Fachhochschul-Masterstudiengang
SOFTWARE ENGINEERING
4232 Hagenberg, Austria

Cross-Platform Desktop Application Development with .NET and Mono

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science in Engineering

Eingereicht von

Daniel KNITTL-FRANK, BSc

Begutachter: Prof. (FH) DI Dr. Stefan WAGNER

September 2012

software engineering
IKM Fakultät Hagenberg

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg/Mühlkreis, September 3, 2012

Daniel Knittl-Frank, BSc

Contents

Abstract

This thesis introduces the concepts and ideas behind the Common Language Infrastructure standard, specified by ISO/IEC and Ecma, and its associated problems when used for cross-platform application development. Two popular implementations of the CLI standard are the .NET Framework from Microsoft and the open-source project Mono which is maintained by Xamarin. In addition, the paper identifies potential incompatibilities and common problem areas in applications intended to run across different platforms.

In the second part of the thesis, the experience gained in the first part is applied to an existing project, the meta-heuristic optimization framework HeuristicLab. Eventually, a working proof-of-concept prototype of HeuristicLab is presented which can be compiled and executed on different platforms. This prototype is then used to compare the performance of HeuristicLab under the Mono and .NET Framework when applying heuristic algorithms to optimization problems.

Kurzfassung

Diese Masterarbeit stellt die Konzepte und Ideen des von ISO/IEC und Ecma spezifizierten Common Language Infrastructure Standards und damit verbundene Probleme bei der Entwicklung von „Cross-Platform“-Anwendungen vor. Zur Zeit existieren zwei gängige Implementierungen dieses Standards, .NET von Microsoft und das Open-Source Projekt Mono. Nachdem die Grundlagen erörtert wurden, werden potenzielle Inkompatibilitäten und häufige Probleme von Anwendungen, welche auf unterschiedlichen Plattformen ausgeführt werden sollen, identifiziert und aufgezeigt.

Der zweite Teil der Arbeit beschäftigt sich mit dem HeuristicLab-Projekt für metaheuristische Optimierungsaufgaben und wendet die Erfahrungen aus dem ersten Teil praktisch an. Schließlich wird ein funktionierender Prototyp von HeuristicLab präsentiert, welcher auf unterschiedlichen Plattformen kompiliert und ausgeführt werden kann. Dieser Prototyp wird anschließend verwendet, um die Laufzeiten heuristischer Algorithmen bei der Anwendung auf Optimierungsaufgaben unter Mono und dem .NET Framework zu vergleichen.

Chapter 1

Introduction

1.1 Motivation

Microsoft .NET allows developers to write applications that are independent of the actual hardware architecture by executing them in a virtual machine. It was introduced by Microsoft in early 2002 and later standardized by ISO/IEC and Ecma International as *Common Language Infrastructure* (CLI). Many features ease application development, such as automatic memory management and a rich set of class libraries. Numerous compilers for different source languages exist and developers can write applications in any of these languages. Popular source languages for .NET include C#, Visual Basic.NET, and F#.

.NET originally ran only on Windows operating systems. The Mono project is an effort to bring the power of .NET to other operating systems besides Windows, such as Linux and Mac OS. It provides a managed runtime environment to run .NET applications, compilers for several source languages, and implementations of the core class libraries.

With Linux' high market share in scientific and high performance computing environments[1], as well as its increasing market share on desktop systems and mobile devices, it only makes sense to leverage .NET's features on Linux operating systems.

Using different implementations of the CLI does not come without complications. Common problems shall be identified and solutions shall be presented as part of this thesis. A case study is conducted to test the practicability of adapting a real-world .NET application. Subject matter of the case study is the HeuristicLab framework.

HeuristicLab is a highly modular framework to tackle heuristic optimization problems of all sorts. The project was started in 2002 at the Johannes Kepler University Linz [38]. The latest released version is HeuristicLab 3.3.7, which was released on July 8, 2012.

[1] http://i.top500.org/overtime, TOP500 Statistics

While several other frameworks exist in the field of heuristic optimization, few of them fulfill requirements with regard to ease of learning, flexibility, user experience, etc. A detailed analysis and comparison of existing software solutions can be found in several papers [38, 30].

When the HeuristicLab project was started, it was decided to use the Microsoft .NET Framework as a runtime basis. The primary reason for this decision at that time was a well integrated GUI framework, whereas Java did not provide a solid GUI framework. Compatibility with distinct implementations of the CLI was not a concern when the project was started, and HeuristicLab currently only executes under .NET on Windows operating systems. To reach a broader user base, it is therefore necessary for the HeuristicLab environment to be compatible with the Mono project.

1.2 Goals

Over the course of this thesis three main goals are pursued. To begin with, different implementations of the CLI, foremost .NET and Mono, are analyzed and compared with each other. Key differences are highlighted and missing features are listed.

Secondly, common problems are identified that appear when converting a .NET application to be compatible with the Mono platform. General guidelines and solutions have to be found where applicable. Furthermore, applications which cannot be ported without an extraordinary effort of the application developers should be identified early on to save time and money.

Finally, a case study is presented in which the HeuristicLab environment is adapted so that it compiles and runs in the Mono environment on Linux. While Mono provides a good and stable basis for running .NET applications on a variety of operating systems, there are still some bugs and deviant behavior compared to Microsoft .NET. The case study helps to discover and identify bugs in Mono, which detain developers from running their applications with the Mono runtime. These bugs have to be worked around on the client side (HeuristicLab) or fixed directly in the runtime (Mono), either by writing patches or simply reporting the bugs back to the Mono development team.

Developers should be able to build HeuristicLab on Linux with the tools provided by Mono and run it with the Mono runtime on at least Linux and Windows systems, in addition to the current .NET runtime.

This thesis deals with version 3.3.6 of HeuristicLab, released in January 2012. A branch of HeuristicLab 3.3.6 has been created in its source code repository[2] to act as an isolated work space to re-write parts of HeuristicLab's source code to support more environments. These changes will eventually get merged to the project's main line of development.

[2] http://dev.heuristiclab.com/svn/hl/core/branches/HeuristicLab.Mono/

1.3 Content

This thesis is divided into six chapters. The first chapter contains the introduction and defines the motivation and goals of this thesis.

The second chapter discusses the relevant aspects of the Common Language Infrastructure (CLI) – including the Common Type System (CTS), Common Language Specification (CLS), Common Intermediate Language (CIL), and Base Class Library (BCL) – as standardized by the International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) in *ISO/IEC 23271* [20] and by Ecma International in *ECMA-335* [11]. The chapter ends with a presentation of popular languages used in the context of the CLI.

There are two popular implementations of the CLI, namely the .NET Framework from Microsoft and the open-source project Mono from Xamarin. An overview of the characteristics of each runtime is given in Chapter 3, with a strong focus on the Mono project. Other implementations of the standard are briefly presented as well.

The next chapter highlights differences and incompatibilities between the two implementations. It concentrates on missing classes from the class libraries and deviating behavior in existing classes. The toolchain for writing and building .NET applications is different when using Mono. Important aspects include the compiler, IDE (Integrated Development Environment) and related tools (build scripts, unit tests, ...). One of the motivations of executing HeuristicLab under Mono is the ability of Mono to run on operating systems besides Microsoft Windows, specifically on Linux distributions such as Ubuntu.

Chapter 5 presents a case study. The HeuristicLab framework for heuristic and evolutionary algorithms is modified to be able to compile with the Mono toolchain and to run under the Mono runtime. The chapter concludes with a performance analysis of Microsoft .NET, Mono on Linux, and Mono on Windows.

This thesis concludes with an outlook of the future of Mono and the HeuristicLab framework in the last chapter.

Chapter 2

Common Language Infrastructure (CLI)

This chapter summarizes the most important aspects of the Common Language Infrastructure. The first version of ISO/IEC 23271 was published in 2003 [18], the third and current revision of the standard was published in February 2012 [20]. The CLI is described as:

> "This International Standard defines the Common Language Infrastructure (CLI) in which applications written in multiple high-level languages can be executed in different system environments without the need to rewrite those applications to take into consideration the unique characteristics of those environments." [20]

The CLI consists of four main parts which are covered in more detail in this chapter [20]:

- The **Common Type System (CTS)** aims to provide a rich set of types and operations for supporting a wide range of different programming languages.
- **Metadata** is used by the CLI to describe types from the Common Type System. It is independent from source languages and provides a common interchange mechanism.
- The **Common Language Specification (CLS)** is a subset of the CTS. Languages should at least support the features specified by the CLS to allow users to access frameworks.
- The **Virtual Execution System (VES)** is responsible for loading and running programs written for the CLI.

2.1 Common Type System (CTS)

The *Common Type System* (CTS) provides a rich set of types and supported operations on these types. It supports types of procedural, object-oriented and functional programming languages. These three paradigms require different entities, *values* and *objects*. The relationship between types is displayed in Figure 2.1.

Values are blocks of bits representing simple scalar data; examples of values are integer and float values. A value type describes the concrete bit pattern used to store a type and the operations which can be performed on that value representation. Every value type has a matching reference type, a so-called *boxed type*.

Objects in the CTS explicitly store their type in their representation. Every object is distinguishable from all other objects through its identity. Other objects and values are stored in slots which can be changed and reassigned. The identity of an object is always immutable and not affected by the actual values of slots.

Families of types and methods can be defined by using the provided *generics feature* of the CTS. This feature is independent of the source language. Generic types can be used in place of normal CLI types, if context permits. Generic types are statically checkable and thus can be validated and verified (see the next sections for details). Generic interfaces and delegates allow generic parameters to be used in *covariant* or *contravariant* position. Source languages with no support for variance can simply ignore variant information on parameters and behave as if the parameters were non-variant parameters.

2.2 Metadata

Metadata is used to represent information which the CLI requires during execution. This information includes the way classes are loaded, how memory is laid out, how methods are invoked, and more [20]. Type declarations for the CTS are expressed in metadata as well.

Since each CLI component has its specific metadata directly attached, a component in CLI is *self-describing*. Metadata belonging to a component is called *component metadata*. Metadata can be accessed through the *Reflection* library or directly through the file format which is described in the CLI standard. For an in-depth explanation of specification details refer to Partition II of the Common Language Infrastructure [20].

Metadata enables a conforming CLI implementation to perform a number of tests to validate and verify CIL (presented in Section 2.5). *Validation* tests ensure the self-consistency of the file format, metadata and CIL. Behavior of a CLI implementation is unspecified, if presented with a file which

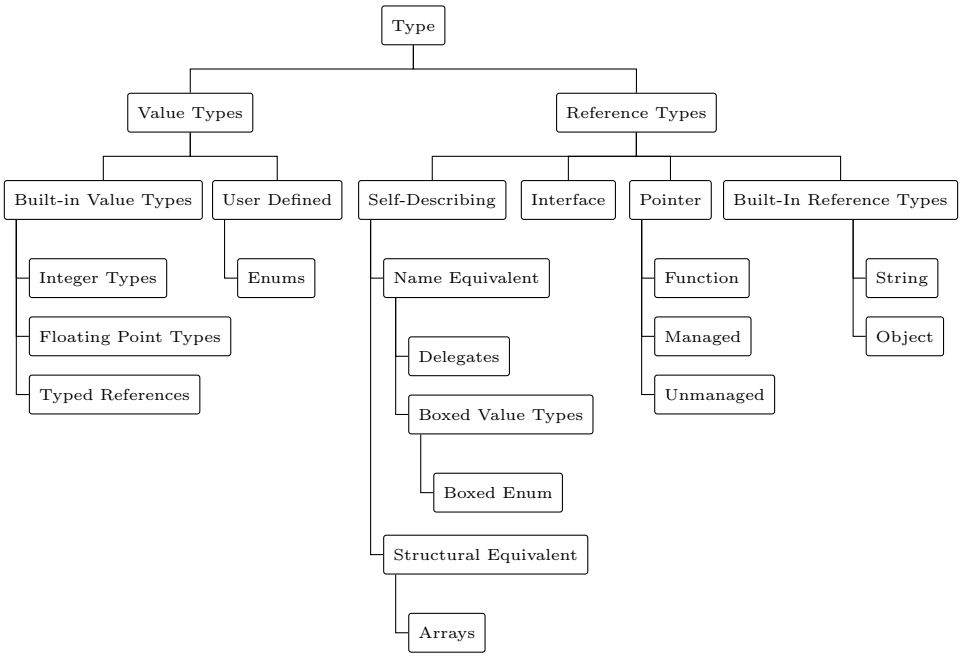

Figure 2.1: Common Type System [20]

did not pass the validation test. *Verification* ensures that CIL instructions only access memory inside the program's address space [15]. The complete list of rules for verifiable CIL instructions can be found in Partition III of the CLI standard [20]. Code might be type-safe, but cannot be proven to be so by the simple verification algorithm of the VES. A conforming CLI implementation is allowed to execute unverifiable code, but might impose additional security controls that are not covered by the standard. The relationship of verifiable and correct CIL is shown in Figure 2.2.

2.3 Common Language Specification (CLS)

The Common Language Specification (CLS) is a set of rules to benefit interoperability of different source languages used with the CLI. Types for execution on a CLI implementation must conform to the CLI standard and CLS rules. The complete list of rules can be found in Partition I of *ISO/IEC 23271* [20]. Only types visible outside their containing assembly must adhere to these rules.

The CLI standard refers to libraries consisting of CLS-compliant code as *framework*. Compilers which only consume frameworks are referred to as *consumers*, those that extend frameworks are referred to as *extenders*.

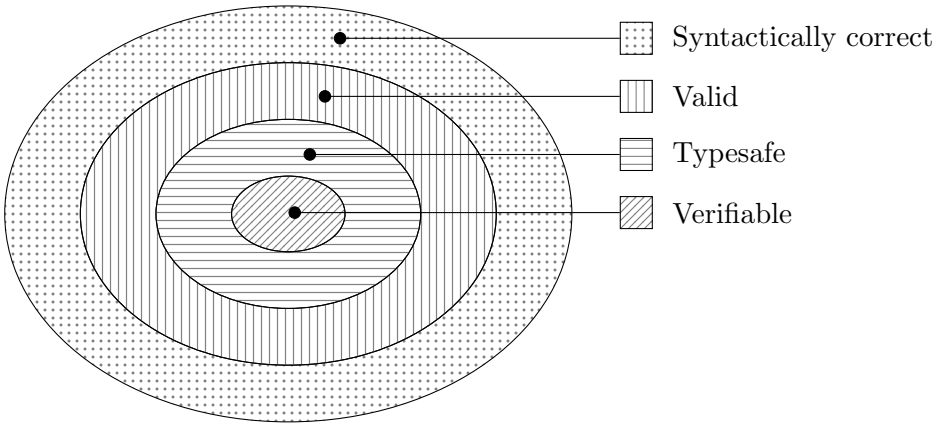

Figure 2.2: Validation and Verification of CIL [20]

2.4 Virtual Execution System (VES)

The Virtual Execution System (VES) provides an environment for executing managed code. It provides the infrastructure to execute the CIL instruction set and defines a hypothetical state machine. The CIL instruction set is discussed in the next section.

The VES directly supports a number of data types, mainly different integer types (signed and unsigned, ranging from 8 bit to 64 bit), and 32 bit and 64 bit IEC 60559:1989 floating point numbers. Additionally, native object references and pointers are supported which must point into managed memory. Direct support means that these types can be modified with CIL instructions. These types are referred to as the *basic CLI types* [20].

The CLI uses a stack to evaluate its intermediate language and allows only a subset of the listed types to be used on the stack: `int32`, `int64` and `native int` – other integer types are widened and narrowed as required when pushed to or loaded from the stack. The reason behind this is that it simplifies compilation from CIL to native code, because compilers only need to track a small number of types internally.

2.5 Common Intermediate Language (CIL)

The VES runs an instruction set which is known as *Common Intermediate Language* (CIL). In the CLI implementation by Microsoft, it is also referred to as MSIL (Microsoft Intermediate Language). IL Instructions only deal with the aforementioned *basic CLI types* (`int32`, `int64`, `native int`, `native float`, `object reference`, `native pointer`). Each instruction has a predefined evaluation stack transition diagram associated as well as a list of exceptions that can be thrown by that specific instruction. The standard

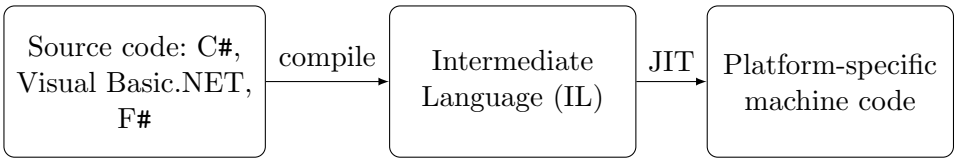

Figure 2.3: Two Step Compilation Model [31]

describes conditions for the verifiability and correctness of each instruction plus its prefixes in clauses 2 through 4 of Partition III [20].

As discussed in Section 2.2 (Metadata), it is important that applications only access their own address space. CIL is tested for *verifiability* (the innermost area in Figure 2.2), since it is not possible to reliably test memory and type safety of an application. An example of a non-verifiable operation is pointer arithmetic which is required for C programs. The algorithm used to test verifiability of IL instructions is explained in-depth in subclause 1.8 of Partition III of the Common Language Infrastructure [20].

Every IL instruction starts with an opcode which is one or more bytes long. Opcodes are followed by a variable number of operands for this instruction. In the current version of the standard opcodes occupy either one or two bytes, and all multi-byte opcodes start with `0xFE`. First bytes in the range `0xF0` through `0xFB` are not used by the standard and may be used for experimental purposes. However, if the first byte of an opcode is in the range `0x00` through `0xEF` or `0xFC` through `0xFF`, then these opcodes are reserved for later standardization. A list of available opcodes can be found in *ISO/IEC 23271* [20].

The CLI uses a two step compilation model: Source code is first compiled to IL byte code, and IL is then efficiently compiled to native machine code when executed by the VES. Implementations need not use JIT compilation and could use an interpreter to execute IL directly instead. The diagram in Figure 2.3 shows the steps and code states involved in this procedure.

2.6 Profiles

ISO/IEC 23271 defines two standard profiles [20]. A profile specifies the minimum amount of features and libraries that a conforming implementation of the CLI must provide. The smallest profile is the *Kernel Profile* which is targeted at low end devices. The *Compact Profile* contains additional useful features, such as XML, networking, and reflection libraries. Conforming implementations of the CLI shall specify which profiles they implement and which libraries they support.

Seven *Standard Libraries* are specified in addition to the profiles. Runtime Infrastructure Library, Base Class Library, XML Library, Network Li-

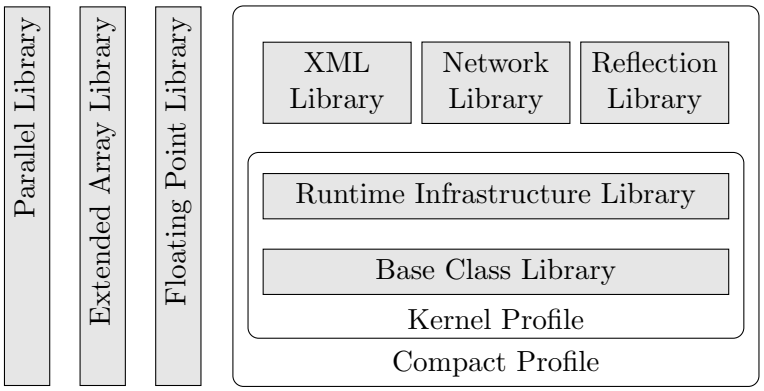

Figure 2.4: CLI Libraries and Profiles [20]

brary and Reflection Library are part of the standard profiles. Provided, but not part of the standard profiles are the Floating Point Library, the Extended Array Library and the Parallel Library. The relationship between standard profiles and standard libraries is depicted in Figure 2.4.

The *Kernel Profile* only contains the Runtime Infrastructure Library and Base Class Library. It does not contain the Floating Point Library. Other features missing from the Kernel Profile are non-vector arrays, reflection, application domains, remoting, variable argument lists, dynamically extending a stack frame (frame growth) and filtered exceptions. The *Compact Profile* is a superset of the Kernel Profile plus XML Library, Network Library, and Reflection Library.

The standard explicitly allows implementations to extend the standard libraries with custom functionality. It restricts modifications by a set of rules to keep programs portable between different implementations. In particular, these rules permit or prohibit the following [20]:

- Contracts of virtual methods shall be maintained when overriding.
- New exceptions can be introduced and thrown by methods. However, these methods should extend `System.Exception`.
- Existing interfaces shall not be extended with additional interfaces or virtual methods.
- Access rules to members of a type can be more permissible than specified. The standard only specifies a minimum accessibility.
- Instance methods shall not be implemented as virtual methods. This improves portability of libraries, as it prevents the usage of implementation specific libraries during compile time.

2.7 Supported Languages

The Common Language Infrastructure is independent of source languages, as long as a compiler exists which transforms the source language into *Intermediate Language*. This chapter gives a short overview of languages commonly used in connection with the CLI. The rest of this thesis mainly focuses on the C# source language.

2.7.1 C#

C# is a modern, general-purpose, object-oriented, strongly-typed programming language. C# was developed by Microsoft and is specified in the standards ISO/IEC 23270 and ECMA-334 [19, 12]. The syntax resembles C and C++, to make it easier for developers to learn the new language – the famous Hello World program implemented in C# can be seen in Listing 2.1. Microsoft released its first implementation of the language in the year 2000.

Listing 2.1: C# Source Code Sample

```
1 using System;
2
3 public class HelloWorld {
4   public static void Main(string[] args) {
5     Console.Out.WriteLine("Hello World");
6   }
7 }
```

A conforming implementation of C# must provide types which map to the Base Class Library of the Common Language Infrastructure, as well as types for the Extended Numerics Library and Extended Array Library (cf. Section 2.6). These types can be found in the reserved `System` namespace.

C# is compiled to Intermediate Language which is then executed by the Virtual Execution System. This means C# inherits any benefits the Virtual Execution System offers, including automatic memory management, true generics, and exceptions.

2.7.2 Visual Basic.NET

Visual Basic.NET (often *VB.NET*) is a language designed by Microsoft and based on the Visual Basic language. It provides the same feature set as C#.

Listing 2.2: Hello World with Visual Basic.NET[1]

```
1 Imports System
2
3 Public Module HelloWorld
4    Sub Main()
5      Console.WriteLine ("Hello World using Visual Basic!")
6    End Sub
7 End Module
```

2.7.3 Managed Extensions for C++

Managed Extensions for C++ is a source language strongly resembling the classic C++ language. It extends the original C++ language with support for automatic garbage collections.

Listing 2.3: Hello World with Managed Extensions for C++[2]

```
1 #using <mscorlib.dll>
2
3 using namespace System;
4
5 void main() {
6    Console::WriteLine(S"Hello World using Managed Extensions for C++!");
7 }
```

2.7.4 F#

F# is a newer source language, designed by Microsoft. Its compiler source code is available under the open-source Apache 2.0 license[3]. F# provides developers with a set of functional and imperative programming paradigms, while still allowing access to all CTS types. With immutable data types, F# enables programmers to develop concurrent and highly scalable systems.

Listing 2.4: Recursive Definition of the Fibonacci Number Function in F#

```
1 let rec fib n =
2     match n with
3     | 0 | 1 -> n
4     | _ -> fib (n - 1) + fib (n - 2)
```

2.7.5 Other Supported Languages

A large number of other source languages (and respective compilers) exist for the Common Language Infrastructure. After compilation to IL, programs can be run in any compliant implementation of the ISO/IEC 23271 standard. The Mono website provides an extensive list of supported source languages[4]. This list contains Boo, Nemerle, Python, and JavaScript, among others. Easton and King have put together a list of languages and compilers for the CLI and present them in Chapter 8 of their book [10]. Noteworthy is the support of *Java* and *Scala*, both originally designed for the JVM, as source languages targeting the Common Language Infrastructure.

[1] http://msdn.microsoft.com/en-us/library/aa309383.aspx, Hello World in Visual Basic

[2] http://msdn.microsoft.com/en-us/library/aa309382.aspx, Hello World in Managed Extensions for C++

[3] http://fsharppowerpack.codeplex.com, F# PowerPack: F# Extras, with F# Compiler Source Drops

[4] http://www.mono-project.com/Languages

Chapter 3

CLI Implementations – .NET and Mono

The previous chapter introduced the concepts of the Common Language Infrastructure, as specified in ISO/IEC 23271 and ECMA-335. There are currently two popular implementations of this standard. The most popular implementation of that standard is the proprietary Microsoft .NET Framework. Mono is an effort to enable developers to use the Common Language Infrastructure on platforms other than Microsoft Windows. Both implementations are discussed on the following pages.

3.1 .NET

.NET is a proprietary implementation of the Common Language Infrastructure standard (ISO/IEC 23271), described in the previous chapter. The .NET Framework is developed and distributed by Microsoft. A beta version of the .NET Framework was available in 2001, and version 1.0 was released in 2002.

Microsoft's implementation of the CLI consists of the Common Language Runtime (CLR), and the Framework Class Library (FCL). The CLR is Microsoft's implementation of the VES, responsible for executing IL; the FCL is a superset of the BCL (Base Class Library).

3.1.1 Version History of .NET

This section gives a short overview of the different versions of the .NET Framework and the most significant improvements compared to the preceding version. The current release of the .NET Framework is version 4.0 and a release candidate version of .NET 4.5 has been available for testing since end of May 2012.

Release dates were aggregated from http://microsoft.com/download. Information on specific versions was compiled from several MSDN articles[1].

.NET 1.0 and .NET 1.1

.NET 1.0 was the first release of the .NET Framework which was released in early 2002. It is no longer supported as of July 2009.

.NET 1.1 added complete IPv6 support and introduced some security changes[2]. The *.NET Compact Framework* for mobile devices was first provided with this release. Part of .NET 1.1 was also a version bump of the CLR to version 1.1. The new version of the CLR introduced side-by-side execution which allows multiple versions of the CLR to be installed on the same operating system. Applications can then choose the particular version of the CLR with which they are executed.

.NET 2.0

The Microsoft .NET Framework 2.0 was released on January 22, 2006[3]. Version 2.0 of the framework was the first version to provide full 64 bit support on the IA-64 and x86 hardware platforms.

The CLR received several updates and was upgraded to version 2.0. .NET 2.0 introduced native generics support on the level of the CLR, as described in Section 2.1. New collection classes which make use of the new generics features were included with the Framework Class Library. Other notable new features of this release included partial class declarations, nullable types, anonymous methods, and iterators.

With the .NET 2.0 release, Microsoft started shipping the *Micro Framework*[4], an even more stripped-down version than the .NET Compact Framework. It targets devices with a 32 bit processor without an external memory management unit and just 64 kilobytes of RAM.

.NET 3.0 and .NET 3.5

Version 3.0 of the framework was released on November 21, 2006[5] and version 3.5 was released one year later, on November 20, 2007. .NET 3.0 comes pre-installed with Windows Vista and Windows Server 2008; Windows 7 includes .NET 3.5 by default.

[1] http://msdn.microsoft.com/en-us/library/bb822049.aspx, .NET Framework Versions and Dependencies

[2] http://msdn.microsoft.com/en-us/magazine/cc164162.aspx, Namespace, Security, and Language Support in the .NET Framework 1.1

[3] http://www.microsoft.com/en-us/download/details.aspx?id=19, Download: .NET Framework Version 2.0 Redistributable (x86)

[4] http://microsoft.com/netmf/about, .NET Micro Framework – About

[5] http://www.microsoft.com/en-us/download/details.aspx?id=31, Download: Microsoft .NET Framework 3.0 Redistributable Package

.NET 3.0 introduced a number of new libraries which are not specified by ISO/IEC and provide additional functionality. These library stacks are WPF (Windows Presentation Foundation), WCF (Windows Communication Foundation), and WF (Windows Workflow Foundation). These technologies are outlined briefly in Chapter 4. The version of the CLR remained at 2.0.

With .NET 3.5, Microsoft made the source code of its implementation of the Standard Libraries publicly available to assist with debugging. It is released under the *Microsoft Reference Source License*[6], which allows developers to view the source code for reference, but nothing else.

.NET 4.0

This next release of the .NET Framework simplified access to parallel programming paradigms. Developers can use the *Task Parallel Library*[7] (TPL) and *PLINQ*[8] (Parallel Language Integrated Query) to improve performance of their applications on multi-core systems. Additionally, new numerical types were added, for instance `System.Numerics.BigInteger`, to perform arbitrary-precision arithmetic and `System.Numerics.Complex` to represent complex numbers. Furthermore, .NET 4.0 introduced some new source language features, such as named and optional parameters. .NET 4.0 platform update 1 (.NET 4.0.1) added some features to WF, notably classes to design state machine workflows.

.NET 4.0 was released on April 12, 2010[9]. It is not pre-installed on the Microsoft Windows operating systems. The CLR version number of this release is 4.

.NET 4.5

.NET 4.5 is the upcoming release of the Microsoft .NET Framework. It includes new core features and extensions of existing technologies, such as WPF, WCF, and WF[10]. Developers currently have the ability to test a release candidate of this framework version. The final version of the .NET Framework 4.5 is planned to be released later this year.

---

[6] http://referencesource.microsoft.com/netframeworklicense.aspx, Microsoft .NET Framework Reference License

[7] http://msdn.microsoft.com/en-us/magazine/cc163340.aspx, Parallel Performance: Optimize Managed Code for Multi-Core Machines

[8] http://msdn.microsoft.com/en-us/magazine/cc163329.aspx, Parallel LINQ: Running Queries on Multi-Core Processors

[9] http://www.microsoft.com/en-us/download/details.aspx?id=17718, Download: Microsoft .NET Framework 4 (Standalone Installer)

[10] http://msdn.microsoft.com/en-us/library/ms171868.aspx, What's New in the .NET Framework 4

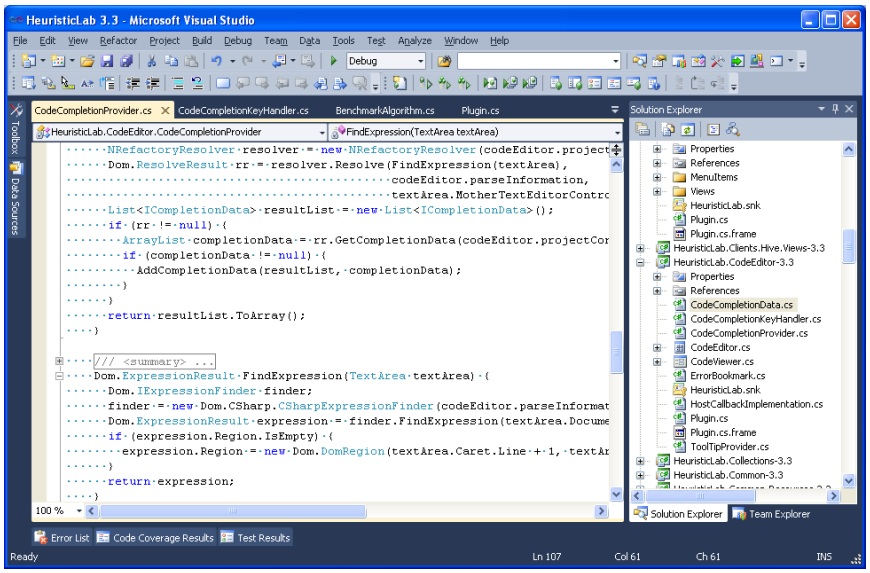

Figure 3.1: Screenshot of Microsoft Visual Studio 2010 Premium Edition

Language extensions include the `async` and `await` keywords for the C# and Visual Basic.NET source languages[11]. These keywords simplify the development of asynchronous operations by automatically suspending asynchronous methods and handing back control to the calling method, while waiting for a long running method to complete. Once the result from the long running method is available, the suspended method is resumed automatically.

3.1.2 Visual Studio

Microsoft Visual Studio is an IDE from Microsoft to aid developers writing – not only .NET – code. The current version of Visual Studio is *Microsoft Visual Studio 2010*[12]. Visual Studio 2012 is set to be released with the upcoming .NET 4.5 release. Figure 3.1 shows a screenshot of Visual Studio 2010 Premium edition.

Visual Studio is available in different editions (Express, Professional, Premium, and Ultimate), each with a different set of features. Microsoft also offers a stripped-down, free version of Visual Studio, *Visual Studio Express*, which offers less functionality and has no add-on support. Prices range from free (Express Edition) to $ 11899 for the Ultimate edition, as of July 2012.

[11] http://msdn.microsoft.com/en-us/library/hh191443.aspx, Asynchronous Programming with Async and Await (C# and Visual Basic)

[12] http://microsoft.com/visualstudio

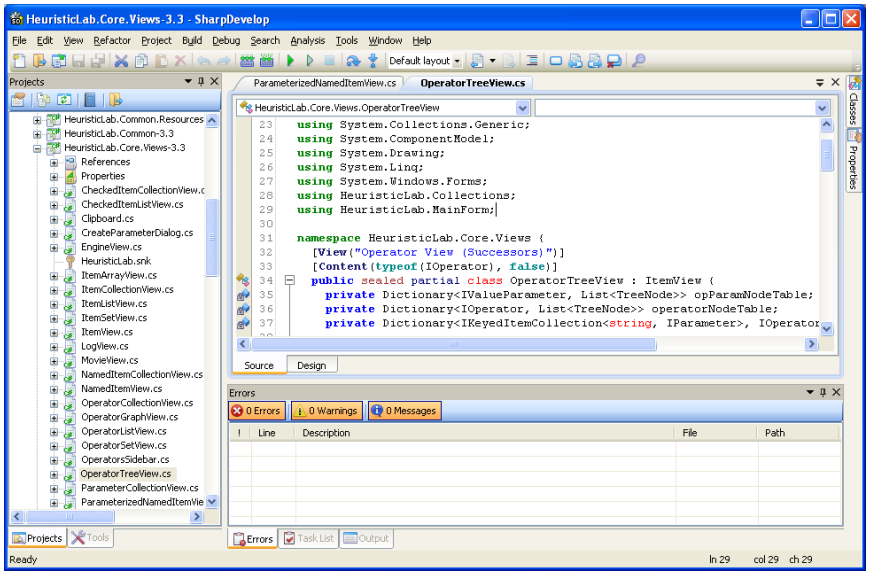

Figure 3.2: Screenshot of SharpDevelop 4.2

3.1.3 SharpDevelop

SharpDevelop (#develop) is an open-source IDE for developing .NET applications, maintained by IC#Code. It provides a feature set comparable to that of Visual Studio Express edition. Some of the features include GUI designers for Windows Forms projects, syntax-highlighting support for different source languages, and code completion [16].

SharpDevelop heavily relies on P/Invoke and will therefore not run on operating systems besides Microsoft Windows. It also uses a WPF based text editor component (AvalonEdit), as of version 4. A screenshot of SharpDevelop 4 is shown in Figure 3.2.

The current version of SharpDevelop is SharpDevelop 4, released on May 6, 2012. Supported operating systems are Windows XP SP2 and later[13].

3.2 Mono

Mono is an open-source implementation of the Common Language Infrastructure which is specified in ISO/IEC 23271 and ECMA-335. It contains a compiler for the different CLI languages, a runtime environment and common class libraries. The project was started in 2002 by Miguel de Icaza who was working at Novell Inc. at that time. Mono is currently maintained by Xamarin. Several books present and discuss development with Mono [14, 33, 9, 21, 26].

[13] http://www.icsharpcode.net/OpenSource/SD/Download, Downloads @ic#code

Critical voices discourage the use of Mono especially for free, open-source software projects. Microsoft holds patents which might be infringed by implementations of the CLI and C# standards. The Free Software Foundation, founded by Richard M. Stallman, is probably the most prominent voice in this direction[14]. Microsoft has put both CLI and C# specifications under the *Microsoft Community Promise*[15]. This promise is irrevocable and states that Microsoft will not assert claims against implementers and users of these specifications.

Developers of Mono generally implement new language and runtime features very fast. For example, the new keywords `async` and `await` of .NET 4.5 and C# 5.0 are already available[16] under Mono. Also, Mono supports syntax extensions of source languages, such as *LINQ*.

The current stable version of Mono is Mono 2.10.8 and was released on December 19, 2011. An alpha version (Mono 2.11.2) is also available for download on the Mono website.

3.2.1 Compiler

Several compilers for different source languages exist for Mono, including compilers for C#, VB.NET, and F#. The C# compiler supports several versions of the language, starting from C# 1.0 to C# 5.0. The compiler of Mono produces conforming CIL code, as specified in ISO/IEC 23271. Therefore assemblies compiled with Mono will run in Microsoft's .NET Framework and any other compliant implementation of the Common Language Infrastructure.

Developers can leverage the compiler's functionality by referencing the `Mono.Sharp.dll` assembly in their projects and using the `Mono.CSharp.Evaluator` class. This allows the usage of the compiler from client code and applications (compiler-as-a-service).

Java support is provided through *IKVM.NET*[17], which is shipped as part of Mono. IKVM.NET contains a static compiler to translate Java files (`.java`, `.class` or `.jar`) into Intermediate Language.

3.2.2 Runtime

Mono provides a runtime which runs on several operating systems, including the popular Windows, Linux and Mac OS operating systems. Behavior of the runtime is specified in ISO/IEC 23271 [20].

[14] http://www.fsf.org/news/dont-depend-on-mono, Why free software shouldn't depend on Mono or C#

[15] http://www.microsoft.com/openspecifications/en/us/programs/community-promise/default.aspx, Community Promise, Microsoft Open Specifications

[16] http://tirania.org/blog/archive/2012/Mar-22.html, Mono 2.11.0 is out

[17] http://www.ikvm.net/

The runtime needs to be able to execute and compile CIL to native instructions on the target system. The Mono runtime can do this with a Just-in-Time and an Ahead-of-Time compiler. They transform Common Intermediate Language into native operations on the target platform.

Java runtime support is again provided through IKVM.NET which provides an implementation of the JVM on top of the CLI and allows Java programs to be executed inside the Virtual Execution System of the CLI. Java byte code can be directly executed under the runtime of a CLI compliant implementation, through the JIT compiler of IKVM.

Garbage Collectors

This section gives a brief overview of the two garbage collectors that are implemented in Mono. Information was compiled from the official Mono website[18] and the source code.

Mono currently provides two distinct garbage collectors. The first one is named *Conservative Boehm GC*, the second, newer GC goes under the name *SGen – Precise Generational GC*, short *SGen* (Simple Generational). In the current release, Mono uses the Boehm GC by default, but it can be configured to use SGen instead.

The Boehm GC of Mono is an implementation of the Boehm-Demers-Weiser garbage collector[19] [4]. This GC was originally written for C and C++ applications and replaces `malloc` and `new`, making it unnecessary to explicitly call `free` or `delete`. Since it was written for unmanaged languages, Boehm has its limitations when used in a managed environment such as the CLI. This was the main motivation which lead to the development of the SGen garbage collector.

SGen uses two generations to optimize GC performance. Objects are created in a so-called *nursery section*, and are moved to the second generation after they survived the first garbage collection. A *Large Object Space* is provided for objects larger than 8000 bytes to prevent memory fragmentation.

SGen can currently be used optionally and is likely to become the default garbage collector with one of the next Mono releases. It is enabled by appending the `--gc=sgen` option when invoking Mono, or by adding this value to the environment variable `MONO_ENV_OPTIONS`.

Both garbage collectors of Mono are *stop-the-world* garbage collectors. This means that all active threads are suspended during garbage collection and are resumed after the collection has completed.

[18] http://mono-project.com/Generational_GC, http://mono-project.com/Working_With_SGen

[19] http://www.hpl.hp.com/personal/Hans_Boehm/gc/, A garbage collector for C and C++

3.2.3 Libraries

To provide a conforming implementation of the CLI, Mono is required to provide the libraries described in Partition IV of the standard and its corresponding XML file, the Standard Libraries [20]. These libraries are located in the `System` namespace. Appendix B of "Cross-Platform .NET Development" contains diagrams listing all namespaces defined by the Common Language Infrastructure, as well as namespaces grouped by their dependence on the underlying architecture [10].

Mono also provides a number of libraries which are not specified by the Common Language Infrastructure. Not all of them are fully implemented though. One such library which is not contained in ISO/IEC 23271 is Windows Forms for writing GUI applications with native Windows look and feel. Section 4.5 on page 28 describes the current status of the Mono implementation of `System.Windows.Forms`.

Microsoft has released the source code of several of its formerly proprietary libraries under open-source licenses. Just recently, the *Entity Framework* was open-sourced under the Apache License 2.0[20]. A few months earlier, ASP.NET MVC 4, ASP.NET Web API, and ASP.NET Web Pages v2 (Razor) were open-sourced[21]. The Mono project incorporates these libraries as soon as their source code is available under a GPL compatible license.

Several libraries are still missing from Mono or lack features due to license incompatibilities, unavailable developer resources, or strong dependencies on the underlying operating system. Examples are the WPF, WCF, and WF libraries, which were introduced as part of .NET 3.0. The current status of these and other libraries in Mono is presented in the next chapter.

3.2.4 MonoDevelop

MonoDevelop is an integrated development environment (IDE) and is the equivalent of Microsoft's Visual Studio IDE. It supports a wide range of programming languages (including C#) and can read and write solution (`.sln`) and project (`.csproj`) files used by Visual Studio. All versions of the format (Visual Studio 2005, 2008 and 2010) are currently supported. MonoDevelop was started in 2003 in an effort to port SharpDevelop to Mono and GTK#.

Many of the features that developers are accustomed to from other IDEs, such as Visual Studio or Eclipse, are available in MonoDevelop as well. These features include, but are not limited to, auto-completion, breadcrumb navigation, class browser, version control integration and a debugger. An

[20] http://blogs.msdn.com/b/adonet/archive/2012/07/19/entity-framework-and-open-source.aspx, Entity Framework and Open Source

[21] http://weblogs.asp.net/scottgu/archive/2012/03/27/asp-net-mvc-web-api-razor-and-open-source.aspx, ASP.NET MVC, Web API, Razor and Open Source

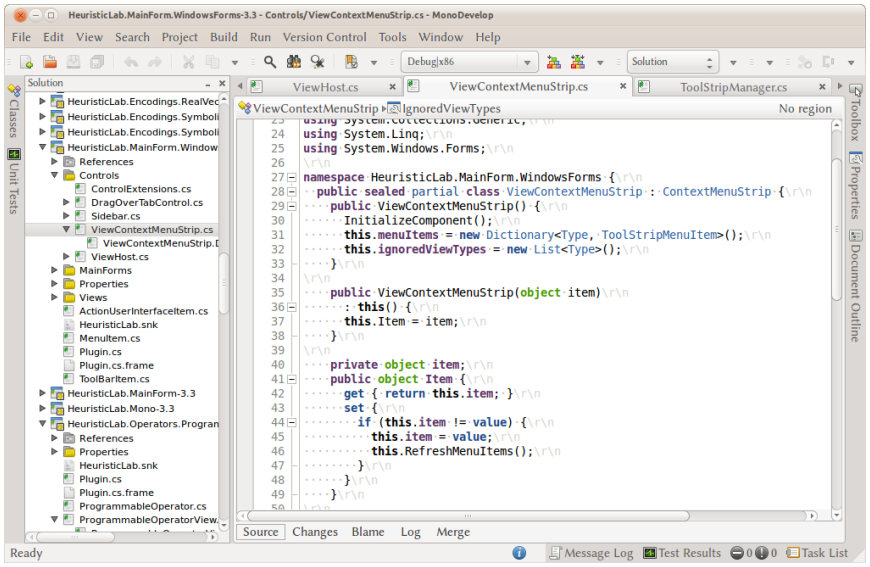

Figure 3.3: The MonoDevelop Integrated Development Environment

add-in manager enables developers to extend the functionality of their IDE with third party add-ins.

Similar to the core Mono infrastructure, MonoDevelop runs on the major operating systems (Linux, Mac OS, Windows). It relies on the multi-platform GTK+[22] project for drawing its GUI and widgets. A screenshot of MonoDevelop running in Ubuntu can be seen in Figure 3.3

The latest release is MonoDevelop 3.0 and was announced on May 14, 2012[23]. A list of important changes for this release can be found on the MonoDevelop website[24].

3.3 Other Implementations

.NET and Mono are not the only implementations of the ISO/IEC 23271 standard. Other implementations exist, but are not as popular. A handful of other implementations is listed in the book "Cross-platform development with .NET" [10]. A short summary of each implementation is given in the next few paragraphs.

[22] http://gtk.org, The GTK+ Project
[23] http://monodevelop.com/Download/Release_Notes/Release_Notes_for_MonoDevelop_3.0
[24] http://monodevelop.com/Download/What%27s_new_in_MonoDevelop_3.0

3.3.1 Microsoft .NET Compact Framework

The Microsoft .NET Compact Framework was introduced together with .NET 1.1 and is a stripped down version of the Microsoft .NET Framework with a smaller memory footprint. It is specifically targeting devices with limited resources, such as PDAs and mobile phones. These devices often run the Microsoft Pocket PC or Windows CE operating systems.

3.3.2 DotGNU Portable.NET

Portable.NET (PNET) is part of the DotGNU project and provides its CLI implementation. DotGNU aims at providing a fully compatible environment for running web services. PNET runs on several operating systems, including Linux, Windows, Solaris, FreeBSD, and Mac OS. It is licensed under the popular GPL.

Source languages are translated to bytecode by the Southern Storm Software's Tree Compiler, `treecc`. `treecc` is not only able to generate IL, but also Java bytecode.

When compiling CIL to native code during execution, Portable.NET takes a different approach than the other runtimes. Instead of directly compiling CIL to native code, another step is inserted. CIL is first compiled to Converted Virtual Machine (CVM), and only then CVM is compiled to native code via an interpreter. This is one of the aspects that makes the PNET runtime highly portable across different platforms.

Looking at the website[25] of the PNET project, it seems that development is discontinued – the last project update found on their website dates back to June 2009. Only a handful commits were made between 2009 and 2012 in the source code repository.

3.3.3 Open CLI Library

The Open CLI Library was created for the Open Runtime Platform by Intel. The Open Runtime Platform can host several distinct JIT compilers and was designed for research on managed runtime environments. Both Open CLI Library and Open Runtime Platform are licensed under an open-source license.

3.3.4 Shared Source CLI

The Shared Source CLI ("Rotor") is developed by Microsoft and Corel and was first released in 2002. It serves as a demonstration of .NET's cross-platform compatibility and provides a sample implementation of the CLI standard. The SSCLI provides a runtime (similar to Microsoft's CLR), a C# compiler, some extra class libraries, development tools, and research tools.

[25] http://www.gnu.org/projects/dotgnu/, DotGNU Project

It is licensed under a shared source license, which means it cannot be used for commercial products, but the source code is publicly available. A big portion of its code is shared with the commercial and proprietary Microsoft .NET Framework. As such, it provides good insights into how the .NET Framework works internally.

Chapter 4

Shortcomings and Incompatibilities

While Mono fully implements the Common Language Infrastructure specified in ISO/IEC 23271 and tries to be compatible with Microsoft's implementation of the standard, it cannot fully replace Microsoft .NET (yet). Additionally, Mono is a cross-platform project and aims to run on all common operating systems (Windows, Linux, Mac OS), and thus can only provide features available on all systems. Some features are restricted to certain environments, such as access to the Windows registry and P/Invoke to call native libraries.

This chapter highlights the most common issues developers might encounter when porting applications from .NET to Mono, and in turn from Microsoft Windows to Linux. The author describes simple solutions and gives general directions on how to resolve these problems, some of which are applied in the case-study in Chapter 5. Other concrete problems and their resolutions are also dealt with in the case study, in which the author describes the process of porting the HeuristicLab 3.3.6 framework to Mono and Ubuntu Linux.

This and the following chapter not only focus on implementations of the Common Language Infrastructure standard, but also discuss the ecosystems involved. The ecosystem includes third-party libraries, non-standard extensions, development tools and differences between operating systems, among others. These are all factors which influence the complexity of the task of porting an application from .NET to Mono.

For many of the problems it should be possible to avoid them in the first place, if an application is developed with cross-platform portability in mind, but this may not always be possible. Also, many .NET applications were written before the Mono project existed or before it was a viable alternative to .NET.

Most of the information from this chapter were aggregated from a number of websites, mostly the website and wiki of the Mono project[1], the personal blog of Mono founder Miguel de Icaza[2] [7], and the *Microsoft Developer Network* (MSDN)[3]. In addition, the source-code of the Mono project was used as a reference and to verify implementation details[4].

4.1 Mono Migration Analyzer (MoMA)

The Mono project provides a tool called the *Mono Migration Analyzer*, short *MoMA*[5]. MoMA analyzes .NET applications (`.exe`) and libraries (`.dll`) and generates a report about potential issues. This report helps to get an initial overview of which components might need fixing to be used with Mono.

MoMA is able to detect four potential problem sources and categorizes them as Missing Methods, MonoTODO, NotImplementedException, and P/Invoke[6].

- **Missing Method:** These methods are not implemented in Mono at all. Source code using these methods does not compile, and calling such methods results in an exception of type `MissingMethodException` being thrown.
- **MonoTODO:** Methods in Mono's source code might have a `MonoTODO` attribute attached. The attribute can also contain an optional description (`Reason`). `MonoTODO` attributes are usually a sign of small bits of missing functionality, but may also be attached to method stubs which do not perform any meaningful action. Code that references methods marked with `MonoTODO` will compile just fine. Running the compiled code should generally work without crashing, but may not produce the expected results.

  Some examples are:

```
1 [MonoTODO("Stub, does nothing")]
2 [MonoTODO("Only implemented for Win32, others always return
      false")]
3 [MonoTODO("Implented for Win32, X11 always returns 0,0")]
4 [MonoTODO("columnIndex parameter is not used")]
```

- **NotImplementedException:** Some methods are not actually implemented and their body only contains a single statement, which throws an exception of type `NotImplementedException`. Code that references these methods compiles, but calling them throws an exception. Some-

[1] http://mono-project.com
[2] http://tirania.org/blog
[3] http://msdn.com
[4] https://github.com/mono/, Mono Project
[5] http://www.mono-project.com/MoMA
[6] http://www.mono-project.com/MoMA_-_Issue_Descriptions

times only certain code paths of methods are affected and calling methods under certain conditions might work as expected.

- **P/Invoke:** Platform Invoke is a mechanism to call unmanaged code of native libraries (e.g. `user32.dll` or `libc`) from managed code. Oftentimes, P/Invoke calls cause problems, because native libraries usually only exist for a particular platform. Mono supports P/Invoke calls, provided the native library exists on the host system. An in-depth discussion of this topic and related problems can be found in Section 4.8

4.2 Windows Communication Foundation

Windows Communication Foundation (WCF) is a framework that provides high level communication between processes on the same system or across system boundaries. It was introduced as part of Microsoft .NET 3.0. WCF can communicate over a variety of channels, such as named pipes, TCP, and HTTP. *Bindings* provide additional functionality on top of the standard protocols in the form of transaction and session support, among others.

Bindings without support for transactions include `BasicHttpBinding`, `NetMsmqBinding`, `NetPeerTcpBinding`, and `WebHttpBinding`. The default bindings `BasicHttpBinding`, `NetNamedPipeBinding`, and `NetPeerTcpBinding` do not support reliable sessions. *Reliable sessions* guarantee that messages are received in correct order and that each message is received exactly once [36].

A WCF endpoint is configured by setting three main properties: *address*, *binding* and *contract*. A sample configuration of a service endpoint is shown in Listing 4.1.

Listing 4.1: WCF Endpoint Configuration[7]

```
<configuration>
 <system.serviceModel>
  <services>
   <service name="Microsoft.ServiceModel.Samples.CalculatorService">
    <endpoint address="net.tcp://localhost:9000/sample/service"
     binding="netTcpBinding"
     contract="Microsoft.ServiceModel.Samples.ICalculator" />
   </service>
  </services>
 </system.serviceModel>
</configuration>
```

Most of the classes which provide WCF's functionality can be found in the `System.ServiceModel` namespace. The assembly `System.ServiceModel.Web` provides a number of classes which help in building REST Web Service [36]. Mono supports the basic functionality of the `System.ServiceModel` assembly and simple bindings of the WCF framework.

[7] http://msdn.microsoft.com/en-us/library/ms734786.aspx, How to: Create a Service Endpoint in Configuration

One of the main problems with Mono's WCF implementation is the lack of the `WS-*` bindings[8]. These bindings are necessary to add security aspects (e.g. authentication, authorization) and transaction support to the basic HTTP bindings. Trying to create an endpoint with one of these bindings under Mono throws an `InvalidOperationException`. There are currently no plans to add support for these bindings to Mono.

A modern alternative to WCF that is supported by both .NET and Mono is *ServiceStack*[9]. ServiceStack does not require XML files for configuration and does not rely on generated code, but requires strongly typed DTOs (Data Transfer Objects) for type inference. Speed and simplicity are also one of the main goals of the ServiceStack project.

4.3 Windows Presentation Foundation

Windows Presentation Foundation (WPF) was first introduced in .NET 3.0 and is the successor of Windows Forms (discussed in Section 4.5) [28]. It is a new presentation framework and combines features of existing frameworks, such as GDI, HTML and Adobe Flash [34].

One of its main advantages is the separation of actual code and layout descriptions. WPF allows developers to describe the layout of widgets declaratively through means of XAML [28]. XAML is discussed in Subsection 4.3.1.

WPF is currently only implemented in Microsoft .NET and is not supported by Mono. According to the Mono project's website, there are currently no plans to work on an implementation within Mono: "The project [WPF] is too large and there has not been any serious interest from the community to make this effort move forward"[10]. The cost of implementing the complete WPF stack within Mono was estimated in March 2011 to be 2 – 3 years worth of work for 15 – 20 developers[11]. A subset of the WPF API is already implemented in *Moonlight*, the open-source implementation of Silverlight. However, the development of Moonlight was discontinued in May 2012[12].

Developers who want to run and compile their applications with Mono should keep that in mind and avoid WPF in favor of Windows Forms or one of the alternatives listed in Section 4.5.

[8] http://www.mono-project.com/WCF_WSHttpBindingHacking

[9] http://servicestack.net/, Open source .NET and Mono web services framework

[10] http://www.mono-project.com/WPF, http://www.mono-project.com/WPF_Notes

[11] http://tirania.org/blog/archive/2011/Mar-07.html, GDC 2011

[12] http://www.infoq.com/news/2012/05/Miguel-Moonlight, Miguel de Icaza on ASP.NET MVC, Moonlight, and the Android Lawsuit

4.3.1 XAML

XAML (Extensible Application Markup Language) was introduced as part of the Windows Presentation Foundation stack to enable developers to define user interfaces declaratively in XML. Elements directly represent classes and attributes are used to access properties of these classes. Every XAML document could be converted into fully procedural code. In contrast to e.g. HTML, XAML is strongly typed. Trying to declare elements of the wrong type results in compilation and runtime errors [25].

XAML is very concise for describing UIs and general object hierarchies. See Listing 4.2 for a simple XAML snippet which declares a page containing a labeled button. These hierarchies can be arbitrarily complex.

Listing 4.2: Simple XAML Document[13]

```
1 <Page
2   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4   x:Class="ExampleNamespace.ExamplePage">
5   <Button Click="Button_Click" >Click Me!</Button>
6 </Page>
```

While WPF is not supported by Mono (see Section 4.3 above), XAML is mostly implemented within Mono. `XamlObjectReader`, `XamlObjectWriter`, `XamlXmlReader`, and `XamlXmlWriter` are implemented to read and write objects and XML files, respectively[14]. An up-to-date overview of the current status of Mono's XAML implementation can be found on the Mono Class Status Pages[15].

4.4 Windows Workflow Foundation

Windows Workflow Foundation (WF) allows developers to design and build workflow enabled applications[16]. It was first introduced in .NET 3.0 together with WPF and WCF. The .NET 4.0 platform update 1 added support to properly implement state machines with WF.

Windows Workflow Foundation is currently not supported by Mono[17]. Six out of nine major features are currently unimplemented, and three features are only implemented partially. Other open-source alternatives for

[13] http://msdn.microsoft.com/en-us/library/ms752059.aspx, XAML Overview (WPF)

[14] http://www.mono-project.com/SystemXamlHacking

[15] http://go-mono.com/status/status.aspx?reference=4.0&profile=4.0&assembly=System.Xaml

[16] http://msdn.microsoft.com/en-us/netframework/aa663328.aspx, Windows Workflow Foundation

[17] http://www.mono-project.com/Workflow

modelling state-machines or state-machine based workflows include stateless[18] and Jazz[19].

4.5 Windows Forms

When the Mono project was started, applications developed for .NET primarily relied on the *Windows Forms* library for creating and interacting with GUI elements. Mono provides its own implementation of Windows Forms and the classes of the `System.Windows.Forms` namespace. The Mono implementation emulates the look and feel of Windows application, but draws all widgets with methods from the `System.Drawing` assembly. This ensures the same functionality and appearance on all operating systems.

Much effort was put into making Mono's Windows Forms implementation compatible with Microsoft's implementation in .NET as much as possible. The Mono implementation of Windows Forms works well enough for simple GUIs and can be used without problems most of the time. However, there are still rough edges in the implementation provided by Mono.

Parts of Mono's Windows Forms implementation, that would benefit from some additional work, are the types related to DataGrids and DataGridViews. Some methods are only dummy implementations that return a constant value. For instance, `GetPreferredWidth()` from the class `System.Windows.Forms.DataGridViewColumn` always returns 0, a value that is not a valid width. Assigning it to the width property throws an exception of type `System.ArgumentOutOfRangeException` during runtime. A patch for this bug was already sent to the Mono developers and should be included in the next Mono releases.

Alternatives to Windows Forms include the multi-platform GTK+ (for instance used by MonoDevelop), Qt, or other GUI toolkits with bindings for .NET and Mono. GTK+ is accessible from CLI implementations through the GTK# library which provides bindings for CLI implementations.

"Cross-Platform .NET Development" [10] contains helpful pointers for using the MVC (Model-View-Controller) pattern to write GUI-toolkit agnostic source code. An overview of popular GUI toolkits and their level of support for different platforms is provided as well.

4.5.1 Charting

The Windows Forms namespace contains a sub-namespace, `System.Windows.Forms.DataVisualization`, which can be used to draw and manipulate interactive charts and diagrams. This library is also known as *Microsoft Chart Controls*. It is currently not implemented in Mono, with many classes and

[18] http://code.google.com/p/stateless/, stateless – A C# Hierarchical State Machine
[19] http://jazz.codeplex.com/

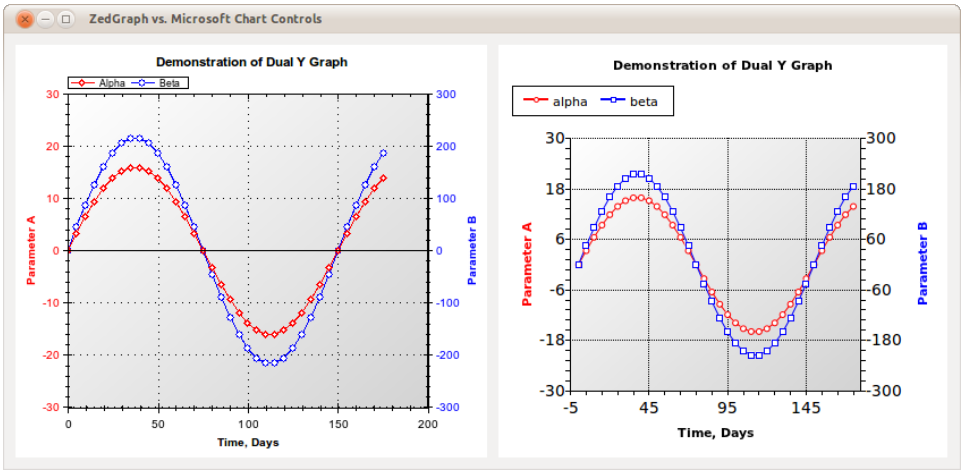

Figure 4.1: ZedGraph (left) vs. Microsoft Chart Controls (right)

methods missing from the assembly or throwing exceptions of type `System.NotImplementedException`.

Depending on the requirements of the project, the open-source project ZedGraph[20] might be used as an alternative to Microsoft Chart Controls. ZedGraph allows developers to draw different types of charts with zoom and pan functionality. Figure 4.1 shows a simple plot generated with ZedGraph side by side with the same data points plotted with Microsoft's `System.Windows.Forms.DataVisualization.Charting` library. ZedGraph development seems to have stalled, since the last revision in ZedGraph's Subversion repository was committed more than three years ago (December 12, 2008).

4.6 Windows Registry

This section discusses how to store and retrieve information in and from the Windows registry with applications written for .NET and Mono. Developers and users are very likely to encounter problems when applications are run on different operating systems. These problems are not directly related to the chosen runtime, but depend on the underlying operating system.

The Windows registry exists, for obvious reasons, only on Windows operating systems. Applications relying on Windows registry will hence not work without alterations on other operating systems. Mono is capable of accessing the Windows registry on Windows operating systems. On other operating systems, Mono emulates the Windows registry: Keys and subkeys are represented by directories and values are stored in XML files. It is possible to store values in the registry and to retrieve them at a later date.

[20] http://sourceforge.net/projects/zedgraph/

It is however not possible to query the registry for information about the operating system [10].

When trying to open non-existent registry keys on other operating systems besides Microsoft Windows, an exception of type `System.Security.SecurityException` is thrown. This is due to the fact that Mono tries to create directories for missing keys on the fly, but fails since the location is usually not writable by normal users.

An approach that does not rely on the Windows registry for storing application configuration data is the use of assembly configuration files (usually `app.config`). They enable developers to store information in XML format on a per-assembly, per-machine, and per-organization basis [10]. Assembly configuration files are simply stored in the file system together with the actual assembly file. Settings stored in these files can be accessed through classes of the `System.Configuration` namespace.

Other approaches include storing information in flat text files or standalone databases, such as SQLite.

4.7 Security

CLI Implementations can use the CLI Metadata to provide *Code Access Security* (CAS) [20]. CAS allows to grant or deny a set of permissions for code, which are checked during execution. Code access security is an implementation detail and is not required to be present in implementations of the CLI.

.NET's code access security model has undergone quite some change with version 4 of the framework[21]. Applications can either be fully trusted or partially-trusted. Local (desktop) applications are always executed with full trust; sandboxed (hosted) applications are partially-trusted and can only run with restrictions. Code in .NET falls into three categories: transparent, critical, and safe-critical code. Transparent code is the most restricted code and cannot contain unverifiable code, call native functions with P/Invoke, or call code categorized as critical, among others. Critical code is the least restricted and can practically do anything. Safe-critical code plays a mediator role, as it can be called from transparent code yet still call critical code itself.

CAS was marked as experimental in Mono 1.2 (released in 2006), with no release date planned[22]. Not much work has been put into code access security since, and the current state is still comparable to what it was then. A subset of code access security however was implemented with Moonlight to support Silverlight's security model.

[21] http://msdn.microsoft.com/en-us/library/ff527276.aspx, Summary of Changes in Code Access Security

[22] http://www.mono-project.com/CAS

4.8 Platform Invoke

Platform Invoke (often shortened to *P/Invoke*) is a mechanism in the Common Language Infrastructure to allow applications to use native libraries. One of the main reasons this feature exists is the amount of COM (Component Object Model) legacy code [31].

Different operating systems use different formats for native libraries. Windows uses `.dll` (Dynamic Link Library) files, Mac OS uses `.dylib` (Dynamic Library) files and Linux uses `.so` (Shared Object) files. Each of these file formats can only be executed on the operating system and architecture they were compiled for.

Managed types are *marshaled* during execution to and from their unmanaged counter parts, when calling a native function made accessible with P/Invoke. If a defined entry point is not found, an exception of type `System.EntryPointNotFoundException` is thrown during execution. This means that applications that rely on Platform Invoke compile without problems with Mono's compiler, but are very likely to crash during execution. They will definitely crash when trying to execute a P/Invoke instruction for a library that is not available on the current system.

Common assemblies that rely on unmanaged code are listed for each CLI implementation in chapter 7 and in Appendix B of "Cross-Platform .NET Development", together with fully managed assemblies [10]. The website http://pinvoke.net collects information on P/Invoke signatures for many unmanaged libraries, including alternative managed APIs where available.

When there are multiple versions of the same library for multiple operating systems, it is possible to use the same P/Invoke calls across all operating systems. In that case the Mono project recommends to declare `DllImports` without a file extension to allow the runtime to pick the correct library, depending on the operating system[23]. This increases maintenance costs, since developers have to maintain multiple variants of the same library. Also, bugs could exist on one operating system, but not on the other.

On the other hand, developers of an application can choose to write wrappers around P/Invoke calls which determine the current runtime and choose the adequate implementation. Easton and King show how to apply object-oriented design patterns to choose the correct version of the library during execution [10]. The maintenance cost of these first two options are comparable, since developers have to maintain multiple version of the same native library.

Another solution is to avoid native libraries altogether and develop applications exclusively with managed code. This is often not possible to implement or not feasible: Developers would have to replace all native functions with managed code counterparts. This is easier, if the native library was

[23] http://www.mono-project.com/Interop_with_Native_Libraries

written by the same team or developers who are working on the managed part of the code base. For external (third party) libraries, this rarely is an option.

4.9 Case Sensitive File Systems

Incompatibilities due to case sensitive paths are not directly a problem of Mono, but a consequence of its multi-platform nature. Mono runs on different platforms and operating systems and these systems all support different types of file systems.

On Windows with .NET the most prominent file system currently is NTFS. NTFS must support case-sensitive filenames to be POSIX compliant[24]. Although NTFS supports path names that only differ in capitalization, Windows does not allow to create such files, but it preserves the capitalization of filenames when creating files and directories. Third party applications might allow creation of and access to files with the same name but different capitalization, a good example is the open-source driver *ntfs-3g*[25] for NTFS. Windows normally accesses files case-insensitively on NTFS volumes and is only able to access one of multiple paths that differ only in capitalization. MAC OS's file system, HFS+, is case-insensitive by default. Restrictions similar to those of NTFS apply.

In contrast to the above, Linux based file systems such as ext4 are case-sensitive. Paths are considered unique if they differ in name or capitalization. When accessing files, the exact name must be given. Trying to access a path whose capitalization does not match exactly that of the path on disk results in an error.

Applications designed only with case-insensitive file systems in mind might consequently reference files that do not exist when working with case-sensitive file systems. This often happens by accident: path names are either written manually and no attention is given to capitalization, or a file is renamed which results in the same name but different capitalization.

Mono offers several options to make migration to other file systems relatively painless. Behavior can be controlled with the `MONO_IOMAP` environment variable. A special profiler (`--profile=iomap`) of Mono traces all re-mappings of file paths and gives a detailed report after execution. This information aids developers immensely to detect incorrect paths spread throughout the source code of a project. Even though Mono has the option of folding the case of path names, the better option for developers is to use the proper case when referencing files. This guarantees that an application works across all

[24] http://support.microsoft.com/kb/100625, Filenames are Case Sensitive on NTFS Volumes

[25] http://www.tuxera.com/community/ntfs-3g-manual/, NTFS-3G Manual

file systems, regardless of the compatibility options specified when executing the application.

Another aspect that distinguishes Windows from other operating systems with regard to file system usage is the directory delimiter. While other operating systems (including Linux and Mac OS) use a forward slash `/` to delimit directories, Windows uses the backslash `\`. Fortunately, Mono correctly translates directory delimiters depending on the platform. Windows properly handles forward slashes in paths too. Hence, directory delimiters need not be fixed for CLI applications to be compatible with Windows and other operating systems.

4.10 Compiling Files and Building Projects

Projects written for the .NET Framework are managed with Visual Studio most of the time. Visual Studio makes use of solution and project files to store project specific configuration options and track dependencies between files. MonoDevelop offers read and write support for Visual Studio solution and project files. Problems arise when project files make use of custom build events.

Microsoft Visual Studio allows developers to define actions which are automatically executed at certain points during the build process. For instance, actions might be triggered before starting the build or after the build process has completed. These build events are usually used to create source files from templates files (skeleton, frame) or to perform additional operations on the build products after the compilation.

Build scripts were historically only written for the Windows operating system, because Microsoft .NET does not run on other operating systems. As such, most of the time they were either Windows executable files (`.exe`) or batch scripts (`.bat`). For some executable files it might be possible to execute them with WINE[26], however, this is not possible for batch files.

A simple solution would be to provide different versions of the solution and project files, referencing the proper build scripts for each target system, e.g. batch files on Windows and shell scripts on Unix-like systems. This requires multiple versions of project files and build scripts to be maintained concurrently. The required effort and maintenance increases directly proportionally with each new system added.

A better solution to this problem is to re-write all build scripts once in a language which is available across multiple operating systems. Scripting languages are a good choice in this regard, since interpreters often exist for all common operating systems. Valid choices of scripting languages include *python*, *ruby*, *perl*, and others. The same build script can then be used across all build platforms.

[26] http://www.winehq.org/

Another possibility which is independent from Visual Studio project files is the management of the build process with a build tool such as *NAnt*. NAnt is the .NET version of the Java build tool *Ant* which uses XML files to define dependencies between files and tasks to execute. It runs on all major operating systems and has support for .NET and Mono. An in-depth explanation on how to properly use NAnt for cross-platform projects can be found in "Cross-Platform .NET Development" [10]. An example of a simple NAnt build script can be seen in Listing 4.3.

Listing 4.3: Simple NAnt Build Script

```
<?xml version="1.0"?>
<project name="simple" default="build">
  <target name="build">
    <csc target="exe" output="simple.exe"
      <sources>
        <include name="simple.cs" />
      </sources>
    </csc>
  </target>
</project>
```

4.11 Unit Tests

Several unit testing frameworks exist for the .NET Framework and other CLI implementations. Popular options include the MSTest framework, NUnit, xUnit.net, and MbUnit. While MSTest is only supported by the .NET Framework, the other unit testing frameworks work well with different CLI implementations such as Mono.

This thesis focuses on NUnit as the primary unit testing framework for .NET and Mono projects. It was originally ported from JUnit and is available for all common languages of the CLI (C#, Visual Basic.NET, ...), which makes it a good cross-platform alternative to MSTest.

MonoDevelop offers NUnit integration by default, and an extension for Visual Studio exists as well[27]. Direct integration into the IDE enables developers to quickly navigate through test cases and to locate failing code. Figure 4.2 shows MonoDevelop's unit test runner for NUnit projects.

Unit test classes and methods are marked with attributes, signalling which methods to execute during the test phase. It is also possible to specify methods that are executed as common startup or shutdown routines.

Simple MSTest unit tests can often be transformed into NUnit tests with little effort by applying text transformations to the source files. On Linux systems, GNU sed comes in handy. Listing 4.4 shows a simple sed

[27] http://visualstudiogallery.msdn.microsoft.com/c8164c71-0836-4471-80ce-633383031099, Visual Nunit 2010 extension

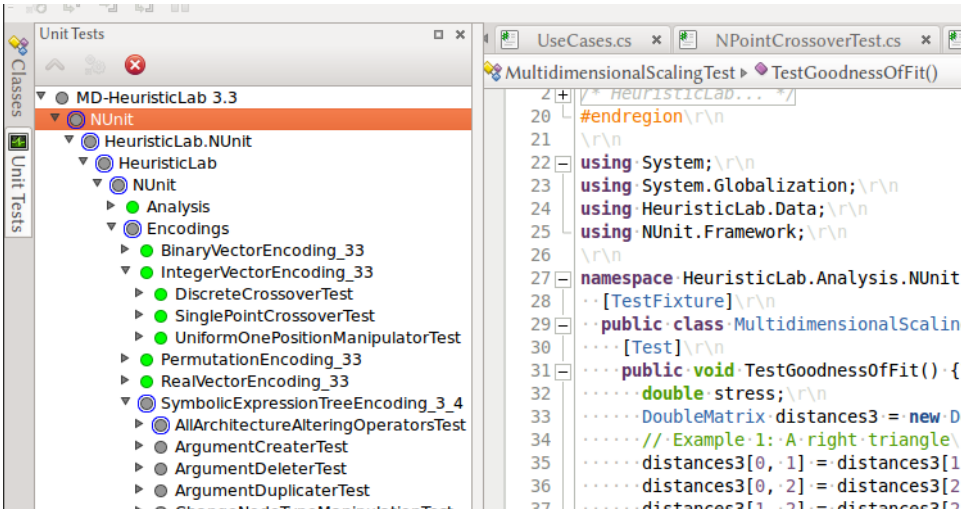

Figure 4.2: NUnit Test Pane inside MonoDevelop

invocation to replace common MSTest attributes and `using` statements with the corresponding types and namespaces of the NUnit framework.

Listing 4.4: Replacing MSTest Attributes with their NUnit Counterpart

```
sed \
-e 's/^using Microsoft.VisualStudio.TestTools.UnitTesting;/using NUnit.
    Framework;/; # use nunit' \
-e 's/\[TestClass\(()\)\?\]/[TestFixture]/; # mark test classes' \
-e 's/\[TestMethod\(()\)\?\]/[Test]/; # mark test methods' \
-e 's/\[ClassInitialize\(()\)\?\]/[TestFixtureSetUp]/;  # mark setup
    methods'
```

This will not work for all test attributes and some attributes have to be inspected manually and replaced by a suitable attribute available in NUnit. Several books cover unit testing with NUnit [10, 17]. The complete and up-to-date documentation can be found on the NUnit website (http://nunit.org).

4.12 Internals

This section deals with differences in distinct runtimes. The two runtimes that are discussed are Microsoft's runtime (CLR) shipped with .NET and the Mono runtime. Usually developers and users should not be concerned about implementation details, because they are abstracted away. Still, rare cases exist, where they could cause problems.

Problems can occur when serializing objects in one and deserializing them in another implementation of the CLI. Serialized object graphs might contain references to internal types which are only available in a specific implementation. For instance, collections often allow to specify comparer

objects to compare keys or values. An example is `ObjectEqualityComparer` which is used by `System.Collections.Generic.Dictionary<TKey,TValue>` under .NET but does not exist in Mono.

Another problem related to serialization is the internal representation of runtime types. Internal types are an implementation detail and are not specified by the Common Language Infrastructure. Microsoft .NET uses `RuntimeType`, whereas Mono uses `MonoType`, both of which can be found in the `System` namespace and derive from the abstract `System.Type`. These types are generally not directly visible to application code, but are part of serialized objects.

Different implementations of the garbage collector, which exhibit different collection patterns, might cause problems as well. Usually, the different GCs should not have a functional impact on applications. There might be race conditions, however, that only show with a particular GC configuration. These errors must be fixed in the application, but they can be very hard to track down.

Chapter 5

Case Study – HeuristicLab 3.3

Over the course of this master thesis, a case study on the practicability of porting HeuristicLab to Mono and Linux was conducted. The following sections give the reader an overview over HeuristicLab's general architecture and what components and technologies are used by the HeuristicLab framework.

Thereafter, incompatible code, data, and components of the HeuristicLab project are identified. In each section, the changes required to compile and execute HeuristicLab under Mono are described. The majority of problems in these sections are already familiar to the reader, who was introduced to potential problem areas in the chapter Shortcomings and Incompatibilities. This time, the aforementioned solutions are applied to a concrete project.

Not all components could be converted to be fully compatible with Mono. This is mentioned in the respective sections, with possible further steps to take. Some fixes are required in the source code of the Mono project itself too.

5.1 About HeuristicLab

HeuristicLab is a paradigm independent and extensible environment for heuristic optimization and data analysis and was initiated by Prof. (FH) Priv.-Doz. DI Dr. Michael Affenzeller and Prof. (FH) DI Dr. Stefan Wagner in 2002 [39, 38]. It has since been developed by the team of the *Heuristic and Evolutionary Algorithms Laboratory* (HEAL)[1]. HeuristicLab is an open-source project and is licensed under the GNU General Public License (GPL). One of the project's main goals is to provide its users with an intuitive graphical user interface to create, modify and use meta-heuristic

[1] http://heal.heuristiclab.com

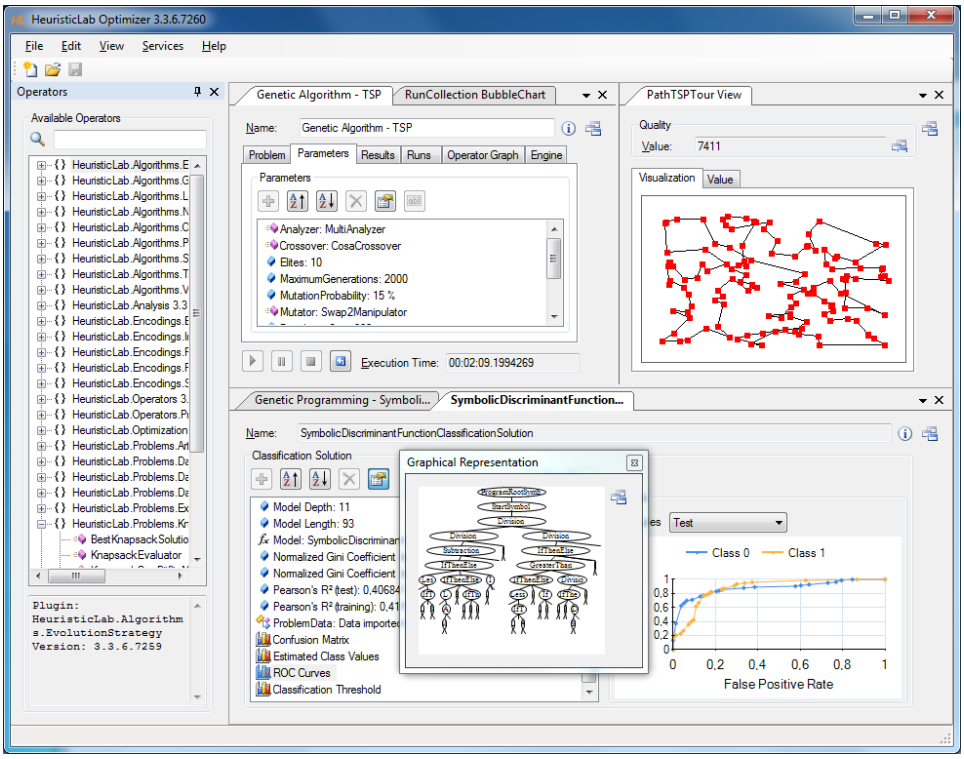

Figure 5.1: Screenshot of the HeuristicLab 3.3.6 Optimizer under .NET on Windows

optimization algorithms, even for users without special training in software development.

Figure 5.1 shows a screenshot of the *Optimizer* plug-in of HeuristicLab 3.3.6 under .NET on Windows. The window contains different views for a Traveling Salesman Problem and a Symbolic Classification algorithm. A look and feel similar to that of Microsoft Visual Studio is provided and views can be arranged freely. Charts are able to provide additional visual representations of certain parameters.

Users can construct algorithms graphically by using drag and drop and customize almost every detail of an algorithm to suit their needs. A flexible plug-in infrastructure enables developers and users to extend HeuristicLab without re-compiling the complete source code.

Algorithms and problems are orthogonally independent and can be freely combined. This enables researchers to remain highly flexible when searching for suitable algorithms for their problems.

The current release of HeuristicLab offers its users a lot of predefined problems and algorithms. The following list should give a quick glance at common problems that can be optimized with HeuristicLab:

- Traveling Salesman Problem (TSP)
- Vehicle Routing Problem (VRP)
- Quadratic Assignment Problem (QAP)
- Knapsack Problem
- Several data analysis problems: Clustering, Single- and Multi-Objective Regression, Single- and Multi-Objective Symbolic Classification

Every problem can be optimized with any of the available algorithms, if appropriate manipulation operators are provided. Algorithms contain an *operator graph* which defines the order in which operations are applied to a problem. Some algorithms provided with the default installation of HeuristicLab 3.3 are:

- Evolution Strategy
- A set of genetic algorithms: a classical Genetic Algorithm, Offspring Selection Genetic Algorithm, Island Genetic Algorithm, Island Offspring Selection Genetic Algorithm, NSGA-II, SASEGASA
- Local Search
- Particle Swarm Optimization
- Simulated Annealing
- Variable Neighbourhood Search
- Tabu Search and Robust Tabu Search
- Various data analysis algorithms: Nearest Neighbour, Neural Network, Random Forest, Support Vector Machines

Algorithms are executed by one of the available *engines*, which can be chosen by the user before running an algorithm. Engines are responsible for executing operations from the operator graph that is provided by an algorithm. HeuristicLab currently offers three distinct engines:

- **Sequential Engine:** This is the basic engine which simply executes an algorithms operator graph in a sequential fashion. Algorithm performance does not increase when this engine is used on a multi-core system.
- **Parallel Engine:** The parallel engine can take advantage of multiple CPUs on a single system and can improve performance significantly. This engine is based on the Microsoft *Task Parallel Library* (TPL). Parallel instructions are modelled with special *parallel operators* in the operator graph.
- **Debug Engine:** The debug engine was introduced to help developers and algorithm designers to precisely monitor and control algorithm execution. Algorithms can be single stepped with this engine and their state can be inspected at any time.

5.2 Architecture

This section outlines HeuristicLab's general architecture on a technical level. Having a picture of the overall architecture helps with pinpointing critical components and identifying potential problems that might arise during the porting phase.

HeuristicLab is written in C# and currently requires the .NET Framework 4.0. It utilizes Windows Forms to provide a graphical user interface. The default docking interface is implemented using the third party DockPanel Suite[2]. HeuristicLab is built using a modular approach and can be extended by writing plug-ins. The plug-ins are loaded dynamically with the reflection mechanism.

Figure 5.2 schematically shows HeuristicLab's core components and their relationships. For a detailed overview, refer to the documentation, which can be found in the HeuristicLab source code repository[3].

5.2.1 External Libraries

HeuristicLab uses a number of external, third-party libraries to provide common functionality. Following is a list of the libraries which are shipped as part of HeuristicLab's source code, but were not developed by the HeuristicLab team:

- **ALGLIB**[4] is a portable library for numerical analysis and data processing. HeuristicLab uses ALGLIB for its *DataAnalysis* plug-in.
- **DayView Calendar** offers a Windows Forms control which presents its users with a calendar, similar to that of Microsoft Outlook, to visualize scheduling events[5]. HeuristicLab uses the DayView Calendar Control in its Hive Admin Client.
- **LibSVM** is a library for *Support Vector Machines*[6]. It is used for vector classification, regression and distribution estimation.
- **Apache log4net**[7] provides a logging infrastructure for .NET applications. log4net is a .NET port of the popular *Apache log4j* Java project.
- The **Netron Diagramming Library** provides a graphical control for Windows Forms, which allows users to create and manipulate networks of objects. Users of HeuristicLab use it when working with operator

[2] http://dockpanelsuite.sourceforge.net/
[3] http://dev.heuristiclab.com/trac/hl/core/browser/trunk/documentation
[4] http://www.alglib.net/
[5] http://calendar.codeplex.com/
[6] http://www.csie.ntu.edu.tw/~cjlin/libsvm/, LIBSVM – A Library for Support Vector Machines
[7] http://logging.apache.org/log4net/

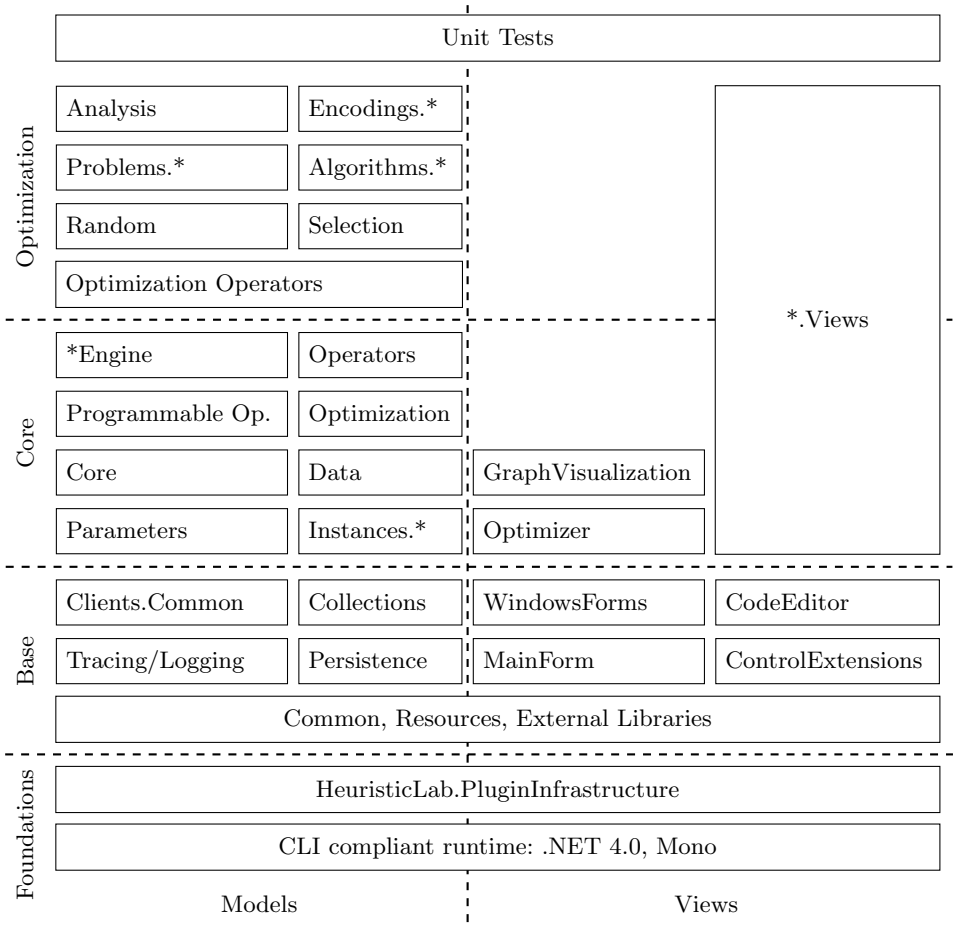

Figure 5.2: Architectural overview of the HeuristicLab 3.3 optimization framework

graphs of algorithms. A project website does not exist anymore and the project was officially discontinued[8].

- **ProtoBuf** is a popular library by Google to implement efficient protocols in an object-oriented way[9]. It is used by HeuristicLab to offer interoperability with external systems.
- **SharpDevelop.TextEditor** is part of the SharpDevelop IDE and was used as the text editing component within SharpDevelop. It has been replaced with AvalonEdit in SharpDevelop 4. HeuristicLab uses `SharpDevelop.TextEditor` to provide syntax highlighting when defining an programmable operator (`HeuristicLab.Operators.Programmable`).

[8] http://visualizationtools.net/default/so-netron-is-back-online/, So Netron is back online?!

[9] https://developers.google.com/protocol-buffers/

- **DockPanel Suite**[10] provides a graphical user interface with docking capabilities for Windows Forms, similar to those of Microsoft Visual Studio. HeuristicLab uses the DockPanel Suite library in its default graphical front end of its optimizer, namely *DockingMainForm*.

5.3 Required Packages

The system used to port HeuristicLab to Mono was Ubuntu Linux 12.04 ("Precise Pangolin"), with Mono and MonoDevelop built from the master branch of the official Mono source code repository to have the latest features and fixes available.

The following list contains the packages that are necessary to successfully compile and execute HeuristicLab under Mono on Linux. Each package can be installed with Ubuntu's package manager, unless noted otherwise:

- **Mono** Mono runtime, Mono compiler, and Mono core libraries. The current version, obtainable from the Mono Git repository, is 2.11.x. The steps required to compile and install Mono from source are outlined in the next section.
- **MonoDevelop** The *MonoDevelop* IDE is recommended for developing and building HeuristicLab. MonoDevelop can be installed with Ubuntu's package manager, but it is advisable to build it from source as well. It also provides an integrated test runner for NUnit test files (see below).
- **protoc** *protoc* (Protocol Buffer compiler) is required for compiling Google Protocol Buffer descriptions from `.proto` files to actual source code.
- **NUnit** The *NUnit* test suite is required to run the unit tests provided by HeuristicLab. NUnit is a cross-platform unit testing framework and was chosen as a replacement for the MSTest framework, previously used by HeuristicLab.
- **SubWCRev** HeuristicLab relies on dynamically generated source files where placeholders get replaced by their revision number from the source control system. This ensures that plug-in assemblies built from different revisions can be distinguished by their assembly version number. HeuristicLab makes use of the command line program *SubWCRev* for this purpose, which is included in the Windows Subversion client *TortoiseSVN*. A port, *svnwcrev*, is available for Linux systems[11]. Alternatively, *SvnRev*[12] can be used on Linux.

[10] http://dockpanelsuite.sourceforge.net/

[11] http://svnwcrev.tigris.org/

[12] http://www.compuphase.com/svnrev.htm, SvnRev: a utility for Subversion and CVS/RCS

- **Other libraries** A number of other packages is required for HeuristicLab to run in Ubuntu. When using the Mono package provided by Ubuntu's package manager, these dependencies are automatically installed. Additional packages that need to be installed on a default installation of Ubuntu are: libpango1.0-dev, libgtk2.0-dev, libmono-cairo2.0-cil, libmono-winforms2.0-cil, and libgdiplus.

Only the Mono runtime with its core libraries and the libraries mentioned in *Other Libraries* are required to run HeuristicLab. All other packages and programs are only required for developers who want to build HeuristicLab from source.

5.3.1 Building Mono

This section shortly describes the steps that are required to build Mono from source and how to install the prerequisites. Building from source not only enables the latest features and bug fixes, but also allows to apply custom patches to the Mono source code, that might be necessary while initially porting an application.

To automate the build process of Mono and additional libraries and tools (such as the MonoDevelop IDE), a bash script[13], which can be found on GitHub, was utilized. The script install most of the build dependencies automatically, but some packages had to be installed manually.

Packages required to download and build Mono's source code are `git`, `autoconf`, `automake`, `libtool`, `build-essential` and `mono-gmcs`. `mono-gmcs` is required for the so-called *bootstrapping*, since parts of Mono require a Mono compiler to be present. Other packages necessary to compile the basic parts of Mono are `libglib2.0-dev`, `libpng12-dev` and `libx11`.

After installing these packages, the bash script should complete without errors. The installer script installs Mono in a separate directory and even allows to install and use different versions of Mono concurrently.

5.4 MoMA Report

Reports generated by MoMA help developers to get an initial overview of assemblies which use features that are unsupported by Mono. All analyzed HeuristicLab assemblies were taken from the HeuristicLab 3.3.6 release, available in the download section of the HeuristicLab website[14]. The Mono Migration Analyzer is presented in Section 4.1 of this thesis.

Since HeuristicLab requires a recent version of Mono that supports the .NET 4.0 features, it was necessary to create and add a new definition file that contains an up to date list of supported classes and methods. Analyzing

[13] https://github.com/firegrass/mono-installer-script
[14] http://dev.heuristiclab.com/trac/hl/core/wiki/Download

the assemblies with MoMA definitions for version 2.11 of Mono finds 1214 missing methods, 36 P/Invoke calls, 16 methods throwing NotImplementedExceptions, and 173 methods marked with `[MonoTODO]`. 72 assemblies pass the initial analysis and 36 assemblies are likely to cause problems with this version of Mono.

MoMA identified potential problems in the following HeuristicLab assemblies. Assembly names starting with "HeuristicLab." were shortened to "HL." to improve readability.

- **MonoTODO** Multiple assemblies call methods that are marked with the `MonoTODO` attribute:
  - HL.Data.Views-3.3
  - HL.Operators.Programmable-3.3
  - HL.PluginInfrastructure-3.3
  - ICSharpCode.NRefactory
  - ICSharpCode.SharpDevelop.Dom
- **P/Invoke** A small number of assemblies are identified by MoMA to call unmanaged, native code (P/Invoke). An in-depth discussion of these assemblies can be found in Section 5.5.1.
  - HL.MainForm.WindowsForms-3.3
  - ICSharpCode.SharpDevelop.Dom
  - ICSharpCode.TextEditor
  - WeifenLuo.WinFormsUI.Docking-2.3.1
- **Method Missing** MoMA reports several assemblies in the category "Method Missing". These assemblies reference the external Microsoft Chart Controls library, which is not fully implemented within Mono. The workaround used for the initial prototype of HeuristicLab is presented in Subsection 5.5.2.
  - HL.Analysis.Views-3.3
  - HL.Optimization.Views-3.3
  - HL.Problems.DataAnalysis.Symbolic.Regression.Views-3.4
  - HL.Problems.DataAnalysis.Views-3.4
  - HL.Visualization.ChartControlsExtensions-3.3

Furthermore, some assemblies were listed in the MoMA report, caused by methods being marked with the `MonoTODO` attribute and the comment "Implement it properly once 4.0 impl details are known". These methods are used to test `System.Type` types for equality and inequality. Currently, Mono only tests identity and compares the addresses of both arguments (`Object.ReferenceEquals`), but that has not caused any problems for HeuristicLab. Hence, assemblies reported due to this attribute are ignored in the further discussion.

5.5 Necessary Changes

This and the following sections present problem areas that have been identified while porting the HeuristicLab optimization framework to Mono and Linux.

Problems can be split into two major groups: compile time problems and runtime problems. Compile time problems are often easier to find, since they make the compilation of the source code impossible. Runtime problems on the other hand, can remain unrecognized for a long time, because they are only discovered by thorough testing or by accident. Good unit tests can help to discover runtime problems much earlier.

5.5.1 Fixing Platform Dependencies

This subsection deals with problems that are not directly related to the different implementations of the CLI standard, but problems which occur when working with different operating systems or different toolchains. Examples are differences in file systems, incompatibilities between IDEs, or non-portable testing frameworks.

Case Sensitive File Paths

In Chapter 4 an overview of the potential problems when running applications across different file systems was given. Different file systems use different directory delimiters or handle case sensitivity of file names differently.

The project files for HeuristicLab contained some file paths with different capitalization compared to actual file on disk. This is quickly discovered during compilation, because the compile process will abort due to non-existent files. Resource files also contain file paths that need to be corrected before the build process can complete successfully.

As mentioned in the introductory paragraphs, compile time errors are relatively easy to find. Some paths of HeuristicLab are only looked up during program execution. One such case is the dependency list and external file list of plug-ins. Required, external files are declared by attaching custom attributes of type `PluginFileAttribute` to classes which are evaluated when loading a plug-in. This is not an error per-se, but HeuristicLab will be unable to find all required dependencies or files, and consequently fail to load the plug-in.

To find plug-in file paths with incorrect capitalization, a temporary hook was added in the method that is responsible for loading plug-ins together with their referenced files. By default, HeuristicLab simply ignores plug-ins with inexistent, referenced files. Here, HeuristicLab could be extended to be more verbose or write a log file when a plug-in is not loaded due to referenced but missing files.

Build Events and Scripts

HeuristicLab uses build event scripts to create actual source code files, that contain plug-in interface definitions, from `.cs.frame` files, by running the `SvnRev` utility before compilation is started. The same procedure is applied to protocol buffer definitions – they are transformed into actual classes by calling the `protoc` and `ProtoGen` programs. After the build is complete, build products are copied to a common directory accessible by all projects of the solution.

Several migration paths for build events and scripts were presented in section "Compiling Files and Building Projects" of Chapter 4. For the first prototype of HeuristicLab for Mono, it was decided to write shell scripts which mimic the behavior of the original batch files.

Platform Invoke

P/Invoke is the CLI's mechanism of invoking native code from machine dependent libraries. Its problems and possible solutions have been presented in Chapter 4. This section shows which components of HeuristicLab rely on P/Invoke to call functions from native libraries.

Native platform invocations can be found by disassembling HeuristicLab's assemblies with `monodis` and using the `grep` command line utility to search the output for `pinvokeimpl`. Alternatively, the `NativeProbe` tool[15] could have been used [10]. As a result, several assemblies with P/Invoke calls were identified:

- **HeuristicLab.MainForm.WindowsForms** This project has a single reference to `SendMessage` of `user32.dll`. It is used to set the suspend and resume redrawing of the window. Removing the call had no directly visible, averse effect on HeuristicLab.
- **WeifenLuo.WinFormsUI.Docking** The external DockPanel Suite library uses multiple P/Invoke calls to interact directly with application windows and the mouse (14 calls to `user32.dll` and one call to `kernel32.dll`). HeuristicLab provides three distinct front ends for its *Optimizer* plug-in: *Docking*, *MultipleDocumentMainForm*, and *SingleDocumentMainForm*. Users are able to configure which front end to use since SVN revision 6827 (2011-09-25). Specific code will be added to HeuristicLab to make the docking front end unavailable to users on non-Windows systems, and provide them with the *MultipleDocumentMainForm* front end as fallback instead.

[15] http://www.tebiki.co.uk/cross-platform/_downloads/NativeProbe_0.2.zip, The NativeProbe Tool

- **ICSharpCode.TextEditor** The TextEditor component of SharpDevelop has eight P/Invoke calls in its source code. A blogpost[16] dating back to 2008 counted 15 P/Invoke calls using MoMA and provides fully managed code to replace some pieces of ICSharpCode.TextEditor. The code from the blogpost relies on compiler directives, which means the same assembly cannot be shared between Mono and .NET. Most of these directives can be easily replaced with a factory pattern.
- **ICSharpCode.SharpDevelop.Dom** This library is referenced from the ICSharpCode.TextEditor assembly. It imports eight functions from `fusion.dll` and one function from `shfusion.dll`. Fortunately, these functions are neither called directly nor indirectly from the ICSharpCode.TextEditor library.
- **log4net** The log4net project supports a wide range of CLI implementations (different .NET releases, Mono). Distinct assemblies exist for each target runtime. log4net uses numerous *Appenders* to generate log output on different output channels. Not all runtimes support the same appenders; for instance, Mono lacks support of *ColoredConsoleAppender*, *NetSendAppender*, and *OutputDebugStringAppender*[17]. These appenders are not used from within HeuristicLab or its libraries and the lack thereof should not pose a problem. HeuristicLab references log4net from the external library ICSharpCode.SharpDevelop.Dom and from some unit test projects. Classes from log4net which rely on P/Invoke calls are hidden with compiler directives.
- **Netron.Diagramming.Core** This library contains two Platform Invoke calls. One is `SHGetFileInfo` in `shell32.dll`, the other is `BitBlt` in `gdi32.dll`. `BitBlt` is actually unused by the Netron code and never referenced. `SHGetFileInfo` is called in a single code path which is not required by HeuristicLab. The Netron.Diagramming library should therefore not cause problems with HeuristicLab under Mono, because only managed code is executed.

5.5.2 Missing Charting Library

HeuristicLab uses the Microsoft Chart Controls to draw interactive plots and graphs. These controls are not available under Mono. One alternative is the *ZedGraph* project mentioned in Chapter 4. For this case-study a workaround was needed, which did not require re-writing all code parts, that were responsible for generating charts, to use ZedGraph instead.

The workaround for this problem was inspired by the previous section. If the Microsoft Chart Controls assembly did not contain P/Invoke calls, then

[16] http://alisdairlittle.blogspot.com/2008/11/mono-version-of-icsharptexteditor.html, Mono Version of ICSharp.TextEditor

[17] http://logging.apache.org/log4net/release/framework-support.html, Apache log4net: Supported Frameworks

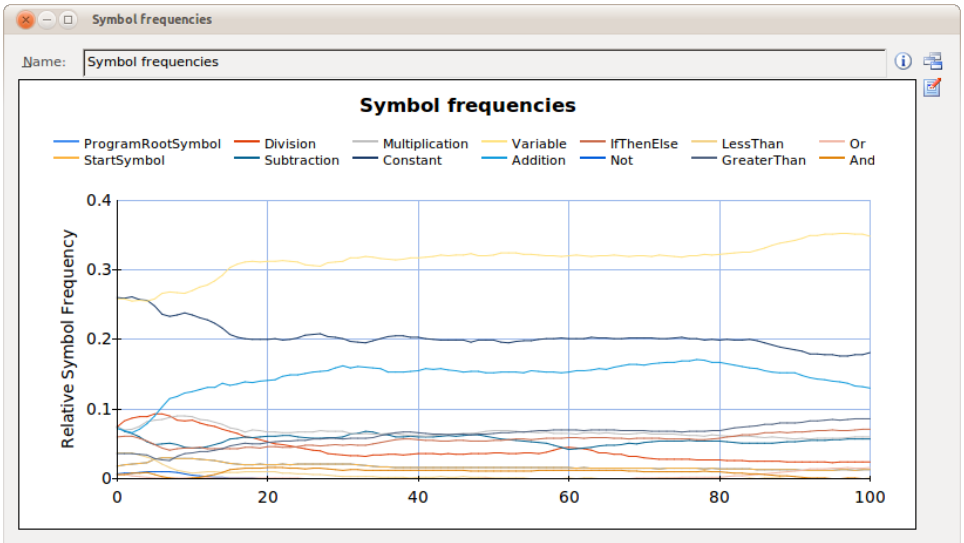

Figure 5.3: A line graph drawn with the assembly `Windows.Forms.DataVisualization` under Mono

it should be possible to use the same assembly under any CLI compliant implementation. Again, the assembly file was passed to `monodis` and its output checked for the occurrence of `pinvokeimpl`. Zero matches indicated an assembly with fully managed code. Mono should be able to consume such an assembly.

After copying the assembly and referencing it from the proper project files, the solution compiled without errors. Executing the application properly displays the charts in HeuristicLab executed under Mono and allows users to interact with them just like under .NET. Figure 5.3 shows a chart in Mono with a plot of symbol frequencies of a Genetic Programing algorithm.

However, this assembly cannot be distributed with HeuristicLab, because its license is incompatible with the GPL. Microsoft forbids the usage of its .NET DLLs on platforms other than Windows too. As a consequence, this is only a temporary workaround until a better solution is found or all charts are re-implemented with ZedGraph.

5.5.3 Unit Tests

The HeuristicLab optimization framework source code comes with a suite of unit tests for its core and its important plug-ins. These unit tests are currently written to be run with the MSTest framework. Relying on Microsoft's unit testing framework is not an option, if HeuristicLab should run and compile under different CLI implementations. Some alternatives to MSTest were

listed in Section 4.11, in particular NUnit. It also showed the first steps of converting existing, simple tests to NUnit tests.

HeuristicLab's original unit tests use standard features of the testing framework, and are written using `Assert.IsTrue` and `Assert.AreEqual` statements mostly. These simple statements can be adopted wholesale for use with the NUnit framework.

Most test attributes could be replaced by employing text substitution, as presented in Section 4.11. Other attributes, such as `DeploymentItem` and `TestContext` are MSTest specific and do not exist in NUnit. The route that was chosen for HeuristicLab is to alias `TestContext` to `System.IO.TextWriter` to have all output redirected to the console. `DeploymentItem` copies files, e.g. compiled assemblies, to the test directory. A naïve implementation was added to HeuristicLab to be used with NUnit.

MSTest makes use of accessor objects which are automatically created by Visual Studio for each class that is tested. They allow to access of private methods from unit tests. These accessor objects are missing when running the tests with NUnit. For HeuristicLab a simple class `Accessor` was written, which acts as proxy and calls the original methods with the reflection mechanism of the CLI. Another possibility is the dynamic creation of distinct accessor classes for each type through code generation mechanisms, such as `Reflection.Emit` or `System.CodeDom`. This option was not researched over the course of this thesis, because only a minimum number of accessor classes were required.

5.5.4 Persistence Layer

HeuristicLab allows users to pause algorithms, store their current state to disk, and at a later date restore them and continue the algorithm execution. This is provided through the *Persistence* plug-in, which was developed for HeuristicLab 3.x. It is able to serialize complex, arbitrary object graphs used by HeuristicLab and deserialize them identically.

Of course, it should be possible to persist algorithms and problems with one implementation of the CLI and to restore them with a different implementation. Great care was already taken to guarantee proper deserialization when a plug-in versions change.

The persistence layer fully serializes the complete object graph and type information. This is a problem, because it includes the non-publicly visible internal runtime type. The .NET framework represents types internally as `System.RuntimeType`, whereas Mono uses `System.MonoType`. Confer Chapter 4 for a general explanation of the subject matter.

`ObjectEqualityComparer`, the default comparer of the generic `Dictionary` class, is not implemented either, because it is an implementation detail of the .NET Framework (see section "Internals" of Chapter 4). There was at

least one open bug report[18] about this issue at the time this thesis was written. Using an adapted version of the workaround mentioned in the bug report it is possible to mitigate this problem within HeuristicLab. The code can be seen in Listing 5.1.

Listing 5.1: ObjectEqualityComparer implementation

```
1 [Serializable]
2 public class ObjectEqualityComparer<T> : EqualityComparer<T> {
3   public override int GetHashCode(T obj) {
4     return EqualityComparer<T>.Default.GetHashCode(obj);
5   }
6   public override bool Equals(T a, T b) {
7     return EqualityComparer<T>.Default.Equals(a, b);
8   }
9 }
```

While working on the persistence code, another bug was discovered, which occurred when adding objects of certain types in particular order to an object of type `HashSet<object>`[19]. Mono would throw an `ArgumentException` when comparing objects that implement `IStructuralEquatabla` with instances of `Tuple<,>`. This bug was fixed shortly after by the Mono development team, and the comparison now correctly returns `false` instead of throwing a runtime exception.

To be able to load `.hl` files saved from the .NET runtime the persistence layer has to convert missing types to their Mono counterparts on the fly. The same problem applies vice-versa when loading files in .NET saved from Mono. There are two options: Always store types as Microsoft .NET's types or store everything as is and perform conversion in both runtimes during deserialization.

While the first option would allow existing, unmodified versions of HeuristicLab to open files stored with Mono, it was decided to go with the second option after discussing the issue with members of the HeuristicLab core developer team. Storing types which do not exist in one runtime may lead to hard to track down errors. Using the native representation of the current runtime is more error resistant, albeit not backwards compatible with older versions of HeuristicLab.

The method responsible for loading and returning the correct type was patched to properly handle missing types that are known to have a matching type. If loading a type initially fails, the exception is caught and the type is converted to the corresponding type of the runtime currently executing. HeuristicLab then tries to load the new, converted type. If loading has failed twice, the error is propagated and displayed to the user. Of course, this only

[18] https://bugzilla.novell.com/show_bug.cgi?id=699747, Binary Serialization De-Serialization of Dictionary

[19] https://bugzilla.xamarin.com/show_bug.cgi?id=2455, `HashSet<object>` throws ArgumentException when inserting `BindingFlag` and `Tuple<,>`

works for types that have a defined mapping. At the moment, mappings are implemented for the types `System.RuntimeType` and `System.MonoType`, and `ObjectEqualityComparer`. Listing 5.2 shows a simplified version of this logic.

Listing 5.2: Type loading during deserialization

```
1 try {
2   return LoadType(typename);
3 } catch(PersistenceException) {
4   typename = GetMatchingType(typename);
5   return LoadType(typename);
6 }
```

For the prototype one-directional mapping was implemented, but bi-directional mappings can be added easily. Future enhancements might include dynamic mapping based on rules defined in an external document which, for instance, is stored in popular XML format. This would also enable users and developers to extend or change mappings without recompiling the HeuristicLab framework.

5.6 Remaining Issues and Future Tasks

Not all issues could be addressed while working on the initial port of HeuristicLab, which should compile and execute under Mono. It is likely that many issues go unnoticed for a while and will only be uncovered after extensive everyday use of HeuristicLab in the Mono environment. This section should give an overview over issues that have been identified over the course of this thesis, but take a longer time to get fixed properly. Several fixes are also required directly in the source code of Mono's class libraries.

5.6.1 ZedGraph Charting Library

Currently, the HeuristicLab framework requires the file `System.Windows.Forms.DataVisualization.dll` of the .NET Framework to be present. This file is not distributed with Mono and only exists as part .NET, however, it can be used by Mono after copying it manually. The problem is, that Microsoft does not permit customers to use .NET assemblies outside of .NET.

ZedGraph is an open-source library that provides charting support for the .NET Framework, as well as for Mono. All references to types from the `System.Windows.Forms.DataVisualization` namespace have to be replaced in HeuristicLab's code base by their corresponding ZedGraph counterparts. Alternatively, an additional abstraction layer could be introduced, which does not depend on the underlying library and would allow simple selection of the proper charting library.

5.6.2 HeuristicLab Hive

For the first proof-of-concept prototype of HeuristicLab with Mono, a port of the HeuristicLab Hive infrastructure was not considered, since the main focus was on getting the core infrastructure running. HeuristicLab Hive is an elastic and scalable infrastructure for executing any type of application distributed across a multitude of systems. The HeuristicLab Hive infrastructure is presented in a paper by several members of HEAL [29].

HeuristicLab Hive is implemented using *WCF*. Hive clients and servers communicate over the *wsHttpBinding* and *net.pipe* bindings. To allow the integration of Hive slaves executed by Mono, wsHttpBinding must be changed to a binding that is supported by Mono's implementation of the WCF stack. ServiceStack promises to be a good replacement for WCF, but first it has to be evaluated, whether a switch to ServiceStack makes sense or not.

5.6.3 Windows Forms

Windows Forms (`System.Windows.Forms`) is implemented within Mono, and provides a good basis most of the time. However, Mono's implementation is not 100 % compatible with the .NET implementation by Microsoft. Some methods are only implemented as stubs that return dummy values or are not implemented at all (`DataGridViewColumn.GetPreferredWidth` is one such example, and always returns 0). Fixes must be made directly in the source of these libraries (`mcs/class/Managed.Windows.Forms/`), and some changes already made their way into the official Mono repository.

A number of `Dispose()` methods were patched in classes related to Windows Forms Toolstrips (`ToolStrip` and `ToolStripDropDownItem`). Previously, they were disposing child components multiple times, which would result in runtime exceptions being thrown by the garbage collector. Interestingly enough, no exceptions were thrown when using the newer SGen garbage collector of Mono. Also, `ToolStripItem` was adapted to not re-calculate its size if the owning control is unset (this would throw an exception as well).

HeuristicLab offers drag and drop functionality for most of its views and components. Users can, for example, move a problem instance on another algorithm view to quickly try different algorithms or parametrization options. Operator graphs are constructed by dragging operators from the operator view to the operator graph window.

Over the course of this thesis, drag and drop functionality sometimes ceased working in an unreproducible manner. The cause for this is still unknown and yet to be found. For the operator graph window this problem can be easily circumvented by adding a new button to select operators from a list and add them to the graph directly without detouring via drag and drop. However, this is not trivially possible for other view types. One approach

would be to offer users a tree view of all open windows and views. Users would then be able to add views by selecting them from this tree.

5.7 Performance Analysis

This section compares the performance of HeuristicLab when executed under different implementations of the CLI. It should help to detect potential bottlenecks in different implementations or configurations. There are three major configurations of environments in which HeuristicLab will be used:

- **.NET (on Windows)** This is the default configuration and the basis of comparison. In the past, HeuristicLab was exclusively executed in this configuration. For running the benchmarks of this thesis, .NET 4.0 was run on Windows 7.
- **Mono on Linux** This will be the main configuration when HeuristicLab is run on other operating systems besides Microsoft Windows. For the benchmarks of this thesis the Ubuntu operating system in version 12.04 ("Precise Pangolin") was chosen. The Mono runtime and libraries were built from the Mono Git repository.
- **Mono on Windows** This configuration is likely to be less important, because most users use the Microsoft .NET Framework when running HeuristicLab on a Windows system. The alpha version Mono 2.11.2 was used on Windows 7 to get results for this configuration.

No benchmarks on other operating systems were performed.

Algorithm execution under Mono is tested in two configurations: on the one hand with the default garbage collector (Boehm) and on the other hand with the newer SGen garbage collector.

HeuristicLab comes with a number of common benchmark problems (TSPLIB, QAPLIB). The default installation of HeuristicLab also comes with a number of sample problems and algorithms. Several samples were selected and used in the benchmarking process, as listed below:

- A *Particle Swarm Optimization* algorithm, which tackles the 2-dimensional Schwefel test function.
- A *Simulated Annealing* algorithm which tackles the 2-dimensional Rastrigin test function.
- A *Genetic Programming* algorithm which tackles a symbolic classification problem.
- A *Genetic Algorithm* which tackles the "ch130" problem instance, imported from TSPLIB.
- A *Tabu Search* algorithm which tackles the same "ch130" problem instance.
- A *Genetic Algorithm* which tackles the "C101" Vehicle Routing problem from Solomon.

One algorithm and problem combination that was benchmarked, which is not available as a sample in HeuristicLab, is a *Evolution Strategy* algorithm which tackles a Quadratic Assignment Problem. Additionally, the HeuristicLab benchmark plug-in allows to run several performance benchmarks (Dhrystone [41], Whetstone [6], Linpack [8]).

For each of the problems, a HeuristicLab batch run experiment of the given instance was created with 50 repetitions. Execution times are recorded directly within the experiment file. Each algorithm was executed with the parallel engine as well as with the sequential engine, offered by HeuristicLab. The remaining parameters of the algorithms were left unmodified at their default values. The actual parameter values are listed in the respective subsection of each algorithm.

Only algorithm execution time is measured in the benchmarks. Other aspects, such as startup time, time to load and save files, etc. were left aside, since the major portion of time the HeuristicLab framework is used to tackle optimization problems. Achieving high solution quality is not a goal of this thesis; cf. other papers specifically dealing with modeling and configuration of meta-heuristic optimization algorithms with the HeuristicLab framework [40, 3, 1].

The machine that was used for the performance analysis tests was a state-of-the art desktop PC with an Intel Core2Duo E8400 CPU clocked at 3.0 GHz and 4 GB of RAM (other hardware factors are negligible for testing HeuristicLab's algorithm execution performance). The different operating systems were installed on separate partitions, and all operating systems were installed as 64 bit versions. Windows 7 was installed on two partitions, one time in combination with .NET Framework 4.0, and one time with a current build of Mono. The third partition contained Ubuntu Linux 12.04 ("Precise Pangolin"), with a custom version of Mono built from source. To run the performance tests, HeuristicLab was compiled without debug information (*Release* configuration).

Following is a presentation of the exact configuration of each algorithm and problem, as well as a detailed discussion of the results obtained from the benchmarks.

5.7.1 Results

This section presents the execution time results obtained by applying meta-heuristic algorithms to known optimization problems. First, the results of executing the three benchmarks Whetstone, Dhrystone, and Linpack are presented. A few selected algorithms and problems are then presented and compared. In particular, two popular optimization problems, *Travelling Salesman* [13] and *Quadratic Assignment* [22], were employed to obtain measurements of execution time, used to compare performance of HeuristicLab in different environments.

Parameter settings for algorithms and problems are displayed in tabular form. The top section contains the problem parameters, and the bottom section of the table contains the settings for the algorithm. Settings marked with an em-dash ("—") denote an empty setting (`null` in HeuristicLab). Most of the tested algorithm-problem combinations are shipped with HeuristicLab as sample files, but their parameter settings are mentioned here for reference and repeatability.

The results are visualized in the form of box plots. The box plots show a number of representative values of the execution time data, which was obtained from executing algorithms in different environments. Lower and upper quartiles are represented by the box; the median is plotted as solid line inside the box. Top and bottom whiskers are placed at the minimum and maximum values of the data samples, and a dashed line is placed at the arithmetic mean value. Median values are additionally printed left of each box numerically.

Each box plot shows eight configurations at most: parallel execution with .NET, Mono, Mono on Windows, and Mono with the SGen garbage collector (labeled as "SGen") – benchmark runs with the sequential engine are marked by appending "Seq." to their respective label.

It is worth mentioning that executing algorithms with the parallel engine under Mono did not result in the expected performance improvement. Actually, the opposite was the case: Performance was often worse compared to execution with the sequential engine. This is most likely due to a different scheduling strategy under Mono or might be caused by overhead incurred during task switches.

Benchmarks: Whetstone, Dhrystone, and Linpack

Only two parameters are available for each benchmark: *ChunkSize* and *TimeLimit*, both of which were left at their default value of 0.

The results of each benchmark algorithm is shown in Table 5.1. The table only lists the arithmetic mean value for each algorithm and configuration; no error ranges are displayed, since the result data showed very little variance. Higher values relate to better performance. Values in parentheses are relative to the .NET results.

Mono performs slower in the Whetstone and Linpack benchmark by 26 % and 16 %, respectively. Striking is the huge performance difference when executing the Dhrystone benchmark – .NET can execute 6 times more operations per second. Utilizing the SGen garbage collector improved performance under Mono by a small margin for all three benchmarks, the biggest improvement being with the Dhrystone benchmark (7.78 % more DIPS).

Running HeuristicLab under Mono on the Windows operating system generally resulted in worse performance, with the exception of the Linpack

Table 5.1: Benchmark results

	Dhrystone [DIPS]	Whetstone [MWIPS]	Linpack [Mflops]
.NET	4321189.76	302.76	600.66
Mono	718068.94 (16.62 %)	223.52 (73.83 %)	504.27 (83.95 %)
Mono SGen	773927.90 (17.91 %)	224.00 (73.99 %)	530.91 (88.39 %)
Mono (Windows)	581542.64 (13.46 %)	146.00 (48.22 %)	514.00 (85.57 %)

benchmark, where a small increase of 1.62 percentage points can be measured, compared to Mono with the Boehm GC.

Genetic Algorithm – Travelling Salesman Problem

The HeuristicLab framework comes with the TSPLIB of the Ruprecht-Karls-Universität Heidelberg[20]. TSPLIB provides a number of sample instances for the TSP and related problems [32]. The Travelling Salesman Problem is defined as follows:

> "Given a finite number of cities and given the travel distance between each pair of them, find the shortest tour visiting all cities and returning to the starting point." [23]

The problem instance for this benchmark is "ch130" from TSPLIB and was tackled by a Genetic Algorithm. The problem and algorithm parameters are listed in Table 5.2.

Figure 5.4 shows box plots of the execution times for the Mono and .NET runtime environment.

.NET clearly executes the algorithm faster. Compared to Mono with default settings it finishes more than twice as fast by 7.03 s (56.74 %). There is little difference between running the algorithm with the parallel or with the sequential engine under .NET. Execution times only differ by 0.3 s (5.6 % slower).

Swapping the garbage collector from Boehm to SGen reduces execution times under Mono by 0.85 s (6.86 % faster). Better execution times can be achieved with the sequential engine of HeuristicLab: execution times decrease by 2.93 s (23.65 % faster). Combining the sequential engine with the SGen garbage collector results in another 0.44 s lower execution time. This is still appreciably slower than execution with .NET, even now Mono is slower by 3.66 seconds (68.28 %).

Execution under Mono on Windows is barely faster with the parallel engine (0.05 s, 0.4 % faster), and considerably slower with the sequential engine (1.8 s, 19.03 % slower). Execution times on Windows decrease by 1.08 s

[20] http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/

Table 5.2: Parameter settings for a Genetic Algorithm and Traveling Salesman Problem

Parameter	Value
Coordinates	cf. ch130 TSP [32]
DistanceMatrix	—
Evaluator	TSPRoundedEuclideanPathEvaluator
SolutionCreator	RandomPermutationCreator
UseDistanceMatrix	True
SetSeedRandomly	True
PopulationSize	100
Selector	ProportionalSelector
CrossOver	OrderCrossover2
MutationProbability	5 %
Mutator	InversionManipulator
Elites	1
Analyzer	MultiAnalyzer
MaximumGenerations	1000

(8.75 % faster), when comparing parallel and sequential engine. A pattern that continues to show throughout all benchmarks is the high dispersion of execution time data points on Windows. This is instantly visible by looking at the box plots.

Tabu Search – Traveling Salesman Problem

This benchmark applies a tabu search algorithm to tackle the TSP "ch130" instance imported from TSPLIB. Settings of the algorithm and the problem are listed in Table 5.3.

Execution time results of solving the TSP instance with a Tabu Search algorithm are shown in Figure 5.5.

.NET with the parallel engine is more than twice as fast as Mono (59.42 % faster). Selecting the sequential engine for algorithm execution slows down performance significantly under .NET by 10.07 seconds (55.09 % longer execution times). In contrast to the above, execution times under Mono decrease by about 5 seconds (10.88 % faster) when the algorithm is executed sequentially.

Using the SGen garbage collector instead of Boehm accounts for 5.91 seconds (13.12 % slower) of longer execution time and for 4.2 s (10.46 %) with the sequential engine. This is different from the other algorithms, where SGen generally improved performance.

Executing the algorithm under Mono on Windows has the worst performance. Compared to Mono on Linux, performance is worse by 8.01 s (17.78 %

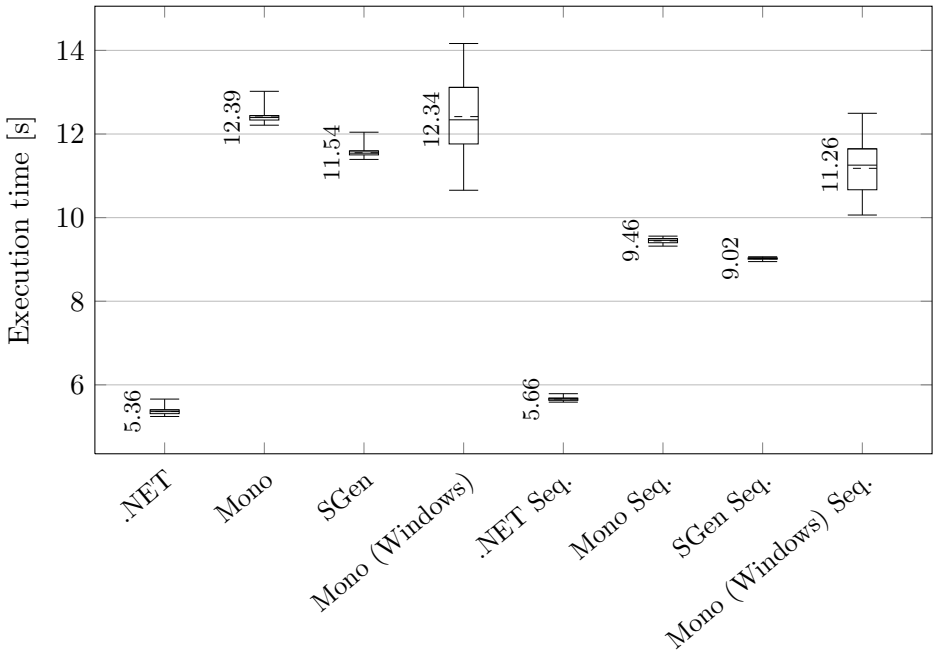

Figure 5.4: Execution times of Genetic Algorithm and the "ch130" Traveling Salesman Problem

Table 5.3: Parameter settings for Tabu Search and TSP

Parameter	Value
Coordinates	cf. ch130 TSP [32]
DistanceMatrix	—
Evaluator	TSPRoundedEuclideanPathEvaluator
SolutionCreator	RandomPermutationCreator
UseDistanceMatrix	True
SetSeedRandomly	True
MoveGenerator	StochasticInversionMultiMoveGenerator
MoveMaker	InversionMoveMaker
MoveEvaluator	TSPInversionMoveRoundedEuclideanPathEvaluator
TabuChecker	InversionMoveSoftTabuCriterion
TabuMaker	InversionMoveTabuMaker
TabuTenure	60
MaximumIterations	1000
SampleSize	750
Analyzer	MultiAnalyzer

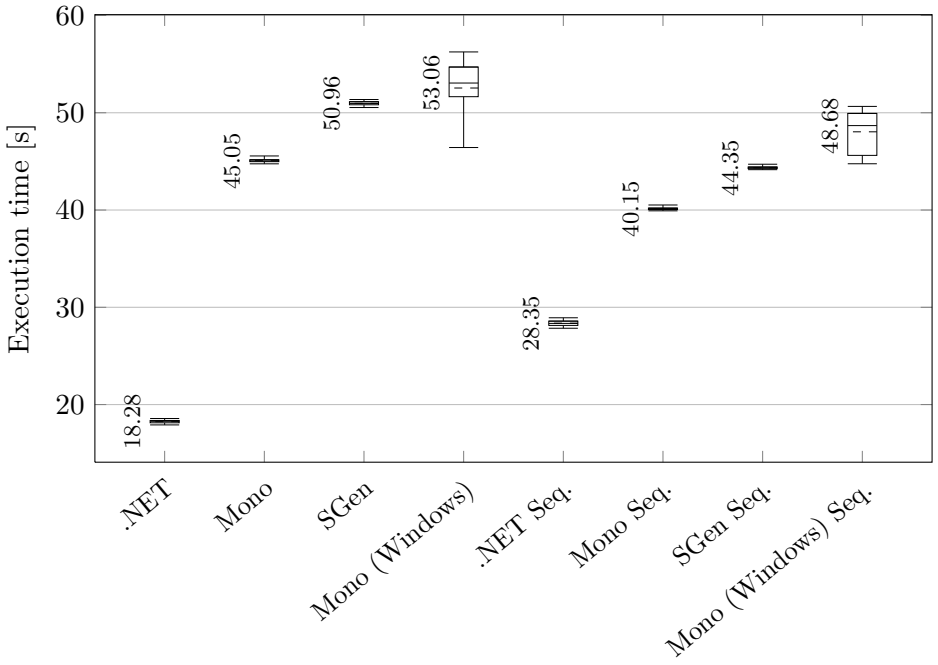

Figure 5.5: Execution times of Tabu Search and the "ch130" Traveling Salesman Problem

slower) when executed with the parallel engine, and worse by 8.53 s (21.24 % slower) when executed sequentially.

Evolution Strategy – Quadratic Assignment Problem

Similar to TSPLIB, HeuristicLab also contains QAPLIB[21], first published in 1991 at the Graz University of Technology, and now maintained at the University of Pennsylvania[22]. QAPLIB provides a unified testbed for Quadratic Assignment Problems [5].

The Quadratic Assignment Problem can be described as assigning a number of facilities to locations so that the sum of transportation costs between facilities is minimal. It can be seen as a generalized form of the Traveling Salesman Problem [22].

The problem instance for this QAP was the "chr12a" instance from QAPLIB and was tackled by an Evolution Strategy algorithm. Parameters for the algorithm (bottom) and the problem (top) are listed in Table 5.4.

[21] http://www.opt.math.tu-graz.ac.at/qaplib/
[22] http://www.seas.upenn.edu/qaplib/

Table 5.4: Parameter settings for Evolution Strategy and QAP

Parameter	Value
Distances	cf. chr12a QAPLIB [5]
Evaluator	QAPEvaluator
SolutionCreator	RandomPermutationCreator
Weights	(chr12a)
SetSeedRandomly	True
PopulationSize	20
ParentsPerChild	1
Children	100
MaximumGenerations	1000
PlusSelection	True
Recombinator	—
Mutator	InsertionManipulator
StrategyParameterCreator	—
StrategyParameterCrossover	—
StrategyParameterManipulator	—
Analyzer	MultiAnalyzer

Figure 5.6 shows the values for execution time. Mono performs very bad with this algorithm compared to .NET. With median values differences of 5.2 seconds, execution under Mono is slower by a factor of 3 (299.24 %).

A pattern similar to that of the execution times of the Genetic Algorithm and "ch130" TSP emerges. Execution times can be reduced by changing the garbage collector or by running the algorithm with the sequential engine of HeuristicLab.

Using the SGen garbage collector in combination with sequential execution improves performance significantly (166.53 %), but it still lags behind .NET by 2.08 seconds (79.7 % higher execution times). Using the SGen garbage collector alone improves performance by 0.49 s, using the sequential engine with the default garbage collector improves performance by 2.29 s (29.32 % faster).

Executing the algorithm sequentially slows it down when executed under .NET and execution times increase by 0.43 seconds (16.48 %). Parallel execution under Mono on Windows is faster by 1.21 s (15.49 % faster) compared to Mono on Linux. The opposite is the case with the sequential engine: Mono on Linux is faster by 1.12 s (16.87 % lower execution times)

Particle Swarm Optimization – Schwefel

The default values of the parameters of the Particle Swarm Optimization sample are shown in Table 5.5. The top section contains problem parame-

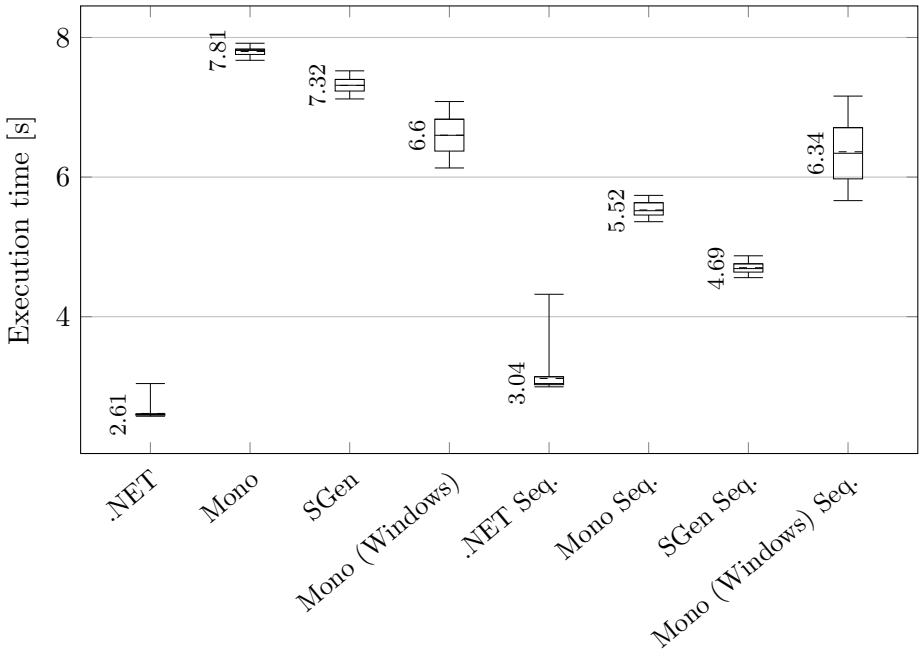

Figure 5.6: Execution times of Evolution Strategy and the "chr12a" QAP

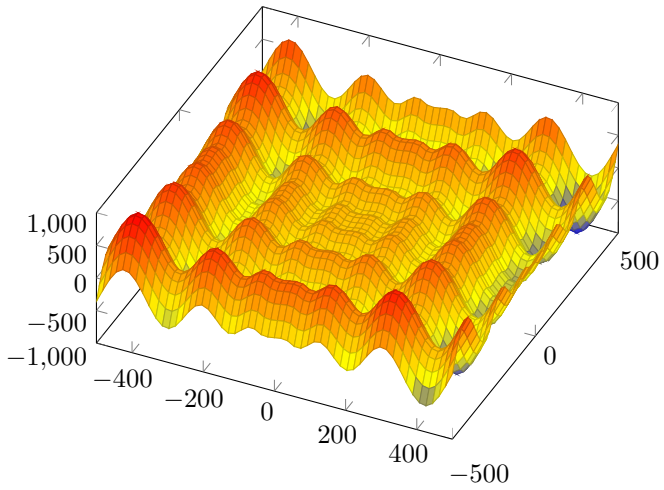

Figure 5.7: The 2-dimensional Schwefel test function, with its global minimum at $[-420.9687; -420.9687]$

ters, while the bottom section displays algorithm specific parameters. The PSO algorithm optimized the Schwefel test function (Equation 5.1) in two dimensions, depicted in Figure 5.7 [27].

Table 5.5: Parameter settings for Particle Swarm Optimization and Schwefel test function

Parameter	Value
Bounds	-500 – 500
Evaluator	SchwefelEvaluator
Maximization	False
ProblemSize	2
SolutionCreator	UniformRandomRealVectorCreator
SetSeedRandomly	True
SwarmSize	50
MaxIterations	1000
Inertia	10
PersonalBestAttraction	-0.01
NeighborBestAttraction	0.5
ParticleCreator	RealVectorParticleCreator
Analyzer	MultiAnalyzer
TopologyInitializer	—
TopologyUpdate	—
InertiaUpdate	ExponentialDiscreteDoubleValueModifier
SwarmUpdater	RealVectorSwarmUpdater

$$f(x) = \sum_{i=1}^{n} \left[-x_i \sin\left(\sqrt{|x_i|}\right) \right] \tag{5.1}$$

The results of measuring execution times of the Particle Swarm Optimization algorithm in different environments is shown in Figure 5.8. The data points are very close and show little variance.

When executed with the Mono runtime, the algorithm needs twice as long (206.07 %) to complete, compared to .NET.

Switching the engine from parallel to sequential had no significant effect on .NET (0.07 s, 3.86 % slower), but improved Mono's performance by 1.27 s (34.04 % faster). Similar to the Evolution Strategy from the previous section, execution under Mono on Windows was again faster with the parallel engine (0.44 s, 11.79 %) and slower with the sequential engine (0.47, 19.11 %).

Execution times with the SGen garbage collector are missing, because it made the Mono runtime crash. The Mono developers have been informed about this issue.

Simulated Annealing – Rastrigin

Table 5.6 shows the parameters used for the Simulated Annealing algorithm and the Rastrigin test function. Problem specific parameters are displayed

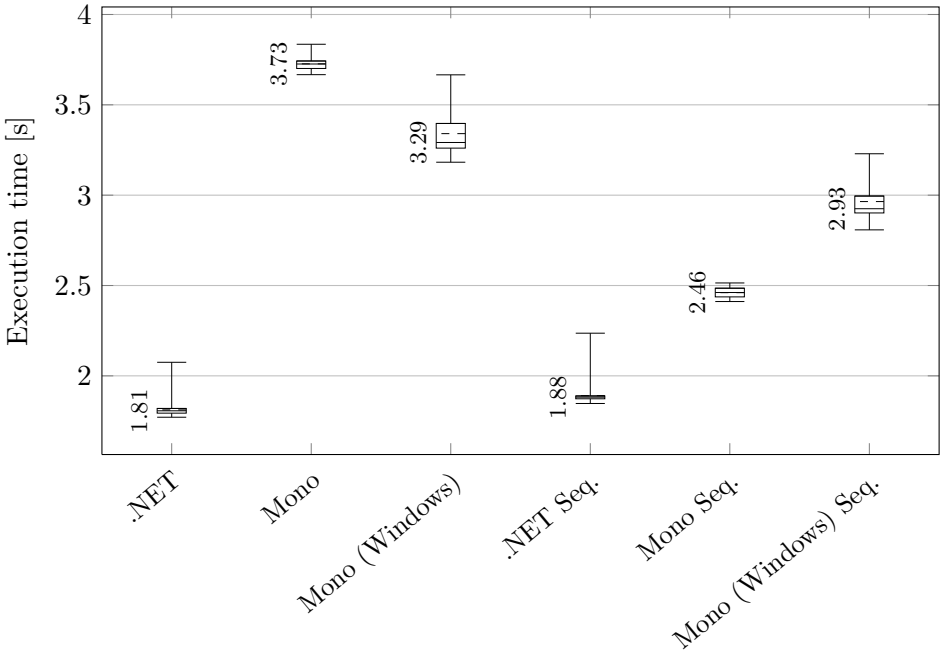

Figure 5.8: Execution times of Particle Swarm Optimization and Schwefel test function

in the top section and the bottom section shows the parameters of the algorithm. The *MaximumIterations* parameter was increased from the default

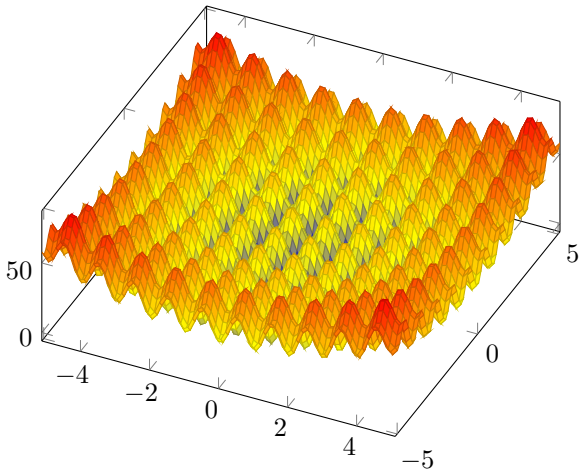

Figure 5.9: The 2-dimensional Rastrigin test function, with its global minimum at $[0; 0]$

Table 5.6: Parameter settings for Simulated Annealing and Rastrigin test function

Parameter	Value
Bounds	-5.12 – 5.12
Evaluator	RastriginEvaluator
Maximization	False
ProblemSize	2
SolutionCreator	UniformRandomRealVectorCreator
SetSeedRandomly	True
MaximumIterations	1000
InnerIterations	50
StartTemperature	1
EndTemperature	0.000001
Analyzer	MultiAnalyzer
MoveGenerator	StochasticNormalMultiMoveGenerator
MoveEvaluator	RastriginAdditiveMoveEvaluator
MoveMaker	AdditiveMoveMaker
AnnealingOperator	ExponentialDiscreteDoubleValueModifier

value of 100 to 1000 in order obtain more representative values for this benchmark. The Simulated Annealing algorithm optimized the Rastrigin test function (Equation 5.2) in two dimensions [27]. Figure 5.9 shows a plot of the Rastrigin function.

$$f(x) = 10n + \sum_{i=1}^{n} \left[x_i^2 - 10\cos(2\pi x_i)\right] \tag{5.2}$$

The results obtained from running the Simulated Annealing algorithm on different systems can be found in Figure 5.10.

Once again, performance of Mono can be improved by choosing the SGen garbage collector. It resulted in a decrease of execution time of 0.26 s (11.76 % faster) with the parallel engine and 0.3 s (13.27 % faster) with the sequential engine. The fastest configuration of Mono (parallel execution, SGen) is 0.26 s (15.38 %) slower than .NET.

Executing the algorithm sequentially lead to slightly worse performance with Mono (2.26 % slower). The difference under .NET was barely measurable: sequential execution resulted in 0.01 s longer execution time (0.59 % slower). Mono on Windows was slower for both engines by a large margin. Compared to execution times under Mono with normal settings, execution times were higher by 0.71 (32.12 % slower) with the parallel engine and by 0.69 s (30.53 % slower) with the sequential engine.

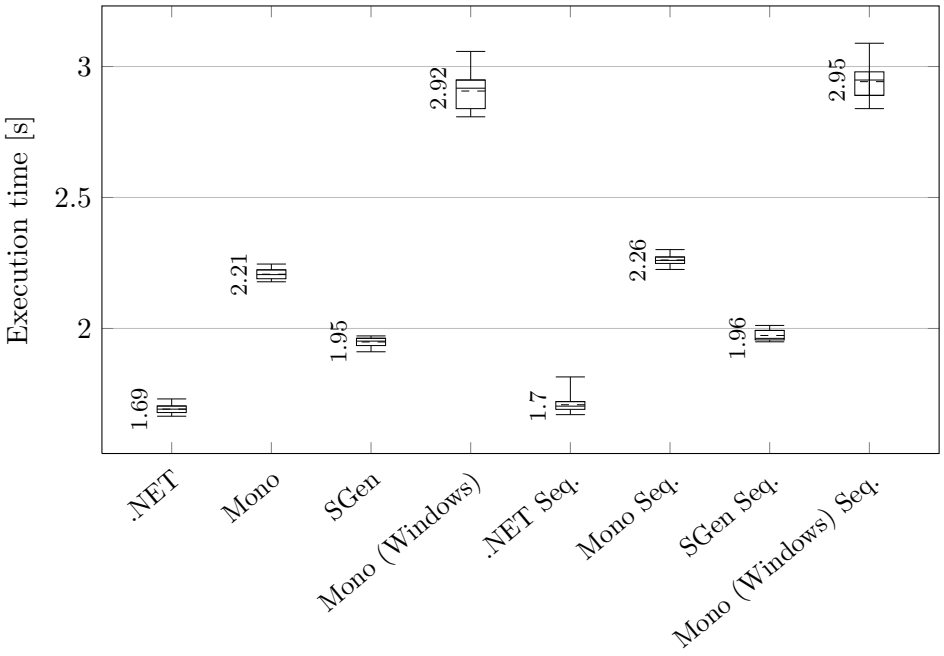

Figure 5.10: Execution times of Simulated Annealing and Rastrigin function

This algorithm was the only one besides Genetic Programming for which the parallel engine performed better for all configurations.

Genetic Algorithm – Vehicle Routing Problem

This benchmark test tackles the C101 Vehicle Routing Problem instance from Solomon [37] with a genetic algorithm. The parameter settings of algorithm and problem are listed in Table 5.7. The repetitions parameter was set to 20 for this benchmark.

Figure 5.11 shows the results obtained from executing the Genetic Algorithm in different runtime environments. Execution times of Mono with the SGen garbage collector could not be measured, because it resulted in reproducible crashes. The Mono team has already been made aware of this issue.

Differences in execution time between Mono and .NET are very obvious in this example. .NET outperforms Mono by 54.98 s (129.06 % faster) when the algorithm is executed with the parallel engine and by 49.55 s (111.83 % faster) with the sequential engine. This means a more than twofold increase in execution time.

Execution times between parallel and sequential engine differ only minimally. Executing the algorithm with the sequential engine is 3.81 % faster

Table 5.7: Parameter settings for Genetic Algorithm and VRP

Parameter	Value
Capacity	200
Coordinates	cf. C101 VRP Solomon [37]
Demand	(C101)
DistanceMatrix	(C101)
DueTime	(C101)
EvalDistanceFactor	1
EvalFleetUsageFactor	100
EvalOverloadPenalty	100
EvalTardinessPenalty	100
EvalTimeFactor	0
Evaluator	VRPEvaluator
ReadyTime	(C101)
ServiceTime	(C101)
SolutionCreator	RandomCreator
UseDistanceMatrix	True
Vehicles	25
SetSeedRandomly	True
PopulationSize	100
Selector	TournamentSelector
Crossover	MultiVRPSolutionCrossover
MutationProbability	5 %
Mutator	MultiVRPSolutionManipulator
Elites	1
Analyzer	MultiAnalyzer
MaximumGenerations	1000

(3.72 s) under Mono, and 4.01 % slower (1.71 s) under .NET. The highest execution times were recorded for Mono on Windows: 114.73 s and 120.95 s.

Genetic Programming – Symbolic Classification

A genetic programming algorithm was applied to tackle a symbolic classification problem. The problem dataset is available in the UCI Machine Learning Repository[23] and contains 961 rows of data from breast cancer screenings, each row consisting of 6 columns (BI-Rads, Age, Shape, Margin, Density, Severity).

[23] http://archive.ics.uci.edu/ml/datasets/Mammographic+Mass, UCI Machine Learning Repository: Mammographic Mass Data Set

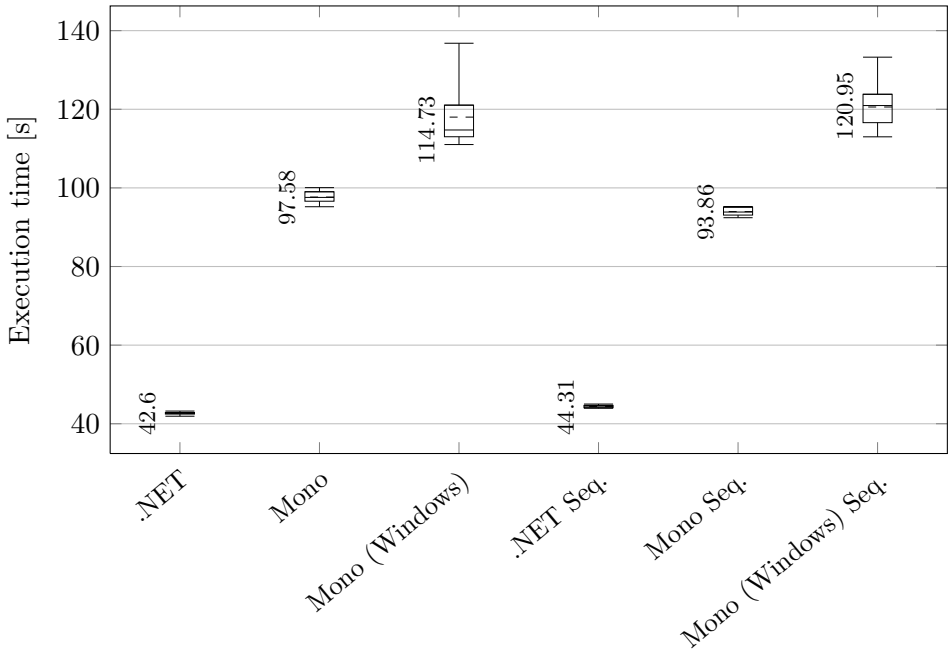

Figure 5.11: Execution times of Genetic Algorithm and "C101" Vehicle Routing Problem

The parameter settings used for the algorithm and the problem are printed in Table 5.8. The repetition parameter of the batch run was set to 20 for this benchmark.

The results of the performance measurements of the symbolic classification problem are shown in Figure 5.12. Recorded execution times of a single configuration vary significantly for this type of algorithm: Relative standard deviation ranges from 9 % to 16 %.

The box plot makes it immediately visible that it takes almost twice as long (192.82 %) to classify the dataset with Mono as compared to .NET. This is especially unfortunate, given that a single pass already takes close to 2 minutes with .NET (3.84 minutes with Mono). Execution under Mono on Windows is even worse: One run takes 4.86 minutes with the parallel engine, and 5.05 minutes with the sequential engine.

Performance under Mono can be improved slightly by selecting another garbage collector (SGen instead of the default Boehm). Executing the algorithm with the sequential engine generally slowed down execution. Lowest execution times were achieved with the parallel engine and the SGen garbage collector, with a median execution time of 220.76 seconds. Slowest was the execution with the default garbage collector and the sequential engine, with a median execution time of 251.72 seconds (excluding Mono on Windows). Us-

Table 5.8: Parameter settings for a Genetic Programming algorithm and a Symbolic Classification problem

Parameter	Value
Evaluator	Mean squared error Evaluator
FitnessCalculationPartition	Start: 0, End: 400
MaximumSymbolicExpressionTreeDepth	10
MaximumSymbolicExpressionTreeLength	100
ProblemData	mammographic_masses.csv
RelativeNumberOfEvaluatedSamples	100 %
SolutionCreator	ProbabilisticTreeCreator
SymbolicExpressionTreeGrammar	TypeCoherentExpressionGrammar
ValidationPartition	Start: 400, End: 800
SetSeedRandomly	True
PopulationSize	1000
Selector	TournamentSelector
Crossover	SubtreeCrossover
MutationProbability	15 %
Mutator	MultiSymbolicExpressionTreeManipulator
Elites	1
Analyzer	MultiAnalyzer
MaximumGenerations	100

ing the sequential engine with .NET decreased the algorithm's performance by 22.39 s (18.67 % slower). This is the only algorithm-problem combination where sequential execution was slower for all tested configurations (.NET, Mono, Mono with SGen, and Mono on Windows).

5.7.2 Bottlenecks

The samples included with HeuristicLab showed worse performance of Mono compared to .NET. To get an initial view of what code areas perform worse in Mono, Mono can be started with the `--profile=log:sample` parameter[24]. The profiler records statistical data periodically and enables developers to generate a report that contains a list of methods sorted by their call frequency. This gives a good overview of the amount of time an application spends in each method during execution. Statistical profiling data was only recorded under Mono on Linux.

The basis for profiling was a Genetic Algorithm solving the "ch130" Travelling Salesman Problem from the previous section. The algorithm was converted to a batch run with 10 repetitions. Execution times were recorded for both garbage collectors (Boehm and SGen) and the parallel and sequential engine of HeuristicLab, totalling four different configurations. Each configuration was run 5 times and its sample values summed up. The recorded

[24] http://www.mono-project.com/Profiler

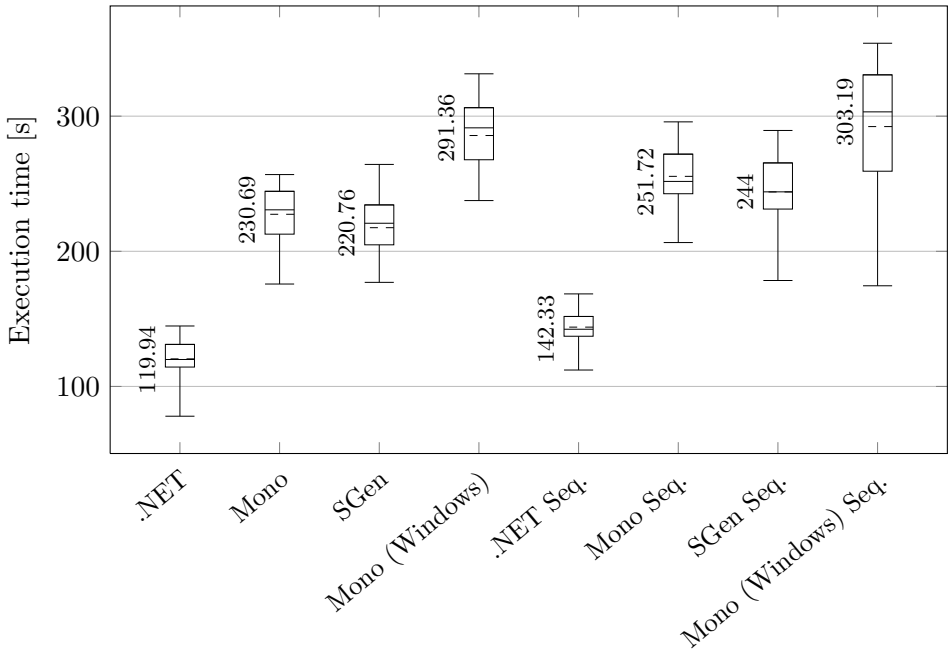

Figure 5.12: Execution times of Genetic Programming and Symbolic Classification

values were normalized afterwards by subtracting the samples recorded while loading HeuristicLab without executing the optimization algorithm.

The following observations were made (The `HeuristicLab` namespace has been shortened to `HL` to improve readability):

- **Boehm GC, Parallel Engine** A lot of time is apparently spent in methods of the garbage collector (25626 hits in `GC_mark_from`, 6071 hits in `GC_local_malloc`, 5575 hits in `GC_compare_and_exchange`, and 2990 hits in `GC_reclaim_clear`) when running HeuristicLab under Mono with the Boehm GC.

  A large number of hits can also be observed for the method `string :GetHashCode()`, ranking fourth with 3490 recorded hits. Several getters for collections follow: 2491 hits in `HL.Data.ValueTypeMatrix`, 1804 hits in `HL.Data.ValueTypeArray`, and 1745 hits in `System.Collections .Generic.Dictionary`.

  Threading related methods and properties – such as `System.Threading .ThreadLocal:Value`, `pthread_mutex_trylock`, `pthread_mutex_unlock` – were hit between 616 and 991 times.
- **Boehm GC, Sequential Engine** Executing the algorithm with the sequential engine lead to a similar result. Again, a large part of the time seems to be spent in methods of the garbage collector, albeit the num-

ber of hits was between 5 to 8 times lower (3272 hits in `GC_mark_from`, 1042 hits in `GC_local_malloc`, 802 hits in `GC_compare_and_exchange`).

`string:GetHashCode()` ranks fourth again, this time with recorded 630 hits. Collections come next as well: 526 hits in `System.Collections.Generic.Dictionary`, 406 hits in `HL.Data.ValueTypeMatrix`, and 289 hits in `HL.Data.ValueTypeArray`.

Generally, most values were lower compared to the values recorded with the parallel engine.

- **SGen GC, Parallel Engine** Using Mono's SGen garbage collector changes picture only somewhat. The largest number of hits still occurs within the garbage collector (12091 hits in `mono_g_hash_mark`, 7791 hits in `add_profile_gcvroot.isra.18`). A lot of new methods, related to the SGen garbage collector, show up under the first ten places of the list.

  `string:GetHashCode()` is still high up the list with 3491 hits. Collection methods and properties were recorded with 1289 hits in `HL.Data.ValueTypeMatrix`, 1770 hits in `HL.Data.ValueTypeArray`, 1747 hits in `System.Collections.Generic.Dictionary`.

  Methods from the `System.Threading` namespace were sampled with 1040, 1019, and 933 hits.

- **SGen GC, Sequential Engine** Changing the parallel engine to the sequential engine has similar effects with SGen as it did with Boehm. Generally, fewer samples were recorded (around 6 times fewer hits). `mono_g_hash_mark` has 2147 hits recorded, followed by 1397 hits in `add_profile_gc_root.isra.18`.

  Collection methods and `string:GetHashCode()` follow with 608, 508, 477, and 334 hits. Threading methods show up with 315, 237, 138, and 134 hits.

Table 5.9 and Table 5.10 list the top methods for the Boehm and the SGen GC, respectively. Again, `HeuristicLab` was replaced by `HL` to save horizontal space. Methods are sorted descending by number of hits recorded with the parallel engine.

5.7.3 Result Summary

This section summarizes the results from the previous sections. Table 5.11 lists all obtained results for easy comparison. The table shows the median execution times in seconds for each algorithm-problem combination and the environment it was run in. Values are rounded to two decimal places. Lower execution times correlate to better performance. If an algorithm could not be tested in the environment, an em-dash ("—") is listed instead.

The benchmarks and performance analysis results of this section clearly showed that Mono lags behind .NET regarding performance, when executing optimization algorithms of the HeuristicLab framework. Execution times

Table 5.9: Profiling data recorded under Mono with Boehm GC

Hits/engine		
Parallel	Seq.	Method
25626	3272	GC_mark_from
6071	1042	GC_local_malloc
5575	802	GC_compare_and_exchange
3490	630	string:GetHashCode()
2990	507	GC_reclaim_clear
2491	406	HL.Data.ValueTypeMatrix<double>:get_Item(int,int)
2011	358	GC_build_fl
1804	289	HL.Data.ValueTypeArray<int>:get_Item(int)
1745	526	System.Collections.Generic.Dictionary:get_Item(TKey)
1715	283	mono_object_new_specific
1337	394	System.Collections.Generic.Dictionary:ContainsKey(TKey)
1328	225	HL.Encodings.PermutationEncoding.OrderCrossover2:Apply(...)
1185	231	(wrapper) object:virt_stelemref_interface(intptr,object)
1185	235	mono_object_new_alloc_specific
991	302	System.Threading.ThreadLocal:get_Value()

Table 5.10: Profiling data recorded under Mono with SGen GC

Hits/engine		
Parallel	Seq.	Method
12091	2147	mono_g_hash_mark
7791	1397	add_profile_gc_root.isra.18
6287	1054	/lib/x86_64-linux-gnu/libc.so.6
4114	676	encode_sleb128
3491	608	string:GetHashCode()
2583	429	write
2524	358	serial_copy_object
2189	477	HL.Data.ValueTypeMatrix`1<double>:get_Item(int,int)
2163	369	encode_uleb128
2120	254	serial_scan_object
2076	393	ms_sweep
2007	323	mono_gc_memmove
1868	347	alloc_obj
1770	334	HL.Data.ValueTypeArray`1<int>:get_Item(int)
1747	508	System.Collections.Generic.Dictionary`2:get_Item(TKey)

under Mono with default settings range between 122 % and 299 % compared to execution times under .NET, for the seven benchmarks examined.

Execution times change when choosing another garbage collector under Mono (SGen instead of the default Boehm GC). The SGen garbage collector generally improved performance under Mono slightly ($\approx$ 7 %); only the Traveling Salesman Problem optimized by an Tabu Search algorithm

Table 5.11: Median Execution Times of Different Algorithm Classes and Problem Instances in Seconds

	TSP, Genetic Algorithm	TSP, Tabu Search	VRP, Genetic Algorithm	Particle Swarm Optimization	Simulated Annealing	QAP, Evolution Strategy	Genetic Programming
.NET	5.36	18.28	42.60	1.81	1.69	2.61	119.94
.NET Seq.	5.66	28.35	44.61	1.88	1.70	3.04	142.33
Mono	12.39	45.05	97.58	3.73	2.21	7.81	230.69
Mono Seq.	9.46	40.15	93.86	2.46	2.26	5.52	251.72
Mono SGen	11.54	50.96	—	—	1.95	7.32	220.76
Mono SGen Seq.	9.02	44.35	—	—	1.96	4.59	244.00
Mono (Windows)	12.34	53.06	114.73	3.29	2.92	6.60	291.36
Mono (Windows) Seq.	11.26	48.68	120.95	2.93	2.95	6.34	303.19

incurred a slight performance hit (50.96 s compared to 45.05 s, which equals an increase in execution time by 13.12 %).

Unfortunately, SGen cannot be used for all algorithms. The Genetic Algorithm solving a Vehicle Routing Problem and the Particle Swarm Optimization algorithm for the Schwefel test function are crashing when executed under Mono with SGen. This should get fixed in future versions of Mono, when SGen becomes more mature and stable.

On average, algorithms executed under Mono execute longer by 119.3 % (median: 129.1 %). When considering only the fastest configurations under Mono (GC, Parallel vs. Sequential Engine), execution times are still higher by 74.21 % on average (median: 75.9 %). Even then, a relative difference of execution times as high as 120.3 % can be observed for the Genetic Algorithm and VRP benchmark; the lowest relative difference was achieved in the Simulated Annealing benchmark (15.38 % slower). Especially for performance critical applications such as HeuristicLab this is a problem.

Mono on Windows has comparable performance metrics to Mono on Linux. It is significantly slower in the Simulated Annealing benchmark (0.71 s or 32.12 % longer execution times) and in the Genetic Programming benchmark (60.67 s or 26.3 % longer execution times). Performance gains when executing algorithms with the sequential engine of HeuristicLab on Windows were not as high, as compared to executing algorithms with the sequential engine under Mono on Linux. Here, Mono on Linux outperformed Mono

on Windows for all benchmarks. Unfortunately, it was not possible to test HeuristicLab with SGen under Mono on the Windows operating system, due to crashes during startup (Error message: "Windows systems haven't been ported to support `mono_thread_state_init_from_handle`").

When executing algorithms under Mono with the sequential engine, execution times were generally lower for all benchmarks, with the exception of Genetic Algorithm solving a Vehicle Routing Problem under Mono on Windows (5.14 % slower). Under .NET, the opposite was the case: execution times with the sequential engine were higher for all benchmarks, as expected. For the Tabu Search with TSP benchmark, the algorithm finished faster by 55.09 % with the parallel engine. Genetic Programming was always faster when executed with the parallel engine of HeuristicLab (18.7 % faster under .NET, $\approx$ 10 % faster under Mono on Linux, and 4.1 % under Mono on Windows).

On average, the biggest difference in execution times between parallel and sequential engine was achieved when using the Boehm garbage collector (20.34 % faster), closely followed by SGen (18.03 % faster). The biggest relative jump could be observed for the Evolution Strategy solving a Quadratic Assignment Problem. When using the SGen garbage collector, performance could be increased by 37.3 % with the sequential engine. On Windows, performance was better for all tested algorithms when using the parallel engine. Execution times were lower by 11.36 % on average (median: 5.3 %) compared to the execution times with the sequential engine. This leads to think that heavy usage of threads incurs quite some overhead under Mono, especially on Linux systems.

A project that aims to improve the performance of native code is the integration of LLVM[25] into Mono's JIT compiler[26]. LLVM is able perform additional optimization on generated native code, before it is executed by the CPU. These optimizations trade off longer compilation times and higher memory consumption for more optimized native code.

[25] http://llvm.org/, The LLVM Compiler Infrastructure Project
[26] http://www.mono-project.com/Mono:Runtime:Documentation:LLVM

Chapter 6

Outlook and Conclusion

This final chapter summarizes the identified problem areas of the previously analyzed components. Developers should take these into account when developing applications for the .NET Framework, if they plan to make these applications available for other platforms too.

Mono is a mature platform and a compliant implementation of the Common Language Infrastructure (ISO/IEC 23271 and ECMA-335). It provides a solid base for developing and running applications on a number of different operating systems, such as Linux, Windows, and Mac OS.

However, developers cannot expect to compile and run a project targeting .NET without modifications under Mono. While the core libraries described by the CLI standard (found in the namespace `System`) are implemented and well supported, third party libraries and vendor specific extensions make migration of applications more difficult. Especially newer libraries, such as WPF or WCF, are often missing in Mono or not implemented completely.

When starting a new project, it is important to evaluate solutions also based on their compatibility with different implementations of the CLI. Many solutions and libraries exist that work with .NET as well as with Mono. This improves cross-platform compatibility drastically from the start and lessens the effort necessary to port an application.

Another approach is to start new projects in the Mono environment. This way only classes and functionality that is present in Mono can be used. Mono can be used on several operating systems and most parts of Mono are a subset of Microsoft .NET, which means it is more likely that Mono applications run in the .NET runtime than the other way round.

One potential problem source is the choice of the GUI framework. The standard does not specify how GUI frameworks should be implemented and what functionality they must provide. Thus, several different GUI frameworks exist which are provided by third parties. Popular choices include Windows Forms, Windows Presentation Foundation, MonoMac, GTK# and

Qyoto (Qt/KDE). Developers planning to run their applications on different operating systems should either choose Windows Forms (of which Mono provides an implementation) or better, use one of the frameworks that are not restricted to a single platform (GTK#, Qyoto)[1]. Of course, it is still possible to provide different graphical front ends for different operating systems, but this increases the required coding effort and maintenance costs immensely.

The Common Language Infrastructure specifies ways to call native functions from system libraries (DLLs, shared object files, or `.dylib` files) from managed code with *P/Invoke*. Using this mechanism creates a direct dependence on the underlying operating system. Although the Mono runtime is able to choose the correct library depending on the operating system, relying on P/Invoke in a cross-platform landscape generally leads to problems or high maintenance costs sooner or later.

But even though Mono provides a multitude of class libraries and does its best to provide a CLI implementation that closely matches .NET, not every project can be ported with affordable costs and adequate effort. Existing, large scale projects might have accumulated dependencies on the underlying operating system or the concrete implementation of the CLI. GUI intensive applications with strong coupling between the business logic back end and the graphical front end are also difficult to port, if the front end libraries are not available on other operating systems. Given that it requires a lot of developer effort to properly separate and abstract the graphical front end from the back end, it is often not easily possible to exchange the GUI framework.

Mono developers implement amendments to the CLI or C# standards fairly quick. For instance, the keywords `async` and `await` of the upcoming C# version have already been added to Mono's compiler. Existing bugs are fixed on a daily basis and missing features will be added over time. Development around Mono will definitely remain exciting. Just recently, Xamarin, the company behind Mono, raised $ 12 million in venture capital to boost development. Xamarin also puts a strong emphasis on development of mobile apps, providing Mono for mobile devices (MonoTouch for iOS, Mono for Android).

All in all, Mono is and is going to be one of the best alternatives to Microsoft's .NET Framework when working in a managed environment based on the Common Language Infrastructure, which is not only intended to run on the Windows operating system, but also on Linux, Mac OS, and mobile devices.

[1] http://www.mono-project.com/Gui_Toolkits

6.1 Status and Future of HeuristicLab for Mono

This section briefly summarizes the work done to create a first prototype of HeuristicLab which can be compiled and executed with tools provided by Mono on Linux. Possible future developments of HeuristicLab to increase cross-platform compatibility are highlighted.

As the previous chapters have shown, it is possible to use Mono as replacement for .NET in the HeuristicLab project. Not all components have been ported, but the core functionality is working under Mono and Linux, apart from some minor annoyances. Also, some runtime bugs will have yet to be discovered by extensive usage and testing.

The current proof-of-concept prototype of HeuristicLab enables users to use HeuristicLab on different operating systems to design and run algorithms, and to store and retrieve algorithm and problem files. Result data can be analyzed and visualized by different means, including a table view and several interactive charts, such as bubble charts or box plots.

For developers, HeuristicLab now provides additional solution files to be used with MonoDevelop. These solution files include test projects to be used with the NUnit unit testing framework. The supplied shell scripts allow easy and automated deployment of HeuristicLab libraries.

Currently missing features and problems include some incomplete implementation details of the Windows Forms GUI framework and the missing Hive component of HeuristicLab (due to `wsHttp*`-bindings). WCF could be replaced with the ServiceStack project in future releases of HeuristicLab, if it proves to be a viable replacement.

Source code that originated as part of Chapter 5 can be found in the HeuristicLab Subversion repository. It is tracked in its own branch[2] for everyone to browse and inspect. Interested parties are welcomed and encouraged to file bug reports and send patches to further improve HeuristicLab's compatibility with the Mono project. Furthermore, creating the proof-of-concept prototype of HeuristicLab resulted in mutual benefits for both projects. Several bugs were discovered in the Mono runtime, as well as the Mono compiler. Most of them were already fixed in the Mono source code repository.

One big disadvantage of running HeuristicLab inside the Mono runtime environment is the performance impact incurred. Runtime tests performed in Section 5.7 showed higher execution times of up to 300 % for sample files that are included with HeuristicLab. Taking into account only the lowest execution times that were recorded under Mono, execution times were still higher by 73.6 % on average. Differences in execution times might diminish in the future with improvements being made to Mono's JIT compiler or by integrating LLVM.

[2] http://dev.heuristiclab.com/svn/hl/core/branches/HeuristicLab.Mono/

This thesis has laid a solid groundwork – leaving aside the worse performance – for HeuristicLab to be executed under and developed with the Mono project. The HeuristicLab team will hopefully continue this effort and make HeuristicLab more compatible with the Mono project to ensure a wider outreach across many operating systems and platforms. Considering the open-source nature of HeuristicLab, this is only the next logical step to take.

List of Figures

List of Tables

Listings

Bibliography

[1] M. Affenzeller, S. Winkler, S. Wagner, and A. Beham. *Genetic algorithms and genetic programming: modern concepts and practical applications*, volume 6. Chapman & Hall/CRC, 2009.

[2] T. Archer and A. Whitechapel. *Inside C#*. Microsoft Press, 2001.

[3] A. Beham, M. Affenzeller, S. Wagner, and G. Kronberger. Simulation optimization with HeuristicLab. *Proceedings of the 20th European Modeling and Simulation Symposium (EMSS2008)*, pages 75–80, 2008.

[4] H. J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18(9):807–820, 1988.

[5] R. E. Burkard, S. Karisch, and F. Rendl. QAPLIB – a quadratic assignment problem library. *European Journal of Operational Research*, 55(1):115–119, 1991.

[6] H. J. Curnow and B. A. Wichmann. A synthetic benchmark. *The Computer Journal*, 19(1):43–49, 1976.

[7] M. de Icaza. Miguel de Icaza's blog, http://tirania.org/blog.

[8] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart. *LINPACK: users' guide*. Society for Industrial Mathematics, 1979.

[9] E. Dumbill and N. M. Bornstein. *Mono: A Developer's Notebook*. O'Reilly Media, first edition, July 2004.

[10] M. J. Easton and J. King. *Cross-Platform .NET Development: Using Mono, Portable .NET, and Microsoft .NET*. A-Press, first edition, September 2004.

[11] Ecma International. ECMA-335: Common Language Infrastructure (CLI). *ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland*, 2002.

[12] Ecma International. Standard ECMA-334: C# language specification, 2005.

[13] M. M. Flood. The traveling-salesman problem. *Operations Research*, 4(1):61–75, 1956.

[14] E. Geschwinde. *Mono – .NET-kompatible Anwendungen mit dem Open Source-Framework*. Markt+Technik, 2003.

[15] J. Gough. *Compiling for the .NET Common Language Runtime (CLR)*. Prentice Hall, first edition, November 2001.

[16] C. Holm, B. Spuida, and M. Krüger. *Dissecting a C# Application: Inside SharpDevelop*. Wrox Press, first edition, January 2003.

[17] A. Hunt and D. Thomas. *Pragmatic Unit Testing in C# with NUnit (Pragmatic Programmers)*. The Pragmatic Programmers, first edition, May 2004.

[18] ISO. *ISO/IEC 23271:2003: Information technology — Common Language Infrastructure*. International Organization for Standardization, Geneva, Switzerland, 2003.

[19] ISO. *ISO/IEC 23270:2006: Information technology — Programming languages — C#*. Technical report. International Organization for Standardization, Geneva, Switzerland, 2006.

[20] ISO. *ISO/IEC 23271:2012: Information technology — Common Language Infrastructure*. International Organization for Standardization, Geneva, Switzerland, 2012.

[21] C. Kaan. *Mono – .NET goes LINUX*. Franzis Verlag GmbH, December 2007.

[22] T. C. Koopmans and M. Beckmann. Assignment problems and the location of economic activities. *Econometrica: Journal of the Econometric Society*, pages 53–76, 1957.

[23] E. L. Lawler, A. H. G. R. Kan, J. K. Lenstra, and D. B. Shmoys. *The Traveling Salesman Problem – A Guided Tour of Combinatorial Optimization*. Wiley-Interscience series in discrete mathematics and optimization. Wiley, 1985.

[24] J. Levinson. *Software Testing with Visual Studio 2010*. Microsoft .NET Development Series. Addison-Wesley Professional, first edition, March 2011.

[25] L. A. MacVittie. *XAML in a Nutshell*. O'Reilly Media, first edition, April 2006.

[26] M. Mamone. *Practical Mono (Expert's Voice in Open Source)*. Apress, first edition, December 2005.

[27] M. Molga and C. Smutnicki. Test functions for optimization needs. *Test functions for optimization needs*, 2005.

[28] A. Nathan. *Windows Presentation Foundation Unleashed (WPF)*. Sams, first edition, December 2006.

[29] C. Neumüller, A. Scheibenpflug, S. Wagner, A. Beham, and M. Affenzeller. Large scale parameter meta-optimization of metaheuristic optimization algorithms with HeuristicLab Hive.

[30] J. A. Parejo, A. Ruiz-Cortés, S. Lozano, and P. Fernandez. Metaheuristic optimization frameworks: a survey and benchmarking. *Soft Computing – A Fusion of Foundations, Methodologies and Applications*, pages 1–35, 2012.

[31] D. S. Platt. *Introducing Microsoft .NET*. Microsoft Press, 2001.

[32] G. Reinelt. TSPLIB 95 documentation. *University of Heidelberg*, 1995.

[33] H. J. Schönig and E. Geschwinde. *Mono kick start*. Kick Start Series. Sams, 2003.

[34] C. Sells and I. Griffiths. *Programming Windows Presentation Foundation*. O'Reilly Media, September 2005.

[35] C. Sells and M. Weinhardt. *Windows Forms 2.0 Programming*. Microsoft .NET Development Series. Addison-Wesley Professional, second edition, May 2006.

[36] J. Sharp. *Windows Communication Foundation 4 Step by Step (Step by Step (Microsoft))*. Microsoft Press, first edition, November 2010.

[37] M. M. Solomon. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations Research*, 35(2):254–265, 1987.

[38] S. Wagner. *Heuristic Optimization Software Systems – Modeling of Heuristic Optimization Algorithms in the HeuristicLab Software Environment*. PhD thesis, Institute for Formal Models and Verification, Johannes Kepler University, Linz, Austria, 2009.

[39] S. Wagner and M. Affenzeller. The HeuristicLab optimization environment. *Institute of Formal Models and Verification Johannes Kepler University Linz, Austria*, 2004.

[40] S. Wagner, G. Kronberger, A. Beham, S. Winkler, and M. Affenzeller. Modeling of heuristic optimization algorithms. In *Proceedings of the 20^{th} European Modeling and Simulation Symposium*, pages 106–111, 2008.

[41] R. P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, 1984.