Generate2), y, 0.0, null);
         }
      }

      [TELL]
      public void Generate2(double t, double priority, params object[] p)
      {
         bool interrupt=false;

         Console.WriteLine(   "Methode Generate 2, SimTime = {0}",
                              Simulation.SimTime());

         try
         {
            Simulation.Wait(10, ref interrupt);
         }
         catch{};
      }
   }

   class Class1
   {
      [STAThread]
      static void Main(string[] args)
      {
         GeneratorObj gen1 = new GeneratorObj();
         GeneratorObj gen2 = new GeneratorObj();

         object[] parameter = new object[1]{5.0};

         Simulation.Tell(new TellMethod(gen1.Generate) ,10.0, 0.0, parameter);
         Simulation.Tell(new TellMethod(gen2.Generate2), 0.0, 0.0, parameter);

         Simulation.StartSimulation();

         Console.ReadLine();
      }
   }
}
```

**Listing 8.3: Die Methode Generate() im IL-Code mit den eingefügten Erweiterungen**

```
.namespace ConsoleApplication2
{
   .class private auto ansi beforefieldinit GeneratorObj
      extends [mscorlib]System.Object
   {
      .method public hidebysig instance void
         Generate(float64 t,
         float64 priority,
         object[] p) cil managed
   {
      .custom instance void [SimNet]SimNet.TELLAttribute::.ctor() = ( 01 00 00 00 )
      ...
      .maxstack 4
      .locals init ([0] bool interrupt,
      [1] int32 x,
      [2] float64 y,

      //Erweitern der lokalen Variablen
      object[] locals,
      class [SimNet]SimNet.SimObj obj,
      string label)

      //Abfragen des Labels
      call        class[SimNet]SimNet.SimObj
                  [SimNet]SimNet.Scheduler::GetCurrentSchedObj()
      stloc       obj
      ldloc       obj
      callvirt    instance string [SimNet]SimNet.SimObj::get_Label()
      stloc       label
      ldloc       label
      ldstr ""
      call        bool [mscorlib]System.String::op_Equality(string,string)
      brfalse     SIM_JUMP_0
      br          IL_0000
      SIM_JUMP_0:   ldloc      label
      ldstr "SIM_WAIT_0"
      call        bool [mscorlib]System.String::op_Equality(string,string)
      brfalse     IL_0066
      br          SIM_WAIT_0

      IL_0000:  ldc.i4.0
      IL_0001:  stloc.0
      IL_0002:  ldc.i4.s   10
      IL_0004:  stloc.1
      IL_0005:  ldarg.3
      IL_0006:  ldc.i4.0
      IL_0007:  ldelem.ref
      IL_0008:  unbox      [mscorlib]System.Double
      IL_000d:  ldind.r8
      IL_000e:  stloc.2
      IL_000f:  ldstr      "Methode Generate 1, SimTime = {0}"
      IL_0014:  call       float64 [SimNet]SimNet.Simulation::SimTime()
      IL_0019:  box        [mscorlib]System.Double
      IL_001e:  call       void [mscorlib]System.Console::WriteLine(string,
                                                                    object)
```

```
.try
{
   //Speichern der lokalen Variablen, Interruptvariable wird speziell markiert
   IL_0023:
   ldc.i4.s 3
   newarr      [mscorlib]System.Object
   stloc       locals
   ldloc    locals
   ldc.i4.s 0
   ldstr "INTERRUPT"
   stelem.ref
   ldloc    locals
   ldc.i4.s 1
   ldloc.s 1
   box         [mscorlib]System.Int32
   stelem.ref
   ldloc    locals
   ldc.i4.s 2
   ldloc.s 2
   box         [mscorlib]System.Double
   stelem.ref
   ldloc      obj
   ldloc      locals
   callvirt   instance void [SimNet]SimNet.SimObj::set_Locals(object[])
   //Setzen des aktuellen Labels zum Wiedereinstieg
   ldloc      obj
   ldstr "SIM_WAIT_0"
   callvirt   instance void [SimNet]SimNet.SimObj::set_Label(string)
   //Speichern der Parameterliste
   ldloc      obj
   ldarg.3
   callvirt   instance void [SimNet]SimNet.SimObj::set_Params(object[])
   ldloc.1
   IL_0024:  conv.r8
   IL_0025:  ldloca.s   interrupt
   IL_0027:  call       void [SimNet]SimNet.Simulation::Wait(float64,
                                                             bool&)
   //Verlassen der Methode
   nop
   ret
   IL_002c:  leave    IL_0031
}
catch [mscorlib]System.Object
{
   IL_002e:  pop
   IL_002f:  leave    IL_0031
}
//Label zum Wiedereinstieg
SIM_WAIT_0:   nop
//Wiederherstellen der lokalen Variabeln, Interrupt gesondert
ldloc      obj
callvirt   instance object[] [SimNet]SimNet.SimObj::get_Locals()
stloc      locals
ldloc.s    obj
callvirt   instance bool [SimNet]SimNet.SimObj::get_Interrupt()
stloc.s    interrupt
ldloc     locals
ldc.i4.s 1
ldelem.ref
unbox    [mscorlib]System.Int32
ldind.i4
stloc.s 1
ldloc     locals
ldc.i4.s 2
ldelem.ref
unbox    [mscorlib]System.Double
ldind.r8
stloc.s 2
```

```
    IL_0031:  ldloc.0
    IL_0032:  brfalse  IL_004a

    IL_0034:  ldstr      "Methode Generate 1 wurde unterbrochen, SimTime = {0}"
    IL_0039:  call       float64 [SimNet]SimNet.Simulation::SimTime()
    IL_003e:  box        [mscorlib]System.Double
    IL_0043:  call       void [mscorlib]System.Console::WriteLine(string,
                                                                  object)
    IL_0048:  br       IL_0066

    IL_004a:  ldarg.0
    IL_004b:  ldftn      instance void
                        ConsoleApplication2.GeneratorObj::Generate2(float64,
                                                                    float64,
                                                                    object[])
    IL_0051:  newobj     instance void [SimNet]SimNet.TellMethod::.ctor(object,
                                                                        native int)
    IL_0056:  ldloc.2
    IL_0057:  ldc.r8     0.0
    IL_0060:  ldnull
    IL_0061:  call       void [SimNet]SimNet.Simulation::Tell(class
                                                    [SimNet]SimNet.TellMethod,
                                                          float64,
                                                          float64,
                                                          object[])
    //Löschen des Objektes
    IL_0066: call       void [SimNet]SimNet.Scheduler::RemoveSchedObj()
    ret
}
```

## 8.3 IL-Stolperfallen – Probleme während der Implementierung

Während der Implementierung traten immer wieder kleinere Probleme auf, die es zu lösen gab. Dabei stand das Erhalten der Benutzerfreundlichkeit im Vordergrund, d.h. es sollten für den Nutzer keine großen Einschränkungen gemacht werden.

Auf zwei Einschränkungen konnte jedoch nicht verzichtet werden. Bei TELL-Methoden müssen benutzerspezifische Parameter in einer Liste übergeben werden und jede WAIT- und WAITFOR-Anweisung muss allein in einem *try/catch*-Block stehen.

Die Einschränkung hinsichtlich der Parameter musste gemacht werden, damit eine Einheitlichkeit bei den TELL-Methoden erreicht werden kann, weil hier ein Delegate verwendet wird und dadurch vorgegeben werden muss, wie dieses

Auszusehen hat. Es ist nicht möglich ein Delegate mit einer unbestimmten Anzahl von Parameter anzulegen. Auch hier müsste vorgeben werden, wenn zwei Parameter vorhanden sind, dass der erste Parameter die Zeit und der zweite die Priorität ist. Außerdem wäre die Auswertung einer unbestimmten Anzahl von Parametern im Enhancer sehr komplex. Somit hat jede TELL-Methode drei Parameter: die Startzeit, die Priorität und eben die nutzerspezifische Parameterliste.

Die Einschränkung bzw. das zusätzliche Verwenden des *try/catch*-Blocks ergab sich bei diversen Tests zur WAIT-Anweisung. Der erste Parameter, der die Zeit angibt, kann ein beliebiger Ausdruck sein, der einen Double-Wert ergibt oder auch ein Funktionsaufruf mit Rückgabewert. Das folgende Listing zeigt einige Beispiele, jeweils in C# und den dazu gehörigen IL-Code.

**Listing 8.4: Wait-Anweisungen in C# und IL**

```
Simulation.Wait(10, ref interrupt);

IL_003a:  ldc.r8     10.
IL_0043:  ldloca.s   interrupt
IL_0045:  call       void [SimNet]SimNet.Simulation::Wait(float64,
                                                          bool&)


Simulation.Wait(x+y,ref interrupt);

IL_004a:  ldloc.1
IL_004b:  ldloc.2
IL_004c:  add
IL_004d:  ldloca.s   interrupt
IL_004f:  call       void [SimNet]SimNet.Simulation::Wait(float64,
                                                          bool&)

Simulation.Wait(10+Simulation.SimTime(), ref interrupt);

IL_0054:  ldc.r8     10.
IL_005d:  call       float64 [SimNet]SimNet.Simulation::SimTime()
IL_0062:  add
IL_0063:  ldloca.s   interrupt
IL_0065:  call       void [SimNet]SimNet.Simulation::Wait(float64,
                                                          bool&)
```

Hier wird schnell klar, dass es fast unmöglich ist, den Punkt zu finden, an dem die Erweiterung eingefügt werden muss. Möglich wäre dies vielleicht, wenn der ersten Parameter analysiert und auseinander genommen würde, dann entsprechend dem Ergebnis der Analyse, an die entsprechende Stelle gesprungen werden würde. Doch diese Analyse wäre sehr aufwendig und komplex, da eben der erste Parameter ein beliebiger Ausdruck sein kann. Auch beim Implementieren der WAITFOR-Anweisung wurde dies bestätigt. Das folgende Listing zeigt ein Beispiel.

**Listing 8.5: WAITFOR-Anweisung in C# und IL**

```
Simulation.WaitFor(new TellMethod(Class1.gen1.Generate2),

                   0.0,0.0,null,ref interrupt);


IL_002a:  ldsfld     class ConsoleApplication2.GeneratorObj
                     ConsoleApplication2.Class1::gen1
IL_002f:  ldftn      instance void
                     ConsoleApplication2.GeneratorObj::Generate2( float64,
                                                                  float64,
                                                                  object[])
IL_0035:  newobj     instance void [SimNet]SimNet.TellMethod::.ctor(object,
                                                                    native int)
IL_003a:  ldc.r8     0.0
IL_0043:  ldc.r8     0.0
IL_004c:  ldnull
IL_004d:  ldloca.s   interrupt
IL_004f:  call       void [SimNet]SimNet.Simulation::WaitFor(class
                                                  [SimNet]SimNet.TellMethod,
                                                             float64,
                                                             float64,
                                                             object[],
                                                             bool&)
```

Die Anzahl der Zeilen, die zurückgesprungen werden muss, um die Erweiterungen einzufügen, ist demnach unbestimmt. Es musste eine Möglichkeit gefunden werden, die Stelle im Code zu markieren, an der die Erweiterungen eingefügt werden müssen. Die Variante, die Anweisung in einen *try/catch*-Block zu packen, hat sich als die Beste erwiesen.

Ein weiteres Problem, welches während der Implementierung aufgetreten ist, zeigte sich beim Verwenden und Testen des Interrupt-Parameters. Die

Auswertung erfolgt über eine *if*-Anweisung, so dass demnach im IL-Code Verzweigungen verwendet werden. Der Compiler berechnet das Offset, welches zwischen den Sprüngen liegt und verwendet bei entsprechend kleinem, die Short-Varianten der *branch*-Opcodes (z.B. *br.s*). Wird nun der IL-Code erweitert, vergrößert sich dementsprechend auch das Offset, so dass evtl. die Short-Variante nicht mehr ausreichend ist und der ILASM-Compiler demzufolge eine Fehlermeldung bringt. Um dieser Fehlerquelle vorzubeugen, wird im Enhancer nach dem Analysieren des IL-Codes die Methode *RemoveShortOffset()* der Klasse *ILFile* ausgeführt. Diese Methode ändert die Short-Varianten der *branch*-Befehle in die normalen um (*br.s* → *br*).

Weiterhin zeigte sich beim Verwenden von *branch*-Anweisungen, dass der Sprung zum Ende der Methode über das Label vor der *return*-Anweisung erfolgt. Da aber vor dem Verlassen das aktuelle Objekt über die Erweiterung aus der Schedulerliste gelöscht werden muss, musste hierzu das Label der *return*-Anweisung vor die Erweiterung gesetzt werden.

Ein ähnlicher Fall zeigte sich bei der Verwendung von WAIT-Anweisungen in Schleifen. Hier erfolgt der Sprung innerhalb der Schleife genau an die erste Stelle des *try*-Blocks, so dass die Erweiterungen nicht berücksichtigt wurden. Das Problem wurde hier gelöst, in dem das Label vor der ersten Anweisung im *try*-Block vor die eingefügte Erweiterung gesetzt wurde.

# 9 Zusammenfassung / Ausblick

Das Ziel dieser Arbeit war die Einbettung von Elementen zur prozessorientierten Simulation in .NET und C#. Das Konzept der Sprache MODSIM III diente hierbei als Vorlage. Die Elemente, welche zur Simulation benötigt werden, wurden nach diesem Konzept entworfen, implementiert und letztlich in einer Bibliothek zur Verfügung gestellt.

Die wichtigste Realisierung war jedoch die Umsetzung ohne Verwendung von Threads nach dem Coroutine-Prinzip. Die Intermediate Language von Microsoft stellt hier einen sehr großen und wichtigen Part dar. Mit Hilfe der IL konnte dieses Prinzip realsiert werden. Hierzu entstand während der Implementierung ein weiteres Programm, der Enhancer, welcher den vom Nutzer erstellten Quellcode auf der Ebene der IL so erweitert, dass Methoden nach dem Coroutine-Prinzip verlassen werden können und bei der Rückkehr mit der Codeausführung an der Stelle fortgesetzt wird, an dem sie Verlassen wurden. Dabei werden vor dem Verlassen der Methode die lokalen Variablen zwischengespeichert und bei Wiedereintritt in die Methode wiederhergestellt.

Die Diplomarbeit hat gezeigt, dass mit der Intermediate Language hier der richtige Ansatz gefunden wurde. Mit der prototypischen Implementierung wurden bereits erste Beispiele erfolgreich modelliert und simuliert (siehe Anhang B).

Aufbauend auf diese Arbeit sollte die Sprache MODSIM III, aber auch andere Simulationssprachen, wie z.B. GPSS/H oder AutoMod, genauer studiert und eine Analyse zu Vor- und Nachteilen der verwendeten Elemente gemacht werden. Entsprechend dieser Studie kann der in dieser Diplomarbeit entstandene Prototyp verbessert und erweitert werden kann.

Weiterhin sollte auch die IL des kommenden .NET Frameworks (Version 2.0) untersucht werden, ob eventuelle Änderungen oder Erweiterungen nützlich sein können.

Außerdem sollte zur Unterstützung des Nutzers, aber vor allem zur Fehlervermeidung, ein Tool implementiert werden, welches den vom Nutzer erstellten Code vor dem Kompilieren überprüft. Diese Verifikation des Codes ist notwendig, da bei der Implementierung gewisse Einschränkungen gemacht werden mussten. So sollte dieses Tool überprüfen, ob jede WAIT- bzw. WAITFOR-Anweisung allein innerhalb eines *try/catch*-Blocks steht. Dieses Tool kann dann ebenfalls in das Projekt über die Buildereignisse, als Präbuildereignis eingebunden werden.

# Abbildungsverzeichnis

# Tabellenverzeichnis



# Quellcodeverzeichnis

# Literaturverzeichnis

## Fachbücher

[B1] Lidin, Serge: ***Inside Microsoft .NET IL Assembler***

Redmond, Washington: Microsoft Press, 2002

[B2] Bock, Jason: ***CIL Programming: Under the Hood™ of .NET***

Berkeley, CA: Apress, 2002

[B3] Liberty, Jesse: ***Programmieren mit C#***, 2. Auflage

Köln, Deutschland: O'Reilly Verlag, 2002

[B4] Troelsen, Andrew: ***C# und die .NET Plattform***

Bonn, Deutschland: mitp-Verlag, 2002

[B5] Smith, Les: ***Writing Add-Ins for Visual Studio .NET***

Berkeley, CA: Apress, 2002

[B6] Ahrens, Klaus; Fischer Joachim:

***Objektorientierte Prozeßsimulation in C++***

Bonn, Deutschland: Addison-Wesley, 1996

[B7] Barneby, Tom; Bock, Jason: ***Applied .NET Attributes***

Berkeley, CA: Apress, 2003

[B8] Nilges, Edward G.: ***Build Your Own .NET Language and Compiler***

Berkeley, CA: Apress, 2004

[B9] Albahari, Ben; Drayton, Peter; Neward, Ted: ***C# in a Nutshell***

Köln, Deutschland: O'Reilly Verlag, 2003

[B10] Troelsen, Andrew: ***C# and the .NET Platform, Second Edition***

Berkeley, CA: Apress, 2003

[B11] Fischer, J. : ***Modellierung und Simulation paralleler diskreter, kontinuierlicher und kombinierter Prozesse in Simula***

Humbolt-Universität Berlin, 1982

**Magazine und Zeitschriften**

[M1] dot.net magazin, Ausgabe 3.2005, Seite 34 – 37

Koen, Peter: ***Verwalteter Code hinter den Kulissen, Teil 1***

http://www.dotnet-magazin.de

[M2] dot.net magazin, Ausgabe 4.2005, Seite 41 – 44

Koen, Peter: ***Verwalteter Code hinter den Kulissen, Teil 2***

http://www.dotnet-magazin.de

[M3] dot.net magazin, Ausgabe 5.2005, Seite 41 – 44

Koen, Peter: ***Verwalteter Code hinter den Kulissen, Teil 3***

http://www.dotnet-magazin.de

[M4] dotnetpro, Ausgabe 6.2004, Seite 118 – 121

Freiberger, Jörg M.: ***Intermediate Language – Zwischenwelt***

http://www.dotnetpro.de

[M5] dotnetpro, Ausgabe 9.2004, Seite 108 – 113

Freiberger, Jörg M.: ***IL-Programming – Assembler .NET***

http://www.dotnetpro.de

[M6] dotnetpro, Ausgabe 10.2004, Seite 104 – 107

Freiberger, Jörg M.: ***Programmieren in IL – Was hat IL, was C# nicht hat?***

http://www.dotnetpro.de

## Dokumentationen

[D1] ***MODSIM III User's Manual***

La Jolla, CA: CACI Products Company, 1996

[D2] ***MODSIM III Tutorial***

La Jolla, CA: CACI Products Company, 1996

[D3] ***MODSIM III Reference Manual***

La Jolla, CA: CACI Products Company, 1996

[D4] ***Common Language Infrastructure (CLI)***

***Partition II: Metadata Definition and Semantics***

ECMA-335 TC39/TG3, October 2002

http://msdn.microsoft.com/netframework/ecma/

[D5] ***Common Language Infrastructure (CLI)***

***Partition III: CIL Instruction Set***

ECMA-335 TC39/TG3, October 2002

http://msdn.microsoft.com/netframework/ecma/

## Online-Artikel

[O1] ***Erweitern Sie Ihre Assemblies nach dem Kompilieren***

Matytschak, Mirko

msdn Deutschland, 2004

http://www.microsoft.com/germany/msdn/library/net/visualstudio/ErweiternSieIhreAssembliesnachdemKompilieren.mspx

[O2] ***The Code Project – Introduction to IL Assembly Language***

http://www.codeproject.com/dotnet/ilassembly.asp

[O3] ***Die .NET Common Language Runtime***

Willers, Michael

msdn Deutschland, 2004

http://www.microsoft.com/germany/msdn/library/net/DieNetCommonLanguageRuntim.mspx

[O4] ***ILDASM ist Ihr bester Freund***

Robbins, John

msdn Deutschland, 2004

http://www.microsoft.com/germany/msdn/library/net/ILDASMIstIhrBesterFreund.mspx

[O5] ***Build-Events in Visual C# nutzen***

Schiffer, Mathias

msdn, Deutschland 2004

http://www.microsoft.com/germany/msdn/library/net/csharp/BuildEventsInVisualCSharpNutzen.mspx

[O6] ***Round-Tripping a simple .NET-Exe***

http://dotnetjunkies.com/WebLog/jhaley/archive/2004/06/06/15596.aspx

## Webseiten

| | |
|---|---|
| ***Wikipedia*** | http://de.wikipedia.org |
| ***The Code Project*** | http://www.codeproject.com |
| ***DOTNETFramework.de*** | http://www.dotnetframework.de |
| ***codeguru™*** | http://www.codeguru.com |
| ***msdn*** | http://msdn.microsoft.com |
| ***Google™ Groups*** | http://groups.google.de |
| Gruppen: | microsoft.public.dotnet.csharp |
| | microsoft.public.dotnet.framework |
| | microsoft.public.dotnet.general |
| | microsoft.public.dotnet.languages.csharp |
| | microsoft.public.dotnet.languages.vc |

# Anhang A - Benutzerdokumentation

Einbinden der Bibliothek in das Projekt:

- Im Projektmappen-Explorer unter Verweise einen neuen Verweis hinzufügen (Rechtsklick auf Verweise → Verweis hinzufügen).
- Im Dialog „Verweis hinzufügen“, über Durchsuchen die SimNet.dll in das Projekt einbinden.
- Nun noch über *using* angeben, das Typen aus diesem Namensraum verwenden werden sollen.

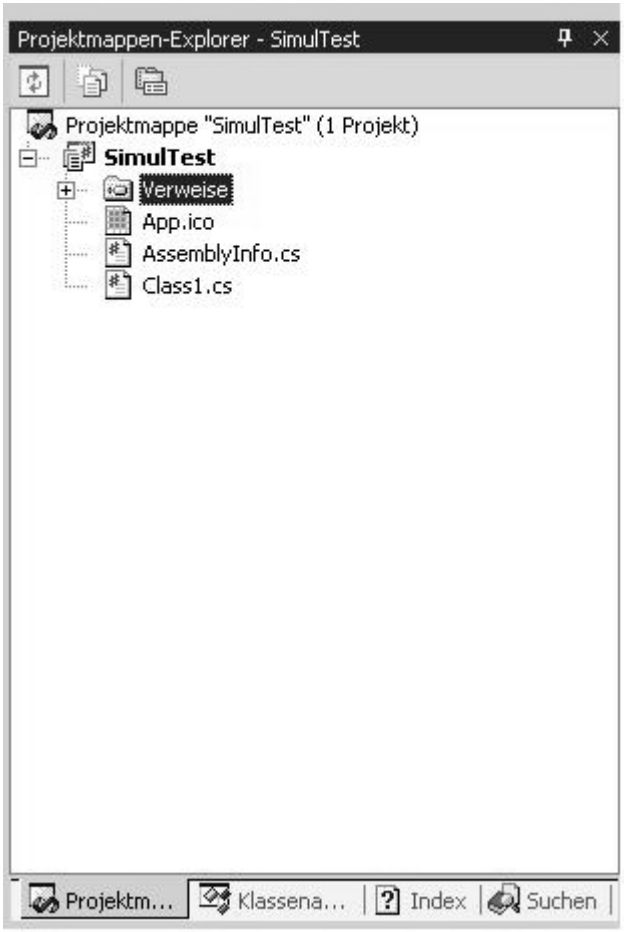


**Abbildung A.1: Projektmappen-Explorer**

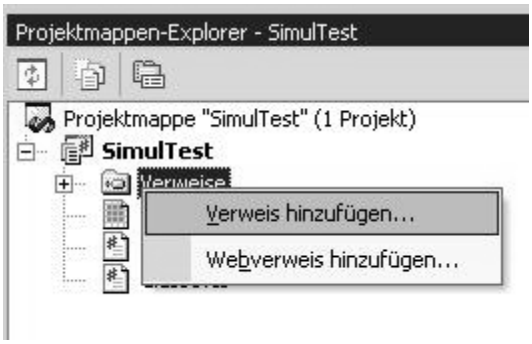


**Abbildung A.2: Verweis hinzufügen**

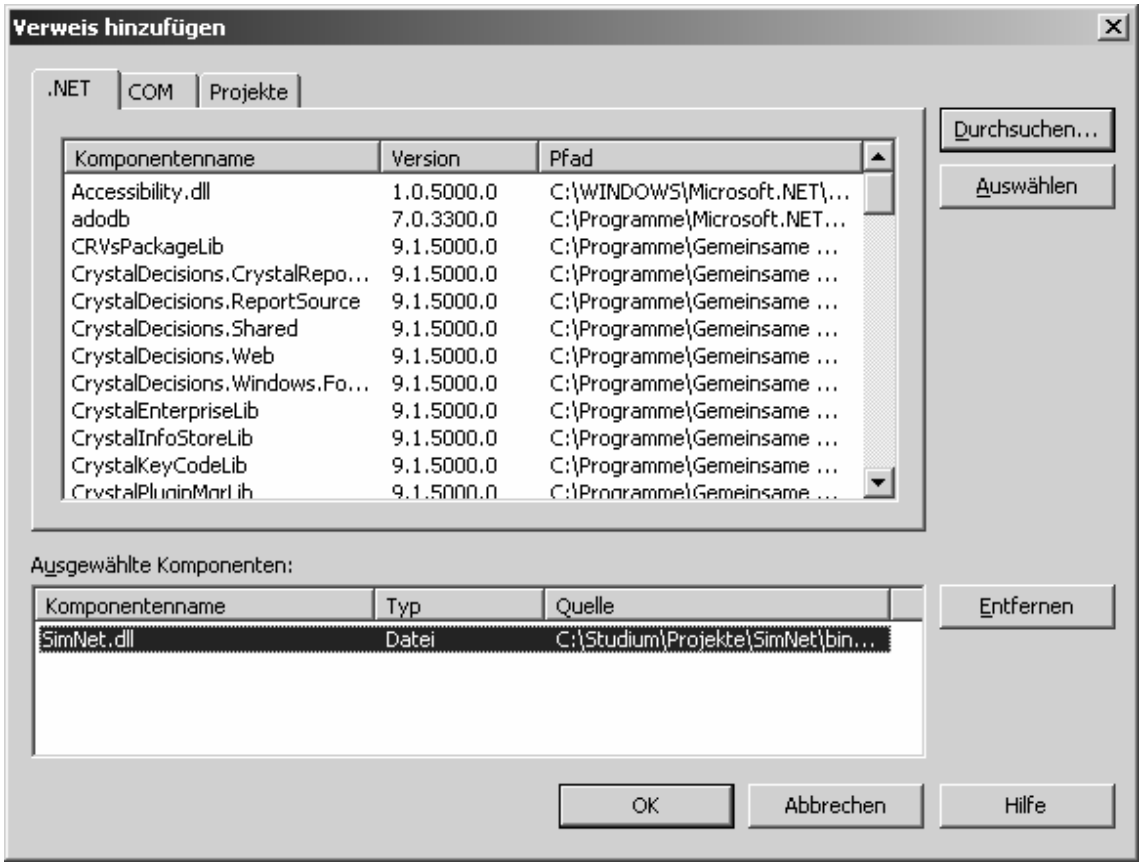


**Abbildung A.3: Verweis zu SimNet.dll hinzufügen**

Die Funktionen der SimNet-Bibliothek stehen nun zur Verfügung. Nun muss noch der Enhancer in das Projekt eingebunden werden:

- Im Projektmappen-Explorer den Eigenschaftendialog des Projektes öffnen (Rechtsklick auf das Projekt → Eigenschaften)
- Nun unter allgemeine Eigenschaften, Buildereignisse, das Postbuildereignis eintragen. Hier müssen der Pfad zum Enhancer und die notwendigen Argumente eingetragen werden.
- Der Enhancer selber hat nur ein Argument, welches mit gegeben werden muss – das erstellte Assembly. Hierzu können Makros verwendet werden, z.B. *$(TargetFileName)* für das erstellte Assembly.

  Beispielpfad: C:\Studium\Enhancer\Enhancer.exe $(TargetFileName)

- Nun noch einstellen wenn das Buildereignis ausgeführt werden soll: Bei erfolgreichem Erstellen

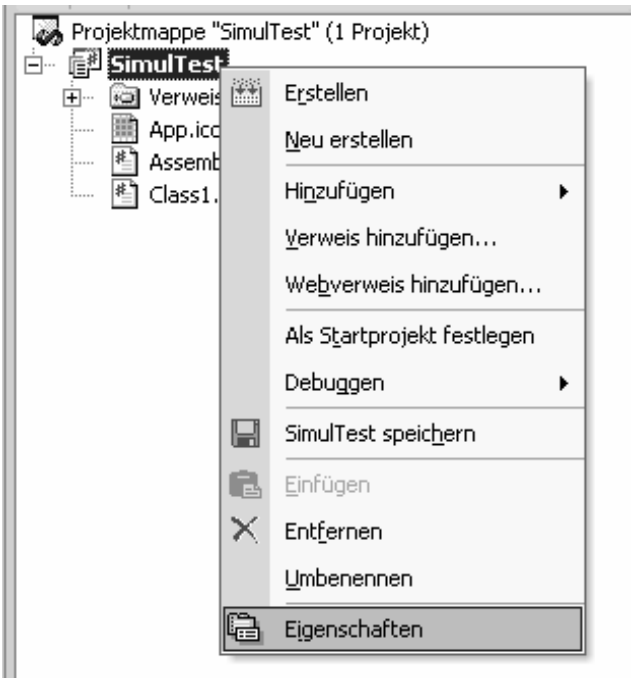


**Abbildung A.4: Eigenschaften des Projektes öffnen**

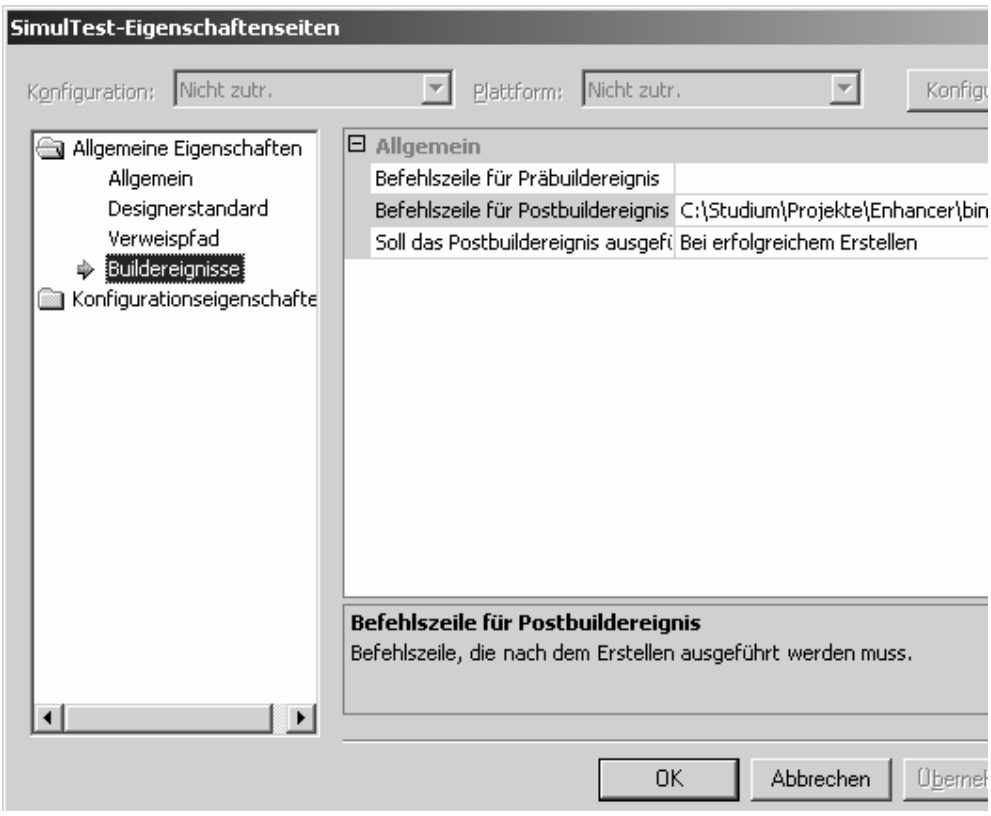


**Abbildung A.5: Buildereignis setzen**

# Anhang B - Test der Implementierung

Zum Test der bisherigen Implementierung wurden zwei Beispiele modelliert.

Nachfolgend sind die Aufgabenstellungen der Beispiele beschreiben.

Beispiel 1 – Modellierung und Simulation von Telefonzellen

Anrufer treffen **gleich verteilt** zwischen **0.5 und 10.0 Minuten** vor **6** nebeneinander stehenden Telefonzellen ein. Der Anrufer wählt diejenige Telefonzelle mit der kürzesten Schlange wartender Anrufer aus. Bei mehreren Schlangen mit gleichlanger kürzester Länge bzw. wenn vor mehreren Zellen keine Warteschlange existiert, dann soll die Zelle mit dem kleinsten Index ausgewählt werden. Die Telefonierdauer soll **gleich verteilt** zwischen **5.5 und 30.0 Minuten** dauern. Nach **500 Minuten** soll kein Anrufer mehr eintreffen. Bei jedem Eintreffen eines Anrufers soll die Anzahl der freien Zellen und die Anzahl der aktuell wartenden Anrufer auf dem Monitor ausgegeben werden. Des Weiteren soll die Nummer der gewählten Zelle bzw. Warteschlange mit aktueller neuer Länge und die aktuelle Uhrzeit ausgegeben werden. Jede Veränderung in Belegung und Freigabe einer Zelle soll ebenfalls über die Ausgabe der Uhrzeit protokolliert werden. Am Ende soll eine Ausgabe der statistischen Werte der Ressourcen erfolgen.

Beispiel 2 – Modellierung und Simulation eines Großmarktes

Ein Großmarkt soll für einen Tag von Beginn der Marktöffnung bis zum Schließen des Marktes modelliert und simuliert werden. Folgende Daten sind gegeben, wobei eine variable und robuste Eingabe der Daten über Tastatur wegen der erreichbaren Allgemeinheit zu bevorzugen ist:

Öffnungszeitpunkt: **8.00 Uhr**

Schließzeitpunkt: **20.00 Uhr**

Zwischenankunftszeit der Kunden: **Exponential** verteilt mit **Mittelwert 3.0 Min.**

Anzahl der Kassen in der Kaufhalle: **5**

Anzahl der Einkaufswagen: **200**

Einkaufszeit: **Dreiecksverteilt** mit **Minimalwert 2.0**, **Mittelwert 12.0** und **Maximalwert 25.0 Minuten**

Dauer des Kassierens: **Gleichverteilung** zwischen **0.4 und 4.4 Minuten**

Ankommende Kunden warten, bis ein Einkaufswagen frei ist. (Die Einkaufswagen entsprechen der Kapazität der Kaufhalle.) Beim Kassieren wählen die Kunden die kürzeste Schlange, bei gleichlangen Schlangen soll eine Kasse über eine **Gleichverteilung** ausgewählt werden. Nach dem Kassieren geben die Kunden den Einkaufswagen wieder frei und verlassen den Großmarkt. Die Ankunftszeit an der Kaufhalle, der Zeitpunkt des Erhaltens eines Korbes, der Zeitpunkt des Erreichens der Kasse mit ausgewählter Kassennummer und das Verlassen der Kaufhalle sind auszugeben.

Am Ende soll eine Ausgabe der statistischen Werte der Ressourcen erfolgen.

Die Umsetzung der zwei Beispiele hat gezeigt, dass mit der bisherigen Implementierung bereits eine Simulation von kleineren Modellen möglich ist. Mit Hilfe der Ressourcen und der dazugehörigen Statistiken (wie z.B. Auslastung oder Verweilzeit) ist auch eine Analyse und Auswertung der Simulation möglich.

Die Implementierung beide Beispiele, sowie die Ausgabe eines Simulationslaufs, befinden sich auf der beiliegenden CD.

# Anhang C - Verteilungsfunktionen

| Verteilungs-funktion | Funktion in RandomObj | Bedeutung von a | Bedeutung von b | Formel für die Wahrschein-lichkeitsverteilung | Transformation |
|---|---|---|---|---|---|
| Normal-verteilung | Normal(double a, double b) | $E(x) = \mu$ (Erwartungswert) | $\sqrt{D^2(x)} = \sigma$ (Standard-abweichung) | $F(x;\mu,\sigma) = \int_{-\infty}^{x} \frac{1}{\sigma\sqrt{2\pi}} e^{\left[\frac{(z-u)^2}{2\sigma^2}\right]} dz$ | $y = \left(\sum_{i=1}^{12} x_i - 6\right) b + a$ |
| Exponential-verteilung | Exponential(double a) | $E(x) = \frac{1}{\lambda}$ (Erwartungswert) | - | $F(x;\lambda) = 1 - e^{-\lambda x}$ | $y = -a \cdot \ln x_i$ |
| Weibull-verteilung | Weibull(double a, double b) | $\lambda$ (Skalenparameter) | $\alpha$ (Formparameter) | $F(x;\lambda,\alpha) = 1 - e^{\left[-\left(\frac{x}{\lambda}\right)^{\alpha}\right]}$ | $y = a(-\ln x_i)^{\frac{1}{b}}$ |
| Erlang-verteilung | Erlang(double a, int b) | $E(x) = \frac{k}{\lambda}$ (Erwartungswert) | $k$ (Anzahl Phasen) | $F(x,k,\lambda) = 1 - e^{-\lambda x} \sum_{n=0}^{k-1} \frac{(\lambda k x)^n}{n!}$ | $y = -\frac{a}{b} \sum_{i=1}^{b} \ln x_i$ |
| Log-Normal-verteilung | LgNormal(double a, double b) | $E(x) = e^{\mu + \frac{\sigma^2}{2}}$ (Erwartungswert) | $e^{\sigma}$ (geometrische Dispersion) | $F(x;\mu,\sigma) = \int_{0}^{x} \frac{1}{z\sigma\sqrt{2\pi}} e^{\left[\frac{(\ln z - u)^2}{2\sigma^2}\right]} dz$ | $y = a \cdot b^{\left(^{nu} - \frac{\ln b}{2}\right)}$ |
| Gleich-verteilung | UniformInt(int min, int max)<br>UniformDouble(double min, double max) | min (a) (untere Grenze) | max (b) (obere Grenze) | $F(x;a,b) = \frac{1}{b-a} \int_{a}^{x} dz \; a \le x \le b$ | bei Gleitkommawerten:<br>$y = a + x_i \cdot (b-a)$<br>bei Festkommawerten:<br>$y = a + x_i \cdot (b-a+1)$ |
| Dreiecks-verteilung | Triangular(double min, double mode, double max)<br>mode c mit a < c < b<br>(Wert mit max. Häufigkeit) | min (a) (untere Grenze) | max (b) (obere Grenze) | F(x,a,b,c)=(x-a)²/((c-a)*(b-a))<br>*für a <= x <= c*<br>F(x,a,b,c)=(((c-a)*(b-c))+(2*x*b-x²-2*b*c+c²))/((b-c)*(b-a))<br>*für c <= x <= b* | y= a+(x_i*(c-a)*(b-a))^1/2<br>*für ((x_i>=0) und (x_i<=(c-a)/(b-a)))*<br>y=b-(((b-c)*(b-a)*(1-x_i))^1/2<br>*für ((x_i>(c-a)/(b-a)) und (x_i<=1.0))* |

$x_i$ gleichverteilte Zufallszahl im Intervall [0,1], entspricht random.NextDouble() $x\%_i$ normalverteilte Zufallszahl mit $\mu = 0$ und $\sigma = 1$, entspricht normal (0,1)

$y$ transformierte Zufallszahl

# Anhang D - IL Grammatikreferenz

## Namespace und Klassendeklarationen

```
<nameSpaceHead> ::= .namespace <compName>

<classHead> ::= .class <classAttrs> <id> <extendsClause>
                       <implClause>

<classAttrs> ::= /* EMPTY */ | <classAttrs> <classAttr>

<classAttr> ::= <classAttr> public
             | <classAttr> private
             | <classAttr> nested public
             | <classAttr> nested private
             | <classAttr> nested family
             | <classAttr> nested assembly
             | <classAttr> nested famandassem
             | <classAttr> nested famorassem
             | <classAttr> value
             | <classAttr> enum
             | <classAttr> interface
             | <classAttr> sealed
             | <classAttr> abstract
             | <classAttr> auto
             | <classAttr> sequential
             | <classAttr> explicit
             | <classAttr> ansi
             | <classAttr> unicode
             | <classAttr> autochar
             | <classAttr> import
             | <classAttr> serializable
             | <classAttr> beforefieldinit
             | <classAttr> specialname
             | <classAttr> rtspecialname
<extendsClause> ::= /* EMPTY */ | extends <classRef>
```

```
<implClause> ::= /* EMPTY */ | implements <classRefs>

<classRefs> ::= <classRefs>,<classRef> | <classRef>

<classRef> ::= [ <compName> ] <slashedName>
           | [.module <compName> ] <slashedName>
           | <slashedName>

<slashedName> ::= <compName>
              | <slashedName>/<compName>

<classDecls> ::= /* EMPTY */ | <classDecls> <classDecl>

<classDecl> ::= <methodHead>  <methodDecls> }
            | <classHead> { <classDecls> }
            | <eventHead> { <eventDecls> }
            | <propHead> { <propDecls> }
            | <fieldDecl>
            | <dataDecl>
            | <secDecl>
            | <extSourceSpec>
            | <customAttrDecl>
            | .size <int32>
            | .pack <int32>
            | .override <typeSpec>::<methodName>
                 with <callConv> <type> <typeSpec>::<methodName>(
                                                      <sigArgs>)
            | <languageDecl>
```

## Typendeklaration – Signature Types

```
<type> ::= class <classRef>
       | object
       | string
       | value class <classRef>
       | valuetype <classRef>
       | <type> [ ]
       | <type> [ <bounds> ]
       | <type> &
       | <type> *
       | <type> pinned
       | <type> modreq(<classRef>)
       | <type> modopt(<classRef>)
       | method <callConv> <type>*(<sigArgs>)
       | typedref
       | char
       | void
       | bool
       | int8
       | int16
       | int32
       | int64
       | float32
       | float64
       | unsigned int8
       | unsigned int16
       | unsigned int32
       | unsigned int64
       | native int
       | native unsigned int

<bounds> ::= <bound>
         | <bounds>,<bound>
<bound> ::= /* EMPTY */
        | ...
        | <int32>
        | <int32> ... <int32>
        | <int32> ...
```

```
<callConv> ::= instance <callConv>
             | explicit <callConv>
             | <callKind>

<callKind> ::= /* EMPTY */
             | default
             | vararg
             | unmanaged cdecl
             | unmanaged stdcall
             | unmanaged thiscall
             | unmanaged fastcall
```

## Typendeklaration – Native Types

```
<type> ::= class <classRef>
         | object
         | string
         | value class <classRef>
         | valuetype <classRef>
         | <type> [ ]
         | <type> [ <bounds> ]
         | <type> &
         | <type> *
         | <type> pinned
         | <type> modreq(<classRef>)
         | <type> modopt(<classRef>)
         | method <callConv> <type>*(<sigArgs>)
         | typedref
         | char
         | void
         | bool
         | int8
         | int16
         | int32
         | int64
         | float32
         | float64
         | unsigned int8
         | unsigned int16
```

```
        | unsigned int32
        | unsigned int64
        | native int
        | native unsigned int

<bounds> ::= <bound>
          | <bounds>,<bound>

<bound> ::= /* EMPTY */
         | ...
         | <int32>
         | <int32> ... <int32>
         | <int32> ...

<callConv> ::= instance <callConv>
            | explicit <callConv>
            | <callKind>

<callKind> ::= /* EMPTY */
            | default
            | vararg
            | unmanaged cdecl
            | unmanaged stdcall
            | unmanaged thiscall
            | unmanaged fastcall
```

## Deklaration des Headers einer Methode

```
<methodHead> ::= .method <methAttr> <callConv> <paramAttr> <type>
                     <methodName>(<sigArgs>) <implAttr> {
               | .method <methAttr> <callConv> <paramAttr> <type>
                     marshal(<nativeType>)
                         <methodName>(<sigArgs>) <implAttr> {

<methAttr> ::= /* EMPTY */
             | <methAttr> static
             | <methAttr> public
             | <methAttr> private
             | <methAttr> family
             | <methAttr> assembly
             | <methAttr> famandassem
             | <methAttr> famorassem
             | <methAttr> privatescope
             | <methAttr> final
             | <methAttr> virtual
             | <methAttr> abstract
             | <methAttr> hidebysig
             | <methAttr> newslot
             | <methAttr> reqsecobj
             | <methAttr> specialname
             | <methAttr> rtspecialname
             | <methAttr> unmanagedexp
             | <methAttr> pinvokeimpl(<compQstring>
                                      as <compQstring> <pinvAttr>)
             | <methAttr> pinvokeimpl(<compQstring> <pinvAttr>)
             | <methAttr> pinvokeimpl(<pinvAttr>)

<pinvAttr> ::= /* EMPTY */
             | <pinvAttr> nomangle
             | <pinvAttr> ansi
             | <pinvAttr> unicode
             | <pinvAttr> autochar
             | <pinvAttr> lasterr
             | <pinvAttr> winapi
             | <pinvAttr> cdecl
```

```
            | <pinvAttr> stdcall
            | <pinvAttr> thiscall
            | <pinvAttr> fastcall

<methodName> ::= .ctor
              | .cctor
              | <compName>

<paramAttr> ::= /* EMPTY */
             | <paramAttr> [in]
             | <paramAttr> [out]
             | <paramAttr> [opt]

<implAttr> ::= /* EMPTY */
            | <implAttr> native
            | <implAttr> cil
            | <implAttr> optil
            | <implAttr> managed
            | <implAttr> unmanaged
            | <implAttr> forwardref
            | <implAttr> preservesig
            | <implAttr> runtime
            | <implAttr> internalcall
            | <implAttr> synchronized
            | <implAttr> noinlining

<sigArgs> ::= /* EMPTY */
           | <sigArgList>

<sigArgList> ::= <sigArg>
              | <sigArgList>,<sigArg>

<sigArg> ::= ...
          | <paramAttr> <type>
          | <paramAttr> <type> <id>
          | <paramAttr> <type> marshal( <nativeType> )
          | <paramAttr> <type> marshal( <nativeType> ) <id>
```

## Deklaration eines Methodenrumpfes

```
<methodHead> ::= .method <methAttr> <callConv> <paramAttr> <type>
                     <methodName>(<sigArgs>) <implAttr> {
               | .method <methAttr> <callConv> <paramAttr> <type>
                     marshal(<nativeType>)
                         <methodName>(<sigArgs>) <implAttr> {

<methAttr> ::= /* EMPTY */
             | <methAttr> static
             | <methAttr> public
             | <methAttr> private
             | <methAttr> family
             | <methAttr> assembly
             | <methAttr> famandassem
             | <methAttr> famorassem
             | <methAttr> privatescope
             | <methAttr> final
             | <methAttr> virtual
             | <methAttr> abstract
             | <methAttr> hidebysig
             | <methAttr> newslot
             | <methAttr> reqsecobj
             | <methAttr> specialname
             | <methAttr> rtspecialname
             | <methAttr> unmanagedexp
             | <methAttr> pinvokeimpl(<compQstring>
                                      as <compQstring> <pinvAttr>)
             | <methAttr> pinvokeimpl(<compQstring> <pinvAttr>)
             | <methAttr> pinvokeimpl(<pinvAttr>)

<pinvAttr> ::= /* EMPTY */
             | <pinvAttr> nomangle
             | <pinvAttr> ansi
             | <pinvAttr> unicode
             | <pinvAttr> autochar
             | <pinvAttr> lasterr
             | <pinvAttr> winapi
             | <pinvAttr> cdecl
```

```
            | <pinvAttr> stdcall
            | <pinvAttr> thiscall
            | <pinvAttr> fastcall

<methodName> ::= .ctor
              | .cctor
              | <compName>

<paramAttr> ::= /* EMPTY */
             | <paramAttr> [in]
             | <paramAttr> [out]
             | <paramAttr> [opt]

<implAttr> ::= /* EMPTY */
            | <implAttr> native
            | <implAttr> cil
            | <implAttr> optil
            | <implAttr> managed
            | <implAttr> unmanaged
            | <implAttr> forwardref
            | <implAttr> preservesig
            | <implAttr> runtime
            | <implAttr> internalcall
            | <implAttr> synchronized
            | <implAttr> noinlining

<sigArgs> ::= /* EMPTY */
           | <sigArgList>

<sigArgList> ::= <sigArg>
              | <sigArgList>,<sigArg>

<sigArg> ::= ...
          | <paramAttr> <type>
          | <paramAttr> <type> <id>
          | <paramAttr> <type> marshal( <nativeType> )
          | <paramAttr> <type> marshal( <nativeType> ) <id>
```

## Deklaration von Feldern

```
<fieldDecl> ::= .field <repeatOpt> <fieldAttr> <type> <id>
                       <atOpt> <initOpt>

<repeatOpt> ::= /* EMPTY */
              | [<int32>]
<fieldAttr> ::= /* EMPTY */
              | <fieldAttr> public
              | <fieldAttr> private
              | <fieldAttr> family
              | <fieldAttr> assembly
              | <fieldAttr> famandassem
              | <fieldAttr> famorassem
              | <fieldAttr> privatescope
              | <fieldAttr> static
              | <fieldAttr> initonly
              | <fieldAttr> rtspecialname
              | <fieldAttr> specialname
              | <fieldAttr> marshal( <nativeType> )
              | <fieldAttr> literal
              | <fieldAttr> notserialized
<atOpt>     ::= /* EMPTY */
              | at <id>
<initOpt>   ::= /* EMPTY */
              | = <fieldInit>
<fieldInit> ::= float32(<float64>)
              | float64(<float64>)
              | float32(<int64>)
              | float64(<int64>)
              | int64(<int64>)
              | int32(<int64>)
              | int16(<int64>)
              | char(<int64>)
              | int8(<int64>)
              | bool(<truefalse>)
              | <compQstring>
              | bytearray( <bytes> )
              | nullref
```

# Anhang E - IL Instruktionen Referenz

| Instruction Parameter Types | |
|---|---|
| ***Type*** | ***Description*** |
| int8 | Signed 1-byte integer |
| uint8 | Unsigned 1-byte integer |
| int32 | Signed 4-byte integer |
| uint32 | Unsigned 4-byte integer |
| int64 | Signed 8-byte integer |
| float32 | 4-byte floating point number |
| float64 | 8-byte floating point number |
| <Method> | MethodDef or MemberRef token |
| <Field> | FieldDef or MemberRef token |
| <Type> | TypeDef, TypeRef, or TypeSpec token |
| <Signature> | StandAloneSig token |
| <String> | User-defined string token |

| Evaluation Stack Types | |
|---|---|
| ***Type*** | ***Description*** |
| int32 | Signed 4-byte integer |
| int64 | Signed 8-byte integer |
| Float | 80-bit floating point number |
| & | Managed or unmanaged pointer |
| o | Object reference |
| * | Unspecified type |

| IL Instructions | | | | |
|---|---|---|---|---|
| Opcode | Name | Parameter(s) | Pop | Push |
| 00 | nop | - | - | - |
| 01 | break | - | - | - |
| 02 | ldarg.0 | - | - | * |
| 03 | ldarg.1 | - | - | * |
| 04 | ldarg.2 | - | - | * |
| 05 | ldarg.3 | - | - | * |
| 06 | ldloc.0 | - | - | * |
| 07 | ldloc.1 | - | - | * |
| 08 | ldloc.2 | - | - | * |
| 09 | ldloc.3 | - | - | * |
| 0A | stloc.0 | - | * | - |
| 0B | stloc.1 | - | * | - |
| 0C | stloc.2 | - | * | - |
| 0D | stloc.3 | - | * | - |
| 0E | ldarg.s | uint8 | - | * |
| 0F | ldarga.s | uint8 | - | & |
| 10 | starg.s | uint8 | * | - |
| 11 | ldloc.s | uint8 | - | * |
| 12 | ldloca.s | uint8 | - | & |
| 13 | stloc.s | uint8 | * | - |
| 14 | ldnull | - | - | &=0 |
| 15 | ldc.i4.m1ldc.i4.M1 | - | - | int32=-1 |
| 16 | ldc.i4.0 | - | - | int32=0 |
| 17 | ldc.i4.1 | - | - | int32=1 |

| IL Instructions | | | | |
|---|---|---|---|---|
| Opcode | Name | Parameter(s) | Pop | Push |
| 18 | ldc.i4.2 | - | - | int32=2 |
| 19 | ldc.i4.3 | - | - | int32=3 |
| 1A | ldc.i4.4 | - | - | int32=4 |
| 1B | ldc.i4.5 | - | - | int32=5 |
| 1C | ldc.i4.6 | - | - | int32=6 |
| 1D | ldc.i4.7 | - | - | int32=7 |
| 1E | ldc.i4.8 | - | - | int32=8 |
| 1F | ldc.i4.s | int8 | - | int32 |
| 20 | ldc.i4 | int32 | - | int32 |
| 21 | ldc.i8 | int64 | - | int64 |
| 22 | ldc.r4 | float32 | - | Float |
| 23 | ldc.r8 | float64 | - | Float |
| 25 | dup | - | * | *,* |
| 26 | pop | - | * | - |
| 27 | jmp | <Method> | - | - |
| 28 | call | <Method> | N arguments | Ret.value |
| 29 | calli | <Signature> | N arguments | Ret.value |
| 2A | ret | - | * | - |
| 2B | br.s | int8 | - | - |
| 2C | brfalse.sbrnull.sbrzero.s | int8 | int32 | - |
| 2D | brtrue.sbrinst.s | int8 | int32 | - |
| 2E | beq.s | int8 | *,* | - |
| 2F | bge.s | int8 | *,* | - |
| 30 | bgt.s | int8 | *,* | - |

| IL Instructions | | | | |
|---|---|---|---|---|
| Opcode | Name | Parameter(s) | Pop | Push |
| 31 | ble.s | int8 | *,* | - |
| 32 | blt.s | int8 | *,* | - |
| 33 | bne.un.s | int8 | *,* | - |
| 34 | bge.un.s | int8 | *,* | - |
| 35 | bgt.un.s | int8 | *,* | - |
| 36 | ble.un.s | int8 | *,* | - |
| 37 | blt.un.s | int8 | *,* | - |
| 38 | br | int32 | - | - |
| 39 | brfalsebrnullbrzero | int32 | int32 | - |
| 3A | brtruebrinst | int32 | int32 | - |
| 3B | beq | int32 | *,* | - |
| 3C | bge | int32 | *,* | - |
| 3D | bgt | int32 | *,* | - |
| 3E | ble | int32 | *,* | - |
| 3F | blt | int32 | *,* | - |
| 40 | bne.un | int32 | *,* | - |
| 41 | bge.un | int32 | *,* | - |
| 42 | bgt.un | int32 | *,* | - |
| 43 | ble.un | int32 | *,* | - |
| 44 | blt.un | int32 | *,* | - |
| 45 | switch | (uint32=N) + N(int32) | *,* | - |
| 46 | ldind.i1 | - | & | int32 |
| 47 | ldind.u1 | - | & | int32 |
| 48 | ldind.i2 | - | & | int32 |

| IL Instructions | | | | |
|---|---|---|---|---|
| Opcode | Name | Parameter(s) | Pop | Push |
| 49 | ldind.u2 | - | & | int32 |
| 4A | ldind.i4 | - | & | int32 |
| 4B | ldind.u4 | - | & | int32 |
| 4C | ldind.i8ldind.u8 | - | & | int64 |
| 4D | ldind.i | - | & | int32 |
| 4E | ldind.r4 | - | & | Float |
| 4F | ldind.r8 | - | & | Float |
| 50 | ldind.ref | - | & | & |
| 51 | stind.ref | - | &,& | - |
| 52 | stind.i1 | - | int32,& | - |
| 53 | stind.i2 | - | int32,& | - |
| 54 | stind.i4 | - | int32,& | - |
| 55 | stind.i8 | - | int32,& | - |
| 56 | stind.r4 | - | Float,& | - |
| 57 | stind.r8 | - | Float,& | - |
| 58 | add | - | *,* | * |
| 59 | sub | - | *,* | * |
| 5A | mul | - | *,* | * |
| 5B | div | - | *,* | * |
| 5C | div.un | - | *,* | * |
| 5D | rem | - | *,* | * |
| 5E | rem.un | - | *,* | * |
| 5F | and | - | *,* | * |
| 60 | or | - | *,* | * |

| IL Instructions | | | | |
|---|---|---|---|---|
| Opcode | Name | Parameter(s) | Pop | Push |
| 61 | xor | - | *,* | * |
| 62 | shl | - | *,* | * |
| 63 | shr | - | *,* | * |
| 64 | shr.un | - | *,* | * |
| 65 | neg | - | * | * |
| 66 | not | - | * | * |
| 67 | conv.i1 | - | * | int32 |
| 68 | conv.i2 | - | * | int32 |
| 69 | conv.i4 | - | * | int32 |
| 6A | conv.i8 | - | * | int64 |
| 6B | conv.r4 | - | * | Float |
| 6C | conv.r8 | - | * | Float |
| 6D | conv.u4 | - | * | int32 |
| 6E | conv.u8 | - | * | int64 |
| 6F | callvirt | <Method> | N arguments | Ret.value |
| 70 | cpobj | <Type> | &,& | - |
| 71 | ldobj | <Type> | & | * |
| 72 | ldstr | <String> | - | o |
| 73 | newobj | <Method> | N arguments | o |
| 74 | castclass | <Type> | o | o |
| 75 | isinst | <Type> | o | int32 |
| 76 | conv.r.un | - | * | Float |
| 79 | unbox | <Type> | o | & |
| 7A | throw | - | o | - |

| IL Instructions | | | | |
|---|---|---|---|---|
| Opcode | Name | Parameter(s) | Pop | Push |
| 7B | ldfld | <Field> | o/& | * |
| 7C | ldflda | <Field> | o/& | & |
| 7D | stfld | <Field> | o/&,* | - |
| 7E | ldsfld | <Field> | - | * |
| 7F | ldsflda | <Field> | - | & |
| 80 | stsfld | <Field> | * | - |
| 81 | stobj | <Type> | &,* | - |
| 82 | conv.ovf.i1.un | - | * | int32 |
| 83 | conv.ovf.i2.un | - | * | int32 |
| 84 | conv.ovf.i4.un | - | * | int32 |
| 85 | conv.ovf.i8.un | - | * | int64 |
| 86 | conv.ovf.u1.un | - | * | int32 |
| 87 | conv.ovf.u2.un | - | * | int32 |
| 88 | conv.ovf.u4.un | - | * | int32 |
| 89 | conv.ovf.u8.un | - | * | int64 |
| 8A | conv.ovf.i.un | - | * | int32 |
| 8B | conv.ovf.u.un | - | * | int64 |
| 8C | box | <Type> | * | o |
| 8D | newarr | <Type> | int32 | o |
| 8E | ldlen | - | o | int32 |
| 8F | ldelema | <Type> | int32,o | & |
| 90 | ldelem.i1 | - | int32,o | int32 |
| 91 | ldelem.u1 | - | int32,o | int32 |
| 92 | ldelem.i2 | - | int32,o | int32 |

| IL Instructions | | | | |
|---|---|---|---|---|
| Opcode | Name | Parameter(s) | Pop | Push |
| 93 | ldelem.u2 | - | int32,o | int32 |
| 94 | ldelem.i4 | - | int32,o | int32 |
| 95 | ldelem.u4 | - | int32,o | int32 |
| 96 | ldelem.i8ldelem.u8 | - | int32,o | int64 |
| 97 | ldelem.i | - | int32,o | int32 |
| 98 | ldelem.r4 | - | int32,o | Float |
| 99 | ldelem.r8 | - | int32,o | Float |
| 9A | ldelem.ref | - | int32,o | o/& |
| 9B | stelem.i | - | int32,int32,o | - |
| 9C | stelem.i1 | - | int32,int32,o | - |
| 9D | stelem.i2 | - | int32,int32,o | - |
| 9E | stelem.i4 | - | int32,int32,o | - |
| 9F | stelem.i8 | - | int64,int32,o | - |
| A0 | stelem.r4 | - | Float,int32,o | - |
| A1 | stelem.r8 | - | Float,int32,o | - |
| A2 | stelem.ref | - | o/&,int32,o | - |
| B3 | conv.ovf.i1 | - | * | int32 |
| B4 | conv.ovf.u1 | - | * | int32 |
| B5 | conv.ovf.i2 | - | * | int32 |
| B6 | conv.ovf.u2 | - | * | int32 |
| B7 | conv.ovf.i4 | - | * | int32 |
| B8 | conv.ovf.u4 | - | * | int32 |
| B9 | conv.ovf.i8 | - | * | int64 |
| BA | conv.ovf.u8 | - | * | int64 |

| IL Instructions | | | | |
|---|---|---|---|---|
| Opcode | Name | Parameter(s) | Pop | Push |
| C2 | refanyval | <Type> | * | & |
| C3 | ckfinite | - | * | Float |
| C6 | mkrefany | <Type> | & | * |
| D0 | ldtoken | <Type>/ <Field>/<Method> | - | & |
| D1 | conv.u2 | - | * | int32 |
| D2 | conv.u1 | - | * | int32 |
| D3 | conv.i | - | * | int32 |
| D4 | conv.ovf.i | - | * | int32 |
| D5 | conv.ovf.u | - | * | int32 |
| D6 | add.ovf | - | *,* | * |
| D7 | add.ovf.un | - | *,* | * |
| D8 | mul.ovf | - | *,* | * |
| D9 | mul.ovf.un | - | *,* | * |
| DA | sub.ovf | - | *,* | * |
| DB | sub.ovf.un | - | *,* | * |
| DC | endfinallyendfault | - | - | - |
| DD | leave | int32 | - | - |
| DE | leave.s | int8 | - | - |
| DF | stind.i | - | int32,& | - |
| E0 | conv.u | - | * | int32 |
| FE 00 | arglist | - | * | & |
| FE 01 | ceq | - | *,* | int32 |
| FE 02 | cgt | - | *,* | int32 |
| FE 03 | cgt.un | - | *,* | int32 |

| IL Instructions | | | | |
|---|---|---|---|---|
| Opcode | Name | Parameter(s) | Pop | Push |
| FE 04 | clt | - | *,* | int32 |
| FE 05 | clt.un | - | *,* | int32 |
| FE 06 | ldftn | <Method> | - | & |
| FE 07 | ldvirtftn | <Method> | o | & |
| FE 09 | ldarg | uint32 | - | * |
| FE 0A | ldarga | uint32 | - | & |
| FE 0B | starg | uint32 | * | - |
| FE 0C | ldloc | uint32 | - | * |
| FE 0D | ldloca | uint32 | - | & |
| FE 0E | stloc | uint32 | * | - |
| FE 0F | localloc | - | int32 | & |
| FE 11 | endfilter | - | int32 | - |
| FE 12 | unaligned. | uint8 | - | - |
| FE 13 | volatile. | - | - | - |
| FE 14 | tail. | - | - | - |
| FE 15 | initobj | <Type> | & | - |
| FE 17 | cpblk | - | int32,&,& | - |
| FE 18 | initblk | - | int32,int32,& | - |
| FE 1A | rethrow | - | - | - |
| FE 1C | sizeof | <Type> | - | int32 |
| FE 1D | refanytype | - | * | & |

# Anhang F - Kommandozeilenoptionen des IL-Assembler und Disassembler

## IL - Assembler

The command-line structure of IL Assembler is as follows:

```
ilasm [<options>] <sourcefile> [<options>][<sourcefile>*]
```

The default source file extension is IL. Multiple source files are parsed in the order of their appearance on the command line. Because options do not need to appear in a prescribed order, options and names of source files can be intermixed. All options specified on the command line are pertinent to the entire set of source files.

All options are recognized by the first three characters following the option key, and all are case-insensitive. The option key can be a forward slash (/) or a hyphen (-). In options that specify parameters, the equality character (=) is interchangeable with the colon character (:). The following option notations are equivalent:

```
/OUTPUT=MyModule.dll

-OUTPUT:MyModule.dll

/out:MyModule.dll

-Outp:MyModule.dll
```

The following command-line options are defined for IL Assembler:

- **/LISTING** Type a formatted listing of the compilation result.
- **/NOLOGO** Suppress typing the logo and copyright statement.
- **/QUIET** Suppress reporting the compilation progress.
- **/DLL** Compile to a dynamic-link library.

- **/EXE** Compile to a runnable executable (the default).
- **/DEBUG** Include debug information and create a program database (PDB) file.
- **/CLOCK** Measure and report the compilation times.
- **/RESOURCE=<res_file>** Link the specified unmanaged resource file (*.res) into the resulting PE file. <res_file> must be a full filename, including the extension.
- **/OUTPUT=<targetfile>** Compile to the file whose name is specified. The file extension must be specified explicitly; there is no default. If this option is omitted, IL Assembler sets the name of the output file to that of the first source file and sets the extension of the output file to DLL if the /DLL option is specified and to EXE otherwise.
- **/KEY=<keyfile>** Compile with a strong name signature. <keyfile> specifies the file containing the private encryption key.
- **/KEY=@<keysource>** Compile with a strong name signature. <keysource> specifies the name of the source of the private encryption key.
- **/SUBSYSTEM=<int>** Set the Subsystem value in the PE header. The most frequently used <int> values are 3 (Microsoft Windows console application) and 2 (Microsoft Windows GUI application).
- **/FLAGS=<int>** Set the Flags value in the common language runtime header. The most frequently used <int> values are 1 (pure-IL code) and 2 (mixed code). The third bit of the <int> value, indicating that the PE file is strong name signed, is ignored.
- **/ALIGNMENT=<int>** Set the FileAlignment value in the PE header. The <int> value must be a power of 2, in the range 512 to 65536.
- **/BASE=<int>** Set the ImageBase value in the PE header.

- **/ERROR** Attempt to create the PE file even if compilation errors have been reported.

Using the /ERROR option does not guarantee that the PE file will be created: some errors are abortive, and others lead specifically to a failure to create the PE file. This option also disables the following IL Assembler autocorrection features:

- An unsealed value type is marked sealed.
- A method declared as both static and instance is marked static.
- A nonabstract, nonvirtual instance method of an interface is marked abstract and virtual.
- A global abstract method is marked nonabstract.
- Nonstatic global fields and methods are marked static.

## IL - Disassembler

The command-line structure of IL Disassembler is as follows:

```
ildasm [<options>] [<in_filename>] [<options>]
```

If no filename is specified, the disassembler starts in graphical mode. You can then open a specific file by using the File Open menu command or by dragging the file to the disassembler's tree view window.

All options are recognized by the first three characters following the option key, and all are case-insensitive. The option key can be a forward slash (/) or a hyphen (-). In options that specify parameters, the equality character (=) is interchangeable with the colon character (:).

The **/ADVANCED** (**/ADV**) option sets the advanced mode of the disassembler, which offers additional viewing and dumping options. The **/ADV** option must be specified before any of the advanced-only options. I recommend that you place the **/ADV** option ahead of all other options.

### Options for Output Redirection

- **/OUT=<out_filename>** Direct the output to a file rather than to a GUI.
- **/OUT=CON** Direct the output to a console window rather than to a GUI.
- **/TEXT** A shortcut for /OUT=CON.

If the **/OUT** option or the **/TEXT** option is specified, the <in_filename> must be specified as well.

### ILAsm Code Formatting Options (PE Files Only)

- **/BYTES** Show the actual IL stream bytes (in hexadecimal notation) as instruction comments.
- **/RAWEH** Show structured exception handling clauses in canonical (label) form.
- **/TOKENS** Show metadata token values as comments.
- **/SOURCE** Show original source lines as comments. This requires the presence of the PDB file accompanying the PE file being disassembled and the original source files. If the original source files cannot be found at the location specified in the PDB file, the disassembler tries to find them in the current directory.
- **/LINENUM** Include references to original source lines (.line directives). This requires the presence of the PDB file accompanying the PE file being disassembled.
- **/VISIBILITY=<vis>[+<vis>...]** Disassemble only the items with specified visibility. Visibility suboptions (<vis>) include the following:
    - **PUB** Public
    - **PRI** Private
    - **FAM** Family
    - **ASM** Assembly

  - **FAA** Family and assembly
  - **FOA** Family or assembly
  - **PSC** Private scope

- **/PUBONLY** A shortcut for /VIS=PUB.
- **/QUOTEALLNAMES** Enclose all names in single quotation marks. By default, only names that don't match the ILAsm definition of a simple name are quoted.
- **/NOBAR** Suppress the pop-up window showing the disassembly progress bar.

**Options for File Output (PE Files Only)**

- **/UTF8** Use UTF-8 encoding for output. The default is ANSI.
- **/UNICODE** Use Unicode encoding for output.

**Options for File or Console Output (PE Files Only)**

- **/NOIL** Suppress ILAsm code output.
- **/HEADER** Include PE header information and runtime header information in the output.
- **/ITEM=<class>[::<method>[(<sig>)]** Disassemble the specified item only. If <sig> is not specified, all methods named <method> of <class> are disassembled. If <method> is not specified, all members of <class> are disassembled. For example, /ITEM="Foo" produces the full disassembly of the Foo class and all its members; /ITEM="Foo::Bar" produces the disassembly of all methods named Bar in the Foo class;

/ITEM="Foo::Bar(void(int32,string))" produces the disassembly of a single method, void Foo::Bar(int32,string).

- **`/STATS`** Include statistics of the image file; an advanced option.
- **`/CLASSLIST`** Include the list of classes defined in the module; an advanced option.
- **`/ALL`** Combine the /HEADER, /BYTES, and /TOKENS options and, in advanced mode, the /CLASSLIST and /STATS options.

### Metadata Summary Option

The metadata summary option is available in advanced mode only. It is suitable for file or console output, and it is the only option that works for both PE and COFF managed files. If an object file or an object library file is specified as an input file, the IL Disassembler in advanced mode automatically invokes the metadata summary, ignoring all other options. In nonadvanced mode, the disassembler does nothing.

- **`/METAINFO[=<specifier>]`** Show the metadata summary. The optional <specifier> is one of the following:
  - **`MDH`** Show the metadata header information and sizes.
  - **`HEX`** Show the hexadecimal representation of the signatures.
  - **`CSV`** Show the sizes of the #Strings, #Blob, #US, and #GUID streams and the sizes of the metadata tables and their records.
  - **`UNR`** Show the list of unresolved method references and unimplemented method definitions.
  - **`VAL`** Invoke the metadata validator and show its output.
- **`/OBJECTFILE=<obj_file_name>`** Show the metadata summary of a single object file in the object library. This option is valid for managed LIB files only.

# Anhang G - CD-ROM

Die beiliegende CD-ROM hat folgende Verzeichnisstruktur: