Cross-Language Compiler using Roslyn and Coco/R for the Common Language Runtime

Marius C. Dinu

BACHELORARBEIT

Nr. 1310307054-B

eingereicht am
Fachhochschul-Bachelorstudiengang

Software Engineering

in Hagenberg

im Februar 2016

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, February 1, 2016

Marius C. Dinu

Contents

Abstract

Interoperability and code generation has become one of the main topics for nowadays modern frameworks, which have to provide various Application Programming Interfaces (APIs) and ways to translate, parse, bind and instantiate objects at compile time and runtime. At the center of all these frameworks mechanisms, concepts such as *Reflection*, lexical analyzers, syntax trees and dynamic bindings are included. These principles are used to solve problems related to cross-language operations or integration not only of languages, but also platform compliant libraries to create a flexible and agile development environment. In some cases it is also relevant to migrate from existing legacy code or programming languages, including the task to remain cost and time efficient.

Although many frameworks offer highly optimized features for the above mentioned requests, they often only cover partial aspects of the requirements, mostly related to cross-language limitations. In many cases only static language parsing or dynamic syntax tree analysis is provided, but the combination of both aspects is hardly found.

These main aspects reviewed by this thesis are the usage of the Microsoft .NET framework for cross-compiling languages at runtime, building syntax trees for analysis and adaption to extend the functionality of the .NET platform including the providence of flexible and extensible ways for operations with multiple libraries in different source languages from one platform.

All the experiences and results gained from this work are published and freely available as an open-source solution on *GitHub* (https://github.com/Xpitfire/CrossCompile).

Kurzfassung

Heutzutage ist Interoperabilität und Code-Generierung eines der wichtigsten Funktionalitäten von modernen Frameworks geworden. Diese bieten eine Vielfalt an API*s* und Möglichkeiten Übersetzungen bzw. kompilierte Einheiten zu Objektinstanzen zur Laufzeit einzubinden. Im Zentrum all dieser Frameworks befinden sich meist Mechanismen wie *Reflection*, lexikalische Analysen, Syntaxbäume und dynamische Objektbindungen. Diese Elemente werden benötigt um sprachübergreifende Kompilierungen bzw. nicht nur Sprachintegrationen, sondern auch gesamte Bibliotheken in eine Laufzeitumgebung zu integrieren, welche für flexible und agile Entwicklungsmethoden benötigt werden. In manchen Fällen ist es auch nötig alten Quellcode in neue Projekte zu portieren. Dies sollte auf kostenergonomische und effiziente Weise geschehen und wenig Zeit in Anspruch nehmen.
Obwohl viele Frameworks hochoptimierte Funktionalitäten für die erwähnten Anforderungen zur Verfügung stellen, decken diese meist nur Teilaspekte der Gesamtlösung ab. In den meisten Fällen sind diese Limitierungen sprachbedingt. Es werden oftmals nur statische Modalitäten für Kompiliervorgänge offeriert oder nur dynamische Syntaxanalysen zur Laufzeit, jedoch selten eine Kombination aus beiden Aspekten.
Der Kernbereich dieser Arbeit befasst sich mit der Verwendung der Microsoft .NET Plattform für die Sprachen Cross-Kompilierung zur Laufzeit. Des Weiteren werden Syntaxbäume für Analysen und Adaptionen als Erweiterung dieser Plattform erläutert. Diese bieten Funktionalitäten für eine flexible Interaktionen mit unterschiedlichen Bibliotheken und Sprachquellen. Alle Fortschritte und gewonnenen Erkenntnisse dieser Arbeit sind veröffentlicht worden und als Open-Source Variante auf *GitHub* (https://github.com/Xpitfire/CrossCompile) frei zum Download bereitgestellt.

Chapter 1

Introduction

Throughout history many programming languages have come and gone, but in the last decade various languages have focused on some dedicated fields of applications. When we are talking about the Java programming language, then it seems natural to assume that we are seeking for cross platform source code interoperability. When Microsoft introduced the C# programming language they had set their focus only on their proprietary operating system Windows. This brought advantages, such as flexibility with byte code adaption and feature extensibility and resulted in significant performance improvements in certain fields.

1.1 Background

The examples above focus only on the initial motivation, but throughout all of nowadays common programming languages their frameworks experienced a tremendous growth. The .NET framework surpassed the mark of 5000 classes long ago and with every update the amount is getting bigger and bigger and for Java it is the same. From the perspective of a company, which made a technology decision in the past and now reconsiders their choice, this can be quite unpleasant to encounter. Developers do not only have to deal with a new language, they also have to deal with a complex and constantly growing framework.

1.2 Motivation

Besides maintainability and extensibility, code reuse is one of the main topics in nowadays companies. This does not only accord to cross platform requirements, it also occurs on the same operating system using multiple applications, which are programmed in different programming languages. Frameworks usually provide web service interoperability features and dedicated interfaces for interoperability, such as *JNI* [Ora15] for Java or *JNBridge*

[JNB15] for C#, to load cross language code from pre-compiled libraries.
In case of web services [ISO15], the frameworks require a huge overhead of pre-conditions; sockets, web servers, managed containers, etc. The *JNI* [Ora15] and *JNBridge* [JNB15] solutions bring great possibilities for code reuse with more or less little overhead, but have the disadvantage that only compiled code can be used for execution.
Since 2002, with the first release of the .NET framework, Microsoft brought in an interesting concept, called the *Common Language Infrastructure* (CLI). This enabled the possibility of cross language programming on a single framework. The frameworks libraries can be accessed throughout all the supported implementations following the *CLI* specification. When compiling C# or C++/CLI code the resulting output format is called *Intermediate Language* (IL) code. The *IL*-Code can then be run with the corresponding *Common Language Runtime* (CLR) implementation [Ecm12].
With this architecture it was also possible to derive an open source implementation (Mono) of the .NET framework, which is cross platform compatible and which enables to run C# code on Linux and Mac. Furthermore, since 2014, Microsoft Open Technologies have started a .NET Compiler Platform ("Roslyn") [Mic15], which enables the possibility to create an *Abstract Syntax Tree* (AST) to integrate own languages into the .NET platform, without requiring to work on *IL* code levels. Besides that it allows to create code at runtime using the CodeDOM implementation of the .NET framework.

1.3 Scope

The current version of the Roslyn project still requires a lot of manual coding to create an *AST* and it is also highly focused on creating code only at runtime. With that in mind, the scope of this bachelor thesis is to develop a proof of concept framework, which allows to include any kind of programming language, after parsing its corresponding *Attributed Grammar* (ATG) file at design time. This will allow to create an executable in C# referencing to foreign programming languages source code. C# classes can be attributed and will be used to dynamically bind members from foreign programming languages into newly created object instances. This requires the implementation of a framework, which binds cross language files, by interpreting the attributes from the denoted classes and pre-compiling the corresponding *ATG* files to create a parser and add the cross language members, via dynamic objects, to the current C# instance objects. The user can live code, create new objects and use newly bound members from another language, such as JavaScript, Perl or any other language, while developing in his known C# environment.

1.4 Target

This bachelor thesis targets software developers, which have experience with language integration into the .NET environment or are interested into learning how to integrate a new programming language into the .NET environment using the Roslyn project.
This thesis requires

- knowledge about *ATG*
- profound knowledge of the C# programming language
- knowledge about object-oriented, component and aspect-oriented development and
- knowledge about common programming design patterns

1.5 Goal

The following points will be covered by this thesis:

- analyze the architecture of the .NET framework
- describe the benefits of the .NET Compiler Platform ("Roslyn")
- give an overview of Compiler Generators
- describe the dynamic objects and member bindings used by *XCompilR*
- illustrate how code generation will be realized with *Coco/R* and the *Roslyn* platform
- examine how to scan, parse and bind foreign code to C# dynamic objects at design-time
- summarize the design, implementation and deployment of the software solution
- capture an outlook for future real-world applications

1.6 Structure

The first section will describe the *CLR* structure of the Microsoft .NET framework in Chapter 2. This includes a description of the *Common Type System* (CTS) and further architectural standards.
Chapter 3 describes the *Roslyn* project and provides an overview how to use it to visualize and generate code.
Chapter 4 will summarize the basics of *ATG* using *LL(1) grammar* types with *Coco/R* as a Compiler Generator.
The Chapter XCompilR illustrates the design of the *XCompilR* architecture, its processing sequences during compilation and outlines a simple language integration, specially designed for the *XCompilR* framework for interoperability between C# and other languages. This includes a more detailed re-

flection on how to use *Coco/R ATG* and how to add semantic meaning to the self-defined demo *ATG* language. Related to *Coco/R*, a description will be provided how to build an *AST* for *Roslyn*. Followed by the description of how foreign language members are included into C# object instances by using dynamic object bindings.
Chapter 6 reviews the development results and gives an overview about future work, which can be performed in this field.

Chapter 2

.NET framework

The development of the .NET framework was initiated in the late 90s and the first beta version (.NET 1.0) was published in late 2000, alongside with the *ISO* standardization of the *CLI*, an open specification developed by Microsoft and external parties [Küh13].
The .NET framework consists of multiple components, which can be accessed from multiple *CLI* compliant languages. Figure 2.1 on the following page shows an overview of the .NET main components. The *CLR* marks the base of the .NET framework stack and functions as the executing instance for the *IL* code. The features expanded from version to version and enable possibilities starting form basic algorithmic libraries, such as mathematical operations, up to concurrent multi-threaded programming solutions.
The *CLI* defines how executable code of multiple high-level languages can be run on different computer platforms due to its runtime environment abstraction. Language specific code is translated to a *CLI* and can be executed by the *CLR*. Furthermore, code that has been written in one language supported by the .NET framework can be reused from another .NET compliant language. This enables the possibility for interoperability between class libraries from C#, VB.NET, C++/CLI, etc. as long as the languages adopt to the rules of the *CTS*.
Originally the .NET framework was purposed for the Microsoft Windows platform, although nowadays there are different derivations available to other operating systems. Microsoft has initiated the .NET Core open-source project, which supports Mac OS X and Linux systems. Additionally to Microsoft's .NET Core initiative an open-source project called Mono followed the same intention, which created a .NET compatible runtime environment also available for Mac and Linux.

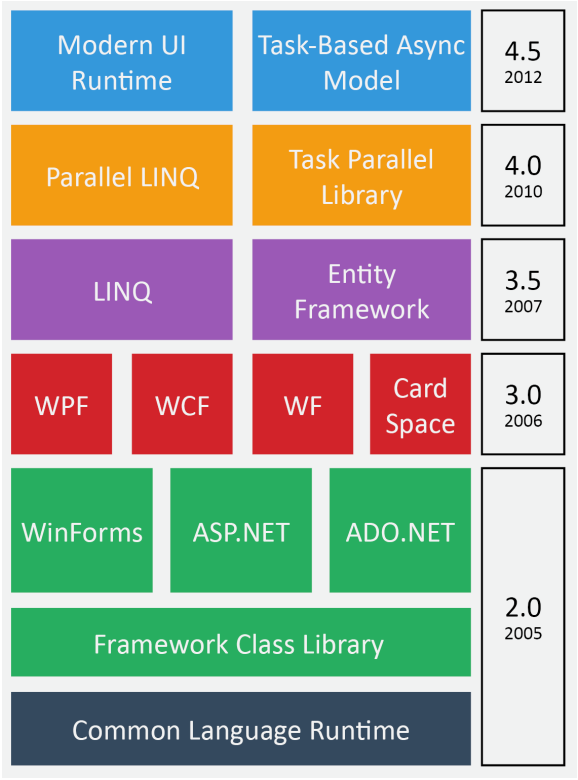


Figure 2.1: .NET framework stack overview [Wik15]

2.1 Common Language Infrastructure

The *CLI* consists of the following main elements: *CTS*, Metadata, *Common Language Specification* (CLS), *Virtual Execution System* (VES) and *Common Intermediate Language* (CIL) and specifies the executable that runs on the provided *VES*. The core of the *CLI* is defined by the *CTS*, which is used by multiple compilers for declaring, using and managing types across various languages [Sch11].

2.1.1 Common Type System

The *CTS* defines language interoperability rules for *CTS* compliant types including their external visibility across multiple assembly parts. Furthermore it describes values, which have to follow a defined contract and allows support for many programming paradigms, such as object-oriented programming, functional and procedural programming. It consists of two entity kinds, *objects* and *values*. The *value types* are self-contained types representing integers, floating point numbers or boolean logic without reference to other types, while the *object types* represent more. They do not only

CIL assembler name	class library name	Description
bool	System.Boolean	True/false value
char	System.Char	Unicode 16-bit char
object	System.Object	Object type
string	System.String	Unicode string
float32	System.Single	32-bit float
float64	System.Double	64-bit float
int16	System.Int16	Signed 16-bit integer
int32	System.Int32	Signed 32-bit integer
int64	System.Int64	Signed 64-bit integer
native int	System.IntPtr	Signed integer
unsigned int8	System.Byte	Unsigned 8-bit integer

Table 2.1: *CLS* compliant value and reference types [Ecm12]

have information about their own types, they can also store references to other object or value types and while their content may change when performing operations on them, their own type information remains consistent. These types are also known as *reference types* [Ecm12]. There are four kinds of reference types:

Object types: A self-describing reference type

Interface types: A Partial description of a value

Built-in reference types: *VES* supported integral parts of the *CTS*

Pointer types: A machine address representation of a value at compile-time, including *managed* and *unmanaged pointer*

Figure 2.2 on the next page provides an overview of the type system available by the *CTS* and table 2.1 provides an overview of the *CLS* compliant types. *Managed pointers* are not exactly data types, they can be interpreted as modifiers for data types. They do not point directly to the beginning of an object or array type, like object reference pointers, they point at the values inside the objects. These can only be used for local variables, parameters of methods or a return value of a method. The C# programming language provides an *explicit* declaration of a *managed pointer* with the *out* keyword. Whereas explicit *managed pointer* declarations as class members cannot be declared.
In comparison to the Java byte code a *CTS* compliant language operating with generics has the possibility for generic placeholder metadata information after compilation and does not lose any generic type information. These types are also statically checkable after the point of definition. In addition the *CLI* itself provides type-safe covariant and contravariant generic param-

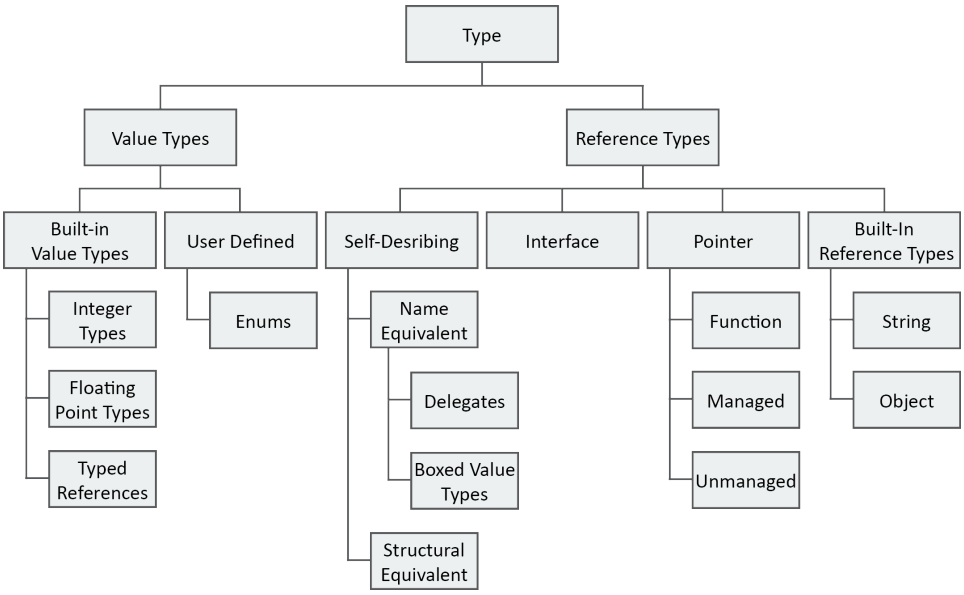


Figure 2.2: Type System [Ecm12]

eters.

2.1.2 Common Language Specification

The *CLS* represents the document specification, which describes how to integrate programming languages into the *IL* code. It includes rules for the compliance to the *CLI* for tool- and compiler-builders. These rules contain language case-sensitivity, naming patterns, supported encodings, operational functionality, method and operator overloadings.

2.1.3 Metadata

In order to declare new types for the *CLI* the *CTS* expresses such information into *metadata*. This allows the *CLI* to locate or load classes, resolve method invocations, translate *CLI* to native code and set up runtime context boundaries. This is only relevant for tool or compiler builders. However for a better understanding of how the *CLI* works it is helpful to know that component-specific metadata contains self-describing information for the reference related objects. *CLI* components and other files are packaged together and deployed as an *assembly*, which contains all the logical information for functionality and which can be run under the *CLR*.
Figure 2.3 on the next page illustrates an overview of an assembly structure referencing multiple internal modules. As a simple abstraction a module includes *metadata* and *IL* code. The *metadata* handles the logical links be-

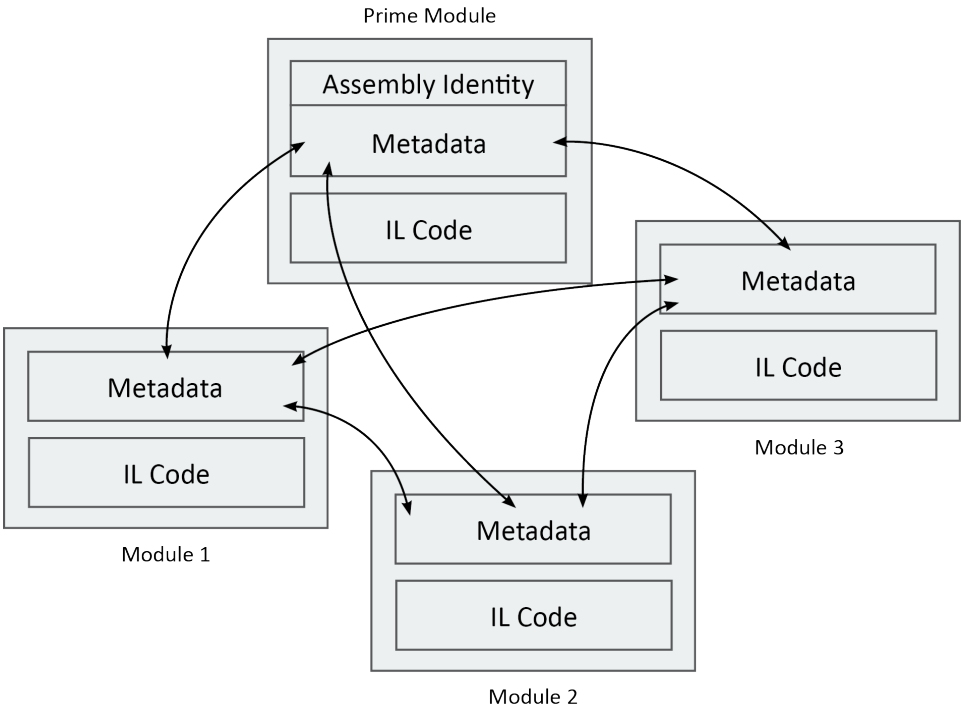


Figure 2.3: IL assembly overview [Lin14]

tween all used modules. The *prime module* represents an executable unit which can be run by the *CLR*.

2.1.4 Common Intermediate Language

The *CIL* or simply *IL* represents the lowest-level programming language within the Microsoft .NET framework and is the base for all above implemented languages, such as C++/CLI or C#. The *IL* code includes all the *CLI* defined specifications to provide a wide programming language functionality coverage, including the concept of classes, field declarations, module imports, exception handling and further more. The following code snippet shows a simple *Hello World* program, written in *IL* code.

```
.assembly extern mscorlib {}
.assembly Hello {}
.module Hello.exe

.class Hello.Program
extends [mscorlib]System.Object
{
  .method static void Main(string[] args)
  cil managed
  {
```

```
      .entrypoint
      ldstr "Hello World"
      call void [mscorlib]System.Console::WriteLine(string)
      ret
    }
  }
```

The main characteristic of the *IL* code is the dot prefix notation at the assembler directives. The first directive informs the assembler that the *mscorlib* is required, which is used to access the upcoming *WriteLine* method.

```
.assembly extern mscorlib {}
```

By analyzing the first line it is now becoming clear that the *IL* code itself has access to the entire .NET framework *managed code* libraries, regardless in which higher-level language they where previously written. The next lines declare the name of the assembly and that it is executable.

```
.assembly Hello {}
.module Hello.exe
```

Afterwards the class name and the base class object from which it inherits are declared.

```
.class Hello.Program
extends [mscorlib]System.Object
```

The following directive specifies the main program entry method.

```
  .method static void Main(string[] args)
  cil managed
  {
    .entrypoint
```

The next step is the declaration of a stack variable with the included string *Hello World* and the call of the framework method *WriteLine* to print to the console.

```
    ldstr "Hello World"
    call void [mscorlib]System.Console::WriteLine(string)
```

The last *ret* declaration defines the return statement of the method.

2.1.5 Common Language Runtime

The *CLR* is the virtual machine component of the .NET framework and executes *CLI* compliant assemblies. A language compiled for targeting the *CLR* is called *managed code* and it enables features for cross-language integration, exception handling, services for various security aspects, versioning and deployment support. It also handles references for objects and provides *Garbage Collection* (GC). The *CLR* is a particular implementation of the *CLI* specified *VES*.

2.1.6 Virtual Execution System

The *VES* is the standardized virtual machine specification of the *CLI*, which implements the *CTS* models to load, manage and execute assemblies. It uses the *metadata* to connect the separately generated modules at runtime. The supported data types are listed within table 2.1.

2.2 Conclusion

The lowest layer represents the *IL* code itself and is relevant to add new languages to .NET or to build tools and compilers for new languages. The Roslyn chapter will illustrate a more profound way to analyze, visualize and even integrate code into .NET using *Roslyn*, Microsoft's Compiler Platform.

Chapter 3

Roslyn

Roslyn is Microsoft's new Compiler Platform of 2012, whereas the first Community Technology Preview (CTP) was released in October 2011. It has been developed for several years. On base of *Roslyn* also the compilers for C# and Visual Basic have been re-written in their own representative language to fully support the new language services provided. Microsoft refers to the *Roslyn* project as a Compiler-as-a-Service platform. It does not operate as a black box compared to conventional compilers, performing some operations on committed input values and returning compiled output binary code. *Roslyn* offers many possibilities to interact and intercept with the actively analyzed input and transparently modeling atop of it. It exposes information regarding the source code parsing process, the meanings from semantic analysis, the bindings between processed elements and finally the *IL* emitting outcome.
This enables the possibility for all developers using the Visual Studio platform to create and provide new *NuGet* or *VSIX* extensions and even create their own *IntelliSense* solutions. For C# or Visual Basic developers the knowledge about *IL* code and assemblers becomes irrelevant, because *Roslyn* handles all the translation between *AST* and *IL* code. The developer can focus on coding in his language of choice and does not have to worry about lower-level conventions.

3.1 Compiler-as-a-Service

Roslyn exposes the compiler code analysis for C# and Visual Basic developers via three main *API* layers. As shown in Figure 3.1 on the following page.
The *Compiler Pipeline* layer handles the lower-level parsing, symbol analysis and metadata processing including the bindings and *IL* emitting. The *Compiler* API layer offers access to *AST* representation of the processed data and functions as a higher level abstraction for C# or Visual Basic develop-

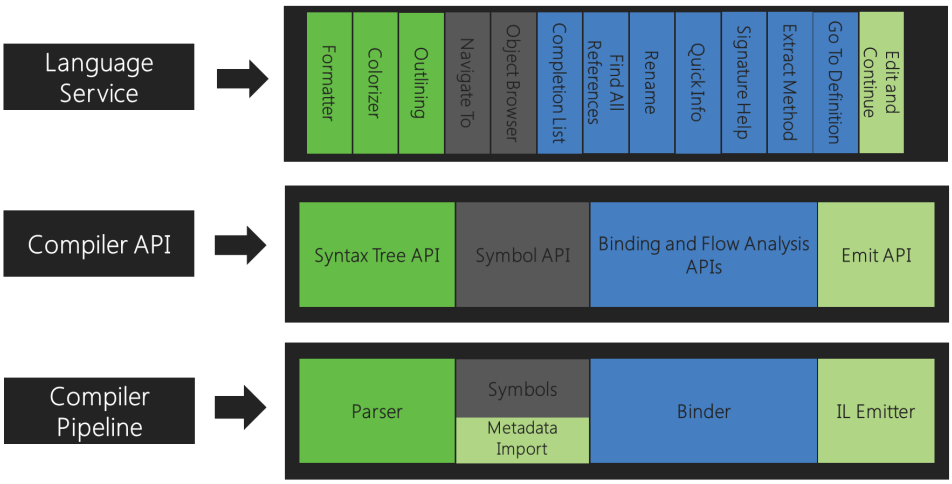


Figure 3.1: Roslyn *API* layers [Mic15]

ers. The *Language Service* layer provides a higher-level tool set to operate on the *Compiler* API data including navigational operations, information and formatting possibilities.

3.1.1 Compiler Pipeline

The *Compiler Pipeline* processes the input source in different phases, starting with the parsing process, where the source is tokenized into the corresponding grammar language rules. Afterwards the metadata is analyzed to extract the required symbols, also called declaration phase. The Binder matches the symbols to the corresponding identifiers and at the end the Emitter builds a complete assembly.

3.1.2 Compiler API

In equivalence to the previously described phases of the *Compiler Pipeline* the *Compiler* API offers a representative object model in the current target language to access the necessary information. The *Parser* phase representation is translated by the *Syntax Tree* API as *AST* model, followed by metadata models for the symbols and bindings, and the *Emit* API abstracts from the corresponding *IL* byte code producer.

3.1.3 Language Service

The *Language Service* adapts to the underlaying layers and phases. For instance it uses the symbol table for the *Object Browser* or navigation features, and the syntax tree representation for outlining and formatting features.

3.2 API

Roslyn is built as a two layer *API* set, the compiler *API* layer and the workspace *API* layer. The compiler layer contains the object models consisting of the extracted information about the individual phases and the syntactic and semantic data parsed from the source. It offers an immutable snapshot of a single invocation of a compiler. This layer is independent from any Visual Studio components. This allows user-defined diagnostic tools to connect to the actual compilation process for error and warnings information. The workspace *API* resolves project dependencies and uses the compiler layer to provide object models for analysis and refactoring solutions [Mic15].

3.3 Syntax Tree

The *Syntax Tree* API is the core component of the *Roslyn* framework. It binds all the provided features to the abstract object model classes, known as the *AST*. Code analysis, refactoring, *IDE* components, source information, grammatical constructs and many more features converge to one point, which is the *Syntax Tree* API. It is also important to know that all parsed output can be reverted into its original source text (completely round-trippable). Furthermore the syntax tree is thread-safe and immutable, which allows multiple users to interact concurrently without interfering or ending up in any race conditions. This also means that no modifications can be submitted on the syntax trees. Factory methods provide additional snapshots of the syntax tree to operate indirectly with them, to the cost of little memory overhead (see also section *Immutable Syntax Tree* 5.4).
To illustrate the main components and features of the *Syntax Tree* API the following *Hello World* program written in C# will serve as a practical illustration.

```
using System;
namespace Demo {
  class Program {
    static void Main() {
      Console.WriteLine("Hello World!");
    }
  }
}
```

Roslyn also provides possibilities to visualize existing code sections as syntax tree using the *.NET Compiler Platform SDK*, which is available as a *NuGet* package for Visual Studio. After installing the additional package, Visual Studio adds a new menu item in the *View / Other Windows / Syntax Visualizer* tab. The *Syntax Visualizer* creates the following syntax tree output from the previously shown code as shown in Figure 3.2.

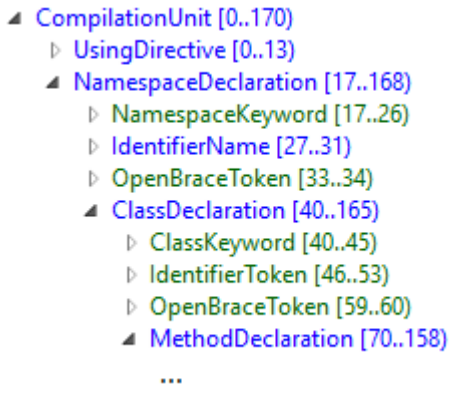


Figure 3.2: *Hello World* syntax tree components

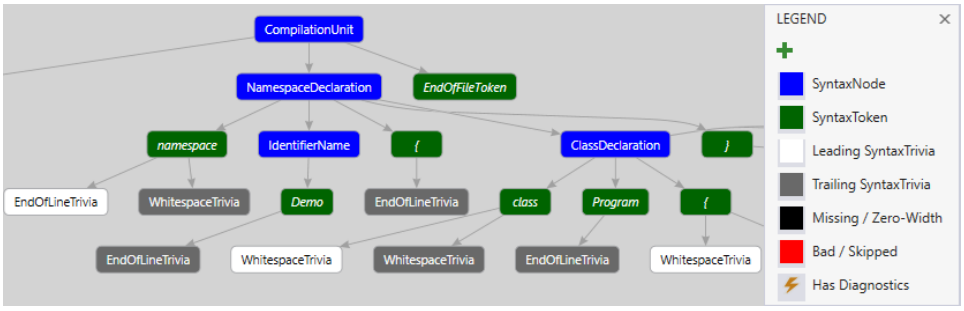


Figure 3.3: *Hello World* visualized syntax tree

It is also possible to view a visual representation of the syntax tree as shown in figure 3.3. The legend on the right hand side of figure 3.3 provides an overview of the syntax tree elements resulting from a parsing procedure. Each *Roslyn* syntax tree consists of the following elements:

Syntax Nodes for declarations, statements, clauses and expressions (*SyntaxNodes*, see Figure 3.3)

Syntax Tokens to represent terminals, the smallest fragments of code (*SyntaxTokens*, see Figure 3.3)

Syntax Trivia for whitespaces, comments and preprocessor directives, which are the insignificant code elements that can be skipped after parsing (*Leading SyntaxTrivia and Trailing SyntaxTrivia*, see Figure 3.3)

Spans for the text positioning and number of characters used by a node, token or trivia (*Missing / Zero-Width*, see Figure 3.3)

Kinds to identify the exact syntax element corresponding, which can be cast to language-specific enumerations

Errors for syntax, which is not grammar conform (*Bad / Skipped*, see Fig-

ure 3.3 on the preceding page) and

additional diagnostic elements can be attached for development or debugging (*Has Diagnostics*, see Figure 3.3 on the previous page)

3.3.1 Syntax Nodes

The main *Syntax Node* types are derived from the *SyntaxNode* base class. The set of *Syntax Nodes* is not extensible. These classes form the construct to create declarations, statements, clauses and expressions. They are non-terminal, which means they have sub-nodes that can be navigated through typed *Properties* or methods.

```
/* tree is given after parsing the Hello World example with the
    Microsoft.CodeAnalysis.CSharp.CSharpSyntaxTree.ParseText method */
var syntaxRoot = tree.GetRoot();
var Demo = syntaxRoot.DescendantNodes().OfType<ClassDeclarationSyntax>()
    .First();
```

3.3.2 Syntax Tokens

The *Syntax Tokens* are terminals and cannot be parents of other nodes or token types. They represent identifiers, keywords, literals and punctuations. It offers *Properties* to access the values parsed from the source input.

3.3.3 Syntax Trivia

Syntax Trivia represent insignificant text portions, which can occur as whitespaces, comments and preprocessor directives at any position within the source input. They are not part of normal language syntax and will not be added as children of *Syntax Nodes*. They also do not have a parent node. The *Syntax Trivia* base class is called *SyntaxTrivia*.

3.3.4 Spans

The nodes, tokens and trivias contain *Spans* for the positioning and number of character occurrences to associate the correct location from the source input. This may be used for debugging or error information determination.

3.3.5 Kinds

Kinds are used as properties for nodes, tokens and trivias to distinguish the corresponding types and provide the correct conversions. The *SyntaxKind* class can be represented as an enumeration type in the target language C# or *VB*.

3.3.6 Errors

When parsing the source input, errors may occur, which can be located and marked as wrong or incomplete, e. g. a token is missing or is invalid. The parser can skip invalid tokens and continue to seek for the next valid token to continue parsing. Skipped tokens will be attached as *Trivia Node* of *Kind SkippedToken*.

3.4 Using the Roslyn API

Roslyn provides a *rich API* to create *AST* compilation units by code. It uses the factory pattern and provides many static classes and methods or enums, which can be used via *chaining* to build these solutions. *Chaining* is a fluent method call design used in object-oriented programming languages, whereas each method returns an object, allowing to call the next statement without requiring variables storing the intermediate results. The following code snipped shows an example how to use the *Roslyn API* to make a simple class declaration.

```
var tree = SyntaxFactory.CompilationUnit().AddMembers(
  SyntaxFactory.NamespaceDeclaration(
    SyntaxFactory.IdentifierName("Example")).AddMembers(
      SyntaxFactory.ClassDeclaration("Demo").AddMembers(
        SyntaxFactory.MethodDeclaration(
          SyntaxFactory.PredefinedType(
            SyntaxFactory.Token(SyntaxKind.VoidKeyword)), "Main")
          .AddModifiers(SyntaxFactory.Token(SyntaxKind.StaticKeyword))
          .AddModifiers(SyntaxFactory.Token(SyntaxKind.PublicKeyword))
          .WithBody(SyntaxFactory.Block()).AddBodyStatements(
            SyntaxFactory.ReturnStatement())
      )
    )
  );
```

3.5 Semantics

Although the *Syntax Tree* representation of the source input is offering many features, it is only analyzing the lexical and syntactic structures. To fully cover all the aspects of programming languages it is also necessary to define their semantics, which mark their behavior. Designations of local and member variables can overlap each other and differ between their scopes and actions rules have to be embedded within those closures. To do so, *Roslyn* uses *symbols*. A symbol represents an element, which carries metadata received from the source input. These can be accessed via the provided symbol table also represented in a tree structure, starting from the root element, which is the global namespace. The symbols are derived from the ISymbol

interface, whereas the *Properties* and methods are provided by the compiler. The symbols offer namespaces, types and members between source code and the metadata and their language concepts are similar to the *Reflection API* used by the *CLR* type system. It is also important to know that the access to the semantic model tree triggers a compilation, which means it is costlier in comparison to the syntax tree access.
Every symbol contains information about

- the location of the declaration (in source or in metadata)
- in which namespace or type this symbol exists
- the information if the symbol is abstract, static, sealed, etc.

The following code snippet shows how to add semantics information to the previously declared *Hello World* example.

```
/* tree is given after parsing the Hello World example with the
    Microsoft.CodeAnalysis.CSharp.CSharpSyntaxTree.ParseText method */
var mscorlib = MetadataReference.CreateFromFile(typeof(object).Assembly.
    Location);
var compilation = CSharpCompilation.Create("DemoCompilation",
    syntaxTrees: new[] { tree }, references: new[] { mscorlib });
var model = compilation.GetSemanticModel(tree);
```

3.6 Conclusion

Roslyn is a rich compiler service platform which enables deep access to Microsoft's .NET framework and its supported language sub-sets. Although *Roslyn* offers a variety of low-level possibilities to building *AST*, it nicely abstracts from these layers and supplies a clean and intuitive *API*.
The next chapter 4 will show how to declare and use *ATG* in foresight to integrate new languages to the .NET framework.

Chapter 4

Attributed Grammar

Attributed grammars describe the syntactic and semantic rules for any given programming language, which have to be fulfilled to ensure correctness in execution of the source code. The rules declared within the grammar files are used by compiler generators to provide the necessary parsers and lexical analyzers that will scan the source input and create the semantically specified output. The output values can have binary, text or even other language compliant formats.
This chapter will cover conform *LL(1) grammar* files, which will be used for the development of *XCompilR* to embed a *CLR* non-compliant language into the .NET framework by transforming the *Coco/R AST* to a *Roslyn* conform *AST* [Prü13].
Coco/R has been developed by the Johannes Kepler University Linz (Austria). The *Coco/R* compiler generator is distributed under the GNU General Public License and comes with a executable file available for the Windows platform. It also offers an *ATG* file describing itself for the C# language, which then can be used to generate C# parser and scanner sources. This means it is possible to adapt the compiler generator grammar template to even customize the generated compiler compiler.
For this thesis the compiler generator templates of *Coco/R* have been modified to integrate the resulting *Parser* and *Scanner* class into the *XCompilR* solution. The mentioned *Coco/R* generator templates have a *.frame* extension and describe the static text regions of the generated output. The text region for the *Parser*- and *Scanner*-class have been extended to derive from abstract defined classes of *XCompilR*, which then can be instantiated by *Reflection* at runtime and type-casted to an accessible type.

4.1 Coco/R

The development of *Coco/R* started in the early 80s. Since then the compiler generator has found wide application all over the globe and has been ported

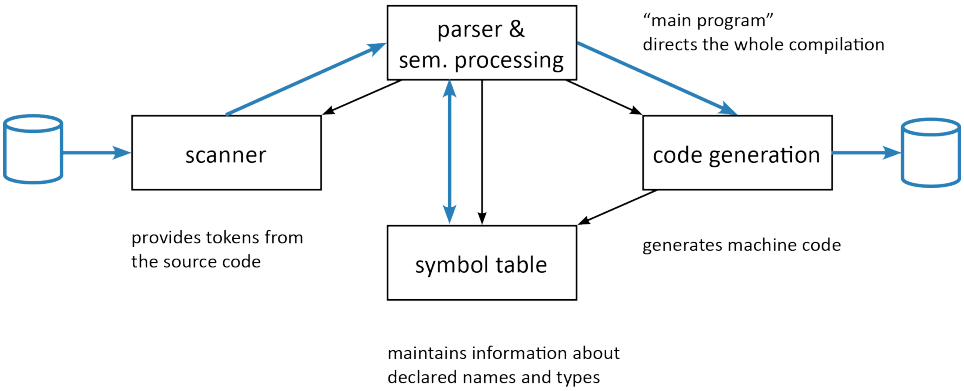


Figure 4.1: Structure of a Compiler (blue lines represent the data flow, black line represent the usages) [JKU15]

to many known programming languages [Han03]. Figure 4.1 provides an overview of how the compiler is structured.
The *Scanner* processes the source text character by character and passes the information to the *Parser*, which includes semantics information to generate the output code. To store information about the declarations and types found within the source text, the *Parser* uses a *symbol lookup table* [Alf88].
The *ATG* used by *Coco/R* to generate the required parsers is declared in *EBNF* notation and generates a recursive descent parsers [Set96]. Additionally to the *EBNF* notation *Coco/R* offers own description extensions to separate logical parts, such as character, tokens, ignorable sequences and the production notation for better readability. The used sections are:

- *COMPILER* and *END* (name of the *ATG* file the including start/end enclosure)
- *CHARACTERS* (used to define basic digit and character sequences)
- *TOKENS* (used for creating basic token sequences by reusing the *CHARACTERS* definition)
- *PRAGMAS* (used for writing preprocessor directives [optional])
- *COMMENTS* (used to define the comment character and their spans behavior [optional])
- *IGNORE* (used to define character, which can be skipped during the scanning procedure [optional])
- *PRODUCTIONS* (used for the *EBNF* notation with additional semantics)

The semantic processing notation in *Coco/R* is clasped by an open parenthesis with a suffixed dot for the opening and a closed parenthesis with a prefixed dot for the closing as shown below.

```
...
PRODUCTIONS
  Demo                          (. int n; .)
  = { Expr<out n>               (. Console.WriteLine(n); .)
...
```

Furthermore the *Coco/R* notation allows non-terminals to attach attributes in angle brackets, which are used to pass arguments similar to parameters of the symbols. To write helper classes and methods and import external modules for the semantic statements. The compiler description also provides regions for using clauses *[UsingClauses]* and global fields and member declarations *[GlobalFieldsAndMethods]*. The following illustration offers an overview of the *Coco/R EBNF* structure.

```
[UsingClauses]
"COMPILER" ident
[GlobalFieldsAndMethods]
ScannerSpecification
ParserSpecification
"END" ident "."
```

The *ScannerSpecification* region contains the *CHARACTERS*, *TOKENS*, *PRAGMAS*, *COMMENTS* and *IGNORE* statements, whereas the *ParserSpecification* consists of the *PRODUCTIONS* statements.
The following code snipped illustrates a very simple, but complete *Say Hello ATG* written for *Coco/R*. The source text committed to the generated compiler (*Parser* and *Scanner*) has to provide a constant *say* keyword followed by any word, also including prefixed number sequences, starting with an upper-case or lower-case character. The resulting output source text is calling the C# *Console* class and printing the text *Hello, <ident>* to the console.

```
using static System.Console;

COMPILER SayHello

CHARACTERS
  letter    = 'A'..'Z' + 'a'..'z'.
  digit     = '0'..'9'.
  cr        = '\r'.
  lf        = '\n'.
  tab       = '\t'.

TOKENS
  ident     = letter {letter | digit}.

IGNORE cr + lf + tab

PRODUCTIONS
SayHello =
"say"             (. Write("Hello, ");    .)
ident             (. Write(t.val + "\n"); .).
```

```
END SayHello.
```

4.2 Conclusion

By writing *LL(1) grammar* with *Coco/R* it is possible to build parsers and lexers to scan languages files and map their behavior by defining semantic elements to create *AST*.
The next chapter 5 will cover how this thesis combines all the previously described technologies to create a cross-language compiler for the .NET framework.

Chapter 5

XCompilR

The XCompilR offers language interoperability by analyzing the source languages, building *AST* and applying the semantics, defined by the *ATG* files, at runtime. The main language used by the *XCompilR* is C# although it would be possible to also implement this framework in *VB*.

5.1 Problem of Language Interoperability

The conventional way of integrating languages into the .NET framework is to fulfill the specified *CLI* contracts and write a compiler to convert the source language into the required target *IL* code. This code is then executed by the *CLR*. Writing *IL* code is a very low-level task and requires deep knowledge about memory management, metadata bindings, assembly management, etc. *Roslyn* already provides a good abstraction to this low-level subset, but offers only services for the C# and *VB* languages due to their compiler reimplementation atop of *Roslyn*. To add new languages at design or compile time via Roslyn is a lot of manual work and tightly connected to the C# or *VB* environment. This is a very inflexible way to incrementally adapt new languages to the current .NET environment. On the other hand by using *Coco/R* to create language parsers, which translate from a source language to another target language, for instance *IL* code, it is hardly possible to integrate the generated output into an existing development design-time workflow. Classes have to be manually loaded via *Reflection* to access these sources. The *XCompilR* solution resolves this problem by embedding the compiler generator *Coco/R* into the project and creating a bridge between *Roslyn* and the written *ATG* files. The user of this framework can write *LL(1) grammar* for *Coco/R* to build a *CompilationUnit* for *Roslyn* without tightly coupling to the C# or *VB* development environment.

5.2 Requirements

The *XCompilR* uses not only object-oriented paradigms, it also uses aspect-oriented paradigms [Geo01] for exception handling and logging. This framework uses *PostSharp* as a pattern-aware extension, to develop custom C# *Attributes* and obtain the possibility to intercept the runtime execution at different stages. Pattern-aware development focuses on code that can be automated and encapsulates these code sequences in *easy-to-use* aspects. They can be attached on methods, classes, properties, etc. This method avoids code to becomes boilerplate and provides better separation between the core logic of an implementation and the maintenance code. *PostSharp* is available as a *NuGet* package.
To use the *Roslyn* framework, an additional download of packages via *NuGet* (*.NET Compiler Platform SDK*) is also required.

5.3 Design

This framework requires to translate a *Coco/R ATG* to the *Roslyn* provided *AST*. This resolve the bindings to integrate the compiled units into the .NET framework and to be able to use external libraries from non-*CLI* compliant languages. This requires three main parts:

- The first part is to embed *Coco/R* into the *XCompilR* framework to trigger and handle the parsing procedures via C#. This requires modifications on the parser generator template to create externally visible accessory methods. These methods will be used to generate the required target language *Parser* and *Scanner*.
- The second part is to create an *ATG* for the source language, which uses the *Roslyn API* and matches terminals and non-terminals via semantic actions. This creates a *CompilationUnit* which can be compiled to .NET *IL* code via *Roslyn*. To dynamically extend the available *CLI* non-compliant language-pool it is necessary to create a Visual Studio extension. This extension loads new *ATG* files and triggers the *Parser* and *Scanner* generation and provides code completion at design-time. How to create the corresponding Visual Studio extension is not topic of this thesis.
- The last part represents the object instantiation and binding creation, which is available at design-time and during the execution. To access the imported external language libraries for C#, extendable placeholder classes have to be declared and attributed with the source code path and the corresponding source language. The attributed class will load the source code and create a *CompilationUnit* for the .NET framework. At this point it is important to mention that *Roslyn* syntax trees are immutable, which means that every adaption on the current tree

will result to a new *SyntaxTree* instance (the next subsection provides a more detailed description about immutable syntax trees5.4).

5.4 Immutable Syntax Tree

In the early design phase of the *Roslyn* project the *Roslyn* team figured out that syntax trees will be the most important parts for a user. Furthermore multiple instances could concurrently analyze and even modify the parsed syntax trees. Therefore they had to ensure that the provided data does not become corrupted during these operations. The resulting design decision concluded to ensure the following characteristics:

- Immutability
- Persistence
- Form and accessibility of the tree
- Low performance costs when accessing child nodes
- Offset positioning possibility between trees and input text

The most important characteristics are immutability and persistence. *Roslyn* performs analysis at every user keystroke or received data input. To avoid multiple re-parsing and analyzing cycles, a previously parsed syntax tree does not change its shape or properties. Instead of rebuilding the entire tree when modifying existing nodes, a new tree is created which is a copy of the previous one including the new changes. More precisely the implementation of *Roslyn* uses two parse trees. Whereas one is built *bottom-up*, persistent, immutable and has no parent references and the other is *top-down* built around the first tree. The second one is an immutable *facade*, which will be thrown away during modifications. The tradeoff by this design are the higher memory costs if the secondary mentioned tree becomes to large.
In practical terms of use, adding a single node to an existing tree will create a new *CompilationUnit* instance as shown below.

```
var compilation = SyntaxFactory.CompilationUnit();
/* This creates a new tree with the newly added class member; the
    previous compilation instance is not affected */
compilation.AddMembers(SyntaxFactory.ClassDeclaration("Demo"));
```

The following code reassigns the new syntax tree after the modification.

```
var compilation = SyntaxFactory.CompilationUnit();
compilation = compilation.AddMembers(SyntaxFactory.ClassDeclaration("
    Demo"));
```

5.5 Architecture

XCompilR has been developed in a *test-driven* way. Not only to be able to test the given components, but also to focus on the usability key points and

avoid unnecessary object construction chaining. The main idea is that the object instantiation should only require minimal user effort. To achieve this goal, *XCompilR* uses aspect-oriented concepts [Iva04]. The user of the framework only needs to declare a new placeholder class and assign C# *Attributes* and define the target compilation properties. The library required for attributing the classes and to control the parsing process is available in the *XCompilR.Core* assembly. In order to be able to handle multiple languages it is required to control the parsing procedure by code, which concludes to embed the *Coco/R* parser generator into the existing project structure. The parser generator translates the *ATG* files and provides the required parsers to process the new language related input sources. To enable the possibility to replace the parser generator and abstract from proprietary code, *Coco/R* has been placed in a separate assembly (*XCompilR.ParserGen.CocoR*) and the access is defined via an interface and loaded via *Reflection*. This offers higher flexibility for class extensibility and separates the code generation unit from the logical core unit.
This results to the following design as shown in figure 5.1 on the next page. *XCompilR* is structured in six main assemblies:

- *XCompilR.Core*
- *XCompilR.Library*
- *XCompilR.ParserGen.Library*
- *XCompilR.ParserGen.CocoR*
- *XCompilR.IntelliSense*
- *XCompilR.Tests*

XCompilR.Core is the primary library, which is used by developers building on top of the *XCompilR* framework. It contains the *XCompileAttribute* class used to declare the placeholder classes for the bindings. *XCompilR.Library* defines abstract classes to ensure loose coupling between the generated parsers form the *ATG* files for the language translation and the core library usage. The generated assemblies (for example to parse JavaScript, Perl or Python sources) implement the abstract classes from the *XCompilR.Library* library. The *Coco/R* (*XCompilR.ParserGen.CocoR*) implementation inherits from abstract classes defined in the *XCompilR.ParserGen.Library* library. The *XCompilR.IntelliSense* solution offers a Visual Studio extension and provides the loading and unloading of available languages, created from *ATG* files, which can be used by the core library. The implementation of Visual Studio extensions will not be covered by this thesis. The generated libraries are precompiled and statically available for the *XCompilR.Tests* library. *XCompilR.Tests* is used for unit testing of the framework.

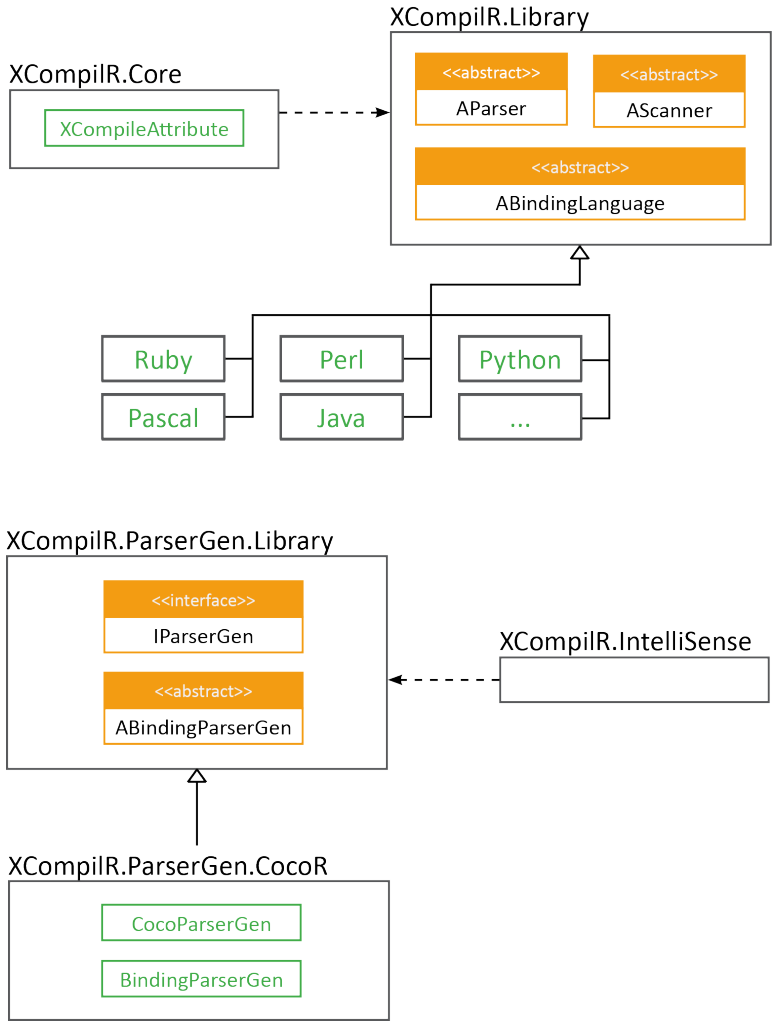


Figure 5.1: *XCompilR* project structure

5.6 Implementation

In order to bind the import external languages via *Attributes* it is required to write a custom class handling the object instantiation process. The *XCompileAttribute* class marks the placeholder classes as the binding targets and receives parameters declaring the source language, target assembly naming, target namespace and target main class entrance point (if available). As an outcome of this design the following code shows an example how the *XCompileAttribute* is used:

```
[XCompile("XCompilR.JavaScript", "test.js", TargetMainClass = "Demo",
    TargetNamespace = "Test")]
public class DemoBinding : XCompileObject { }
```

In addition to the declared *XCompileAttribute* it is necessary to inherit from the *XCompileObject*, which is the base class for all imported language bindings. Furthermore the *XCompileObject* triggers the object creation at runtime, which loads the external language sources and binds them to the current class instance. The created *XCompileObject* is a dynamic object allowing to dynamically assign member declarations and assignments at runtime, which bypass the compile-time type checking (see dynamic objects in the following subsection 5.6.1.
A positive side effect of the *XCompileAttribute* annotation is that it enables the possibility to create a Visual Studio *IntelliSense* extension, searching for the declared *Attributes* and triggering precompilation cycles to offer code completion at design-time.

5.6.1 Dynamic Objects

In C# 4.0 the *dynamic* keyword was introduced and enables the declaration and assignment of members of any given type at runtime without the necessity to cast between reassigned value types. The declared members bypass the compile-time type checking and the dynamic objects offer a flexible way to declare members similar to the *JavaScript var* keyword. This provides a simplified access to *COM* API*s*, dynamic API*s* such as *IronPython*, *HTML DOM*, etc. The following example illustrates the dynamic assignment using an *ExpandoObject* instance (*ExpandoObject* represents a simple object, whereas members can be added and removed at runtime).

```
dynamic sample = new ExpandoObject();
/* Test is not a provided member of ExpandoObject, it is dynamically
    declared and assigned */
sample.Test = 10;
sample.Test = "Hello, world!";
Console.WriteLine(sample.Test);
sample.Test = (Action)(() => { Console.WriteLine("Hello again"); })
sample.Test(); /* Prints out Hello again to the console. */
```

The *XCompilR* framework uses dynamic objects for lazy loading bindings between the non *CLI* compliant languages and the current instances created by the user. Because *ExpandoObject* is a sealed class which cannot be inherited, *XCompilR* uses another class (*DynamicObject*) to customize the required constructor and properties for the bindings. The base class for all bindings in *XCompilR* is called *XCompileObject*, which is an abstract class and inherits from *DynamicObject*. *XCompileObject* provides a no-argument constructor verifying the attribute assignment of the derived class and calling the *BindMembers* method of the *XCompileAttribute* to perform the language translation.

```
protected XCompileObject() {
  /* Verify if the XCompileAttribute is available. */
```

```
    var attributeArray = (XCompileAttribute[])GetType().
      GetCustomAttributes(typeof(XCompileAttribute), false);
    if (attributeArray.Length != 1) {
      throw new XCompileException("Invalid attribute notation on target
      class!");
    }
    /* Commit this object for binding. */
    attributeArray[0].BindMembers(this);
  }
```

5.6.2 XCompileAttribute

This *Attribute* loads the supported languages and calls the parsing procedure to process the committed source file and binds the members to the *XCompileObject*. To fulfill this task it inherits from the C# *Attribute* base class and declares the contract related attributes. To create custom attributes in C# it is required to declare the class as *Serializable*. To prevent inappropriate usage of this attribute the *AttributeUsage* declaration restricts the usage only for classes and the *ContractClass* attribute request, that only classes typed as *XCompileObject* are allowed to be signed with this attribute.

```
  [Serializable]
  [AttributeUsage(AttributeTargets.Class)]
  [ContractClass(typeof(XCompileObject))]
  public class XCompileAttribute : Attribute
```

The constructor of the *XCompileAttribute* receives the user defined values and loads the required parser assembly for the ongoing computation via reflection.

```
  public XCompileAttribute(string bindingLanguageAssembly, string
      sourceFile) {
    /* Load import language assembly via reflection. */
    Assembly assembly = Assembly.Load(bindingLanguageAssembly);
    Type type = assembly.GetType($"{bindingLanguageAssembly}.
      BindingLanguage");
    Language = (ABindingLanguage)Activator.CreateInstance(type);
    _sourceFile = sourceFile;
    /* Verify committed values. */
    if (TargetNamespace == null || TargetNamespace.Equals(string.Empty))
      TargetNamespace = bindingLanguageAssembly;
    if (_sourceFile == null || _sourceFile.Equals(string.Empty))
      throw new XCompileException("Invalid source file path!");
  }
```

The *ABindingLanguage* abstract class has all the required properties to be able to load the parsers.
The *BindMembers* method is assigned with the *XCompilRExceptionHandler* attribute which handles the exceptions that can occur during the computation (the exception handling will be covered more detailed in the following section *Exception handling and logging* 5.6.4).

```
[XCompilRExceptionHandler(typeof(XCompileException))]
public void BindMembers(dynamic bindingObj)
```

The main task of this method is to initialize the parser with the dynamic object for the bindings and trigger the parsing process. Afterwards it builds an assembly containing the new imported library information and attaches these properties to the current dynamic object instance.
The following example illustrates the parser initialization and usage.

```
var parser = Language.Parser;
parser.BindingObject = bindingObj;
parser.Parse(_sourceFile);
```

Every provided parser for the supported languages is derived from an abstract class *AParser* (see section *Parser* 5.6.3) and contains a *CompilationUnitSyntax* (covered in chapter *Roslyn* 3) property and the instance of the currently processed dynamic object. The resulting outcome of a successful parsing procedure is a fully qualified *Roslyn AST*, which can be used for the assembly generation. To create an assembly it is first required to use the *CSharpCompilation.Create* method to create a *CSharpCompilation* object from the *CompilationUnitSyntax.SyntaxTree* property and afterwards emit and load the data stream from the memory.

```
/* Creates a dll compilation from the syntax tree received from the
    parser and adds references at runtime including metadata references
    of System library. */
var mscorlib = MetadataReference.CreateFromFile(
  typeof(object).Assembly.Location);
var compilation = CSharpCompilation.Create(
  $"{TargetMainClass}.dll",
  references: new[] { mscorlib },
  syntaxTrees: new[] { parser.CompilationUnitSyntax.SyntaxTree });
compilation.GetSemanticModel(
  parser.CompilationUnitSyntax.SyntaxTree, false);

/* The compiled code is emitted into memory stream which is used to
    create a assembly at runtime. */
Assembly assembly;
using (var stream = new MemoryStream()) {
  compilation.Emit(stream);
  assembly = Assembly.Load(stream.GetBuffer());
}
```

Finally the new assemblies are bound to the dynamic object instance and include representative helper methods for the object creation. The type casting can be performed via reflection and by using the attached assembly.

```
bindingObj.Add(Language.AssemblyName, assembly);
bindingObj.Add($"CreateInstanceOf{TargetMainClass}",
  new Func<object>(() =>
    assembly.CreateInstance($"{TargetNamespace}.{TargetMainClass}")));
```

The get an overview of the entire object instantiation sequence see figure 5.2 on the following page.

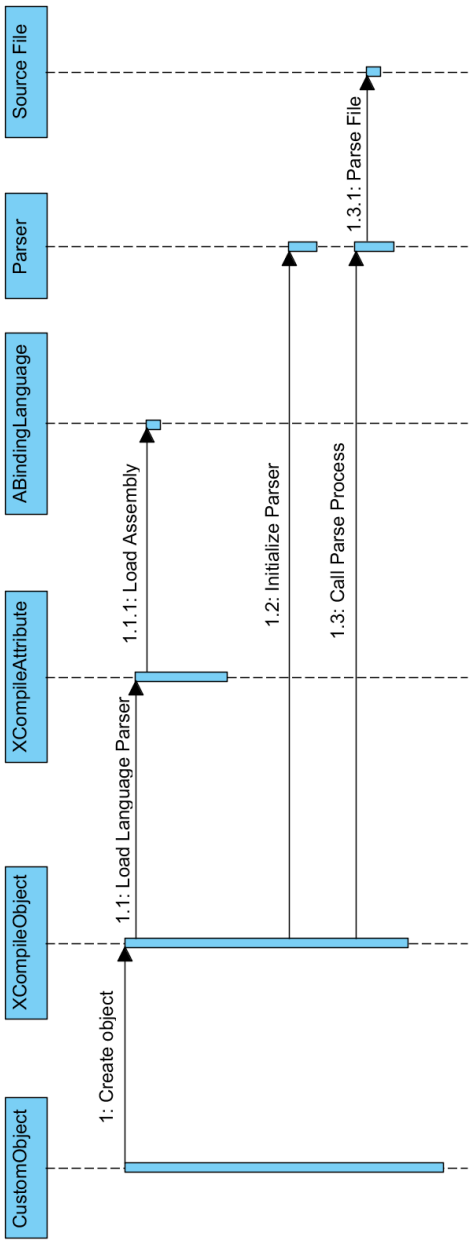


Figure 5.2: Object creation illustration

5.6.3 Parser

In order to be able to cross-compile multiple languages with *XCompilR* it is required to use a parser generator, such as *Coco/R*. The parser generator compiles the language specific parsers from the *ATG* files and builds assemblies that can be used by the *XCompileAttribute*. *XCompilR* is foreseen

to be used in combination with a Visual Studio extension, which manages the loading and unloading of *ATG* files. It also triggers the *Coco/R* embedded implementation (*XCompilR.ParserGen.CocoR*) to provide the necessary assemblies. The implementation of an Visual Studio extension is not topic of this thesis and will be assumed. Every supported compiler generator must inherit from the interface *IParserGen* included in the *XCompilR.ParserGen.Library* library. To be able to load the parser generator at runtime via *Reflection* the abstract class *ABindingParserGen* is used, including the metadata information for each parser generator.

```
[Serializable]
public abstract class ABindingParserGen {
  public abstract string ParserName { get; }
  public abstract string AssemblyName { get; }
  public abstract IParserGen Parser { get; }
}
```

The following code shows the *IParserGen* interface implemented in the *XCompilR.ParserGen.CocoR* assembly:

```
public interface IParserGen {
  AParser CreateParser(string grammarFile);
  Assembly GenerateAssembly(string grammarFile);
}
```

Parser Generator (Coco/R)

The *Coco/R* source classes are provided by the University of Linz and are included within a sub-package (*JKU.Coco*) of the *XCompilR.ParserGen.CocoR* library. The *CocoParserGen* class implements the *CreateParser* method of the *IParserGen* interface and creates the parsers from the committed *ATG* files.
Occurring exceptions are handled via the *XCompilRExceptionHandler* attribute described in the subsection *Exception handling and logging* 5.6.4.
The parser generator also provides a method to load previously generated parsers from an *ATG* file. The following code snipped illustrates the instantiation of a parser generated from an *ATG* file.

```
[LogException]
[XCompilRExceptionHandler(typeof(XCompileException))]
public AParser CreateParser(string srcName) {
  try {
    /* Load available assembly. */
    string nsName = $"XCompilR.{srcName.Split('.')[0]}";
    Assembly assembly = LoadAssembly(srcName, nsName);

    /* Create and initialize Parser and Scanner. */
    Type type = assembly.GetType(nsName + ".Scanner");
    var s = (AScanner)Activator.CreateInstance(type);
    type = assembly.GetType(nsName + ".Parser");
```

```
      var p = (AParser)Activator.CreateInstance(type);
      p.InitParser(s);
      return p;
    } catch (Exception exception) {
      throw new XCompileException(
        "Could not generate Parser and Scanner from grammar file!",
      exception);
    }
  }
```

Generated ATG parsers

All generated parser assemblies include a *Parser* and a *Scanner* class, derived from the abstract classes *AParser* and *AScanner*. The following abstract methods define the *AParser* contract which has to be fulfilled to implement a valid language parser:

```
public abstract void InitParser(AScanner scanner);
public abstract void ReInitParser();
public abstract void Parse(string fileName);
```

Furthermore the implementation of the *AParser* class has two properties provided to build the *CompilationUnitSyntax* object required for *Roslyn* and to attach members to the dynamic binding object.

```
public dynamic BindingObject { get; set; }
public CompilationUnitSyntax CompilationUnitSyntax { get; set; }
```

The implemented scanners have to define the following abstract methods, which are used by the parsers:

```
public abstract AScanner Scan(string fileName);
public abstract AScanner Scan(Stream stream);
```

Creating ATG files

To generate a language parser it is required to write an *ATG* file. The written *ATG* files have access to the *CompilationUnitSyntax* and *BindingObject* property which are used to build the syntax tree and to bind members to the dynamic object. The sample below shows how to attach semantic elements to an *ATG* file to create a *Roslyn AST*.

```
...
Sample =        (. CompilationUnitSyntax =
                     SyntaxFactory.CompilationUnit();
                   string name; .)
"class"
Ident<out name> (. CompilationUnitSyntax =
                     CompilationUnitSyntax.AddMembers(
                       SyntaxFactory.ClassDeclaration(name); .)
"{"
...
```

The *Ident* non-terminal contains an out parameter initialized by the lexer (*Scanner*) during the recursive descent and offers the name of the class. The *CompilationUnitSyntax.AddMembers* method adds a new class declaration to the current compilation node. It is important to recall that *Roslyn* syntax trees are immutable objects and that all performed changes on an existing node create a new syntax tree object. This requires reassignment of existing instances during the modifications in the semantic elements.

5.6.4 Exception handling and logging

XCompilR uses an aspect-oriented way of handling exceptions and logging at runtime. To simplify the creation of aspects and to use logging attributes the *PostSharp* framework has been included. *PostSharp* supports multiple logging frameworks such as *NLog* or *log4net* and enables simple usage of *Log* or *LogException* attribute aspects. These can be declared on methods or classes and will be triggered depending on the preset configurations at the entrance and/or exit of the denoted methods or on exception occurrences.
The general *XCompilR* exception class is *XCompileException*. It can occur during language parser generation or language source file parsing.
A C# attribute declaration is required to handle the *XCompileException* exception. The defined attribute is declared as shown below:

```
[Serializable]
[AttributeUsage(AttributeTargets.Method)]
public sealed class XCompilRExceptionHandlerAttribute :
    OnExceptionAspect
```

OnExceptionAspect is a class defined in the *PostSharp* framework and offers methods which can intercept at different phases on an exception occurrence. These methods have the ability to steer the flow control of a method by offering meta-level information. The following snippet shows how the *OnException* method can set the flow control to return regularly although an exception has been thrown:

```
[Log]
public override void OnException(MethodExecutionArgs args) {
  if (_exceptionType == typeof(XCompileException)) {
    /* Set the method behavior after an exception occurred. */
    args.FlowBehavior = FlowBehavior.Return;

    /* Print the exception stacktrace to the console. */
    Console.WriteLine(args.Exception.StackTrace);

    /* TODO further exception handling. */
  }
}
```

All the methods marked with the *XCompilRExceptionHandler* attribute will be intercepted and the *OnException* method is called before the general ex-

ception flow behavior is triggered. Furthermore any call of the *OnException* method will be logged due to the *Log* attribute assigned.

5.7 Evaluation

The *XCompilR* framework offers simple ways to bind external sources to an existing C# project environment, which allows reuse of legacy code and interoperability. Although it seems very beneficial to embed new languages into the .NET framework by translating existing sources over *Roslyn* syntax trees and binding them to dynamic objects at runtime, there is still a negative tradeoff that has to be mentioned. These processes are performed at runtime, so they afford very high computational efforts during the parsing procedure. Depending on the imported sources the memory footprint can also be extremely high. For instance if a cross-compilation is performed with the Java programming language and the referenced sources use many Java framework related classes, then all these sources have to be loaded, scanned, parsed and bound to an object instance. This would mean a translation of hundreds of Java classes, which can result to very unpleasant runtime performances.
Furthermore, the current version of the *XCompilR* does not efficiently handle multiple occurring library references of translated languages, meaning that two or more unrelated *CLI* non-compliant libraries of the same programming language referencing to a shared resource, will still create multiple instances of the same shared resource under different namespaces. This can dramatically increase the requirements of the memory usage.
Another key point is that the translated libraries are only one-way available and operable. It is only possible to import language sources into a C# or *VB* environment to perform operations on behalf of them, but it is not possible to access from the internals of these sources .NET environment instances or the instances using them.
The framework also lacks the possibility to generate and manage the available assemblies for the language parsing. A managing instance — such as a Visual Studio extension — is required to offer loading and unloading of *ATG* files used for the language parsing and offering code completion for the imported sources at design-time.
But looking at the results from an conceptual point of view, the *XCompilR* fully fulfills the requirements. The framework fully supports all currently available programming languages after providing a *ATG* to building the required *Roslyn AST*. This enriches the C# and *VB* programming language for interoperability and enables a simplified migration process from a previously used programming language to the .NET environment.
Chapter 6 reviews and offers conclusions gained of this thesis.

Chapter 6

Conclusion

This thesis covered the importance of cross-language compilation based on the .NET framework. It mentioned the importance of nowadays language interoperability and covered the idea and internals of the *CLI* fulfilling this task. In addition it also gave an overview of terms related to the .NET framework and how their interactions are linked together. On top of these fundamentals the new Microsoft .NET Compiler Platform (*Roslyn*) was introduced and the key points were explained. This included the internal mechanics of the *Roslyn* framework, its *API*, the usage of syntax trees and important terms required to build AST*s*. Related to the terms of AST*s* the parser generator *Coco/R* was illustrated and how to create *ATG* files to generate lexers and parsers, which are used to compile multiple language sources.
Furthermore the chapter containing the *XCompilR* design, architecture and implementation was discussed. It included cross references to the previous defined topics and explained how to apply cross-language compilation using *Coco/R*, *Roslyn*, aspect-oriented concepts and dynamic objects for the instance bindings [Let12].
Although the implementation is only in prototype phase for the proof of concept, the created framework is still available under the *GNU General Public License* due to its great results. *XCompilR* offers a very easy and flexible way to integrate non-*CLI* compliant languages into the .NET framework without having to access lower-level implementation layers. It is possible to fork or contribute to the current version on *GitHub* (https://github.com/Xpitfire/CrossCompile).

6.1 Future Work

The realized framework prototype is fully functional and offers the possibility to import multiple language sources into the .NET framework. But there are still many questions unanswered and analysis that can be performed to

denoted this framework as complete.

6.1.1 Syntax tree building helpers

The generated language parsers offer multiple ways to bind the sources to a customer defined dynamic object instance via the defined *AParser* abstract class contract. Due to the immutable design of the *Roslyn* syntax trees it is still quite complicated to write *ATG* files mapping the source language to the dynamic language instances. To ease the work of the *ATG* developer, the *AParser* class must be extended and offer multiple helper methods and properties in order to select and exchange nodes, which can be used in the semantic actions of the *ATG* files.

6.1.2 Code completion for ATG files

When developing an *ATG* file, it is necessary to use the dynamic object (*BindingObject*) instance for the bindings and the *CompilationUnitSyntac* instance for the *AST* creation. When creating these files, no syntax highlighting, code completion or type checking is available. This can easily result to typos or misuse of declared syntax nodes. To create a *rich* development environment a customized editor is required, which provides the *Roslyn* API classes and verifies the assigned semantic actions.

6.1.3 Visual Studio Extension

To manage the languages available for the *XCompilR* and to trigger the parsing procedures, when loading a new *ATG* file, a Visual Studio extension would be required. This extension could also provide an editor for creating new *ATG* files including the necessary syntax highlighting, code completion for *Roslyn* and type checking for semantic actions.

6.1.4 Performance improvements

Because the current stage of the framework was foreseen as a prove of concept and marked as experimental, there are still many performance improvements and testing procedures available. For instance, a release version of the cross-language translation framework would have to handle multiple occurring shared resources. It should detect that there are already existing resource instances loaded and share this objects to newly imported resources to avoid unnecessary reparsing cycles. Furthermore it has to improve the syntax tree building and cache often occurring instances. All these key points would improve the overall performance at runtime.

6.2 Experiences

Writing this thesis was a very challenging task for me. It required a very sophisticated knowledge base about the internals of the .NET framework, multiple API*s*, framework related structures and design paradigms and the functionality of compilers, which also represented one of the most time consuming tasks. It was also interesting to find out that even huge companies such as Microsoft do not always document all their sources and sometimes lack design guidelines. Also when using frameworks such as *PostSharp* it is important to be aware of licensing and sometimes think twice before integrating them into newly developed projects. Problems can occur when developers require more from a framework than usually expected. This can become very expensive, when trying to remove wrongly chosen frameworks in an late project phase. The framework decisions established at the beginning of a project may influence an entire design and architecture. Afterwards the hardest part was to combine all of these topics into a unified functional set.
A few months ago I had no profound idea about the practical usage of syntax trees, parser generators or the core mechanics of the .NET framework, but finally I managed to gain all the required information and was able to apply them to this thesis. Great thanks are also directed to my professors which guided me into the right direction and offered me many code samples, papers and sources for researching. So I could *crack the code* and was able to create the *XCompilR* project. Of course, I have to admit, that I still have many areas of deficits and that I am continuously trying to improve my skills, but at this point I want to emphasize that it was a great pleasure to learn about the declared topics and to create this work. In addition to the technical terms I was also able to improve my time management and organizational skills during the development and the writing phase.

References

Literature

[Alf88] Jeffrey D. Ullman Alfred V. Aho Ravi Sethi. *Compilerbau Teil 1*. Deutschland: Addison-Wesley, 1988 (cit. on p. 20).

[Ecm12] Ecma. *Common Language Infrastructure (CLI) – Partitions I to VI*. Tech. rep. Geneva: Ecma International, June 2012. URL: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-335.pdf (cit. on pp. 2, 7, 8).

[Geo01] William T. Councill George T. Heineman. *Component-Based Software Engineering – Putting the Pieces Together*. United States of America: Addison-Wesley, 2001 (cit. on p. 24).

[Han03] Markus Löberbauer Hanspeter Mössenböck Albrecht Wöß. "Der Compilergenerator Coco/R". In: *Peter Rechenberg – Festschrift zum 70. Geburtstag*. Linz: Universitätsverlag Rudolf Trauner, 2003. Chap. 13, pp. 47–59 (cit. on p. 20).

[ISO15] ISO. *Information technology — Web Services Interoperability*. ISO Online Browsing Platform (OBP). 2015. URL: https://www.iso.org/obp/ui/#iso:std:iso-iec:29361:ed-1:v1:en (cit. on p. 2).

[Iva04] Pan-Wei Ng Ivar Jacobson. *Aspect-Oriented Software Development with Use Cases*. Boston: Addison-Wesley, 2004 (cit. on p. 26).

[JKU15] JKU. *Coco/R Tutorial*. Johannes Kepler University of Linz Website. 2015. URL: http://www.ssw.uni-linz.ac.at/Coco/ (cit. on p. 20).

[JNB15] JNBridge. *JNBridge*. JNBridge Website. 2015. URL: http://jnbridge.com/software (cit. on p. 2).

[Küh13] Andreas Kühnel. *Visual C# 2012 – Das umfassende Handbuch*. 6th ed. Bonn, DE: Galileo Press, 2013 (cit. on p. 5).

[Let12] Julian Lettner. "A Code Generation Pipeline for .NET". Masterarbeit. Hagenberg, Austria: University of Applied Sciences Upper Austria, Software Engineering, Sept. 2012 (cit. on p. 36).

[Lin14] Serge Lindin. *.NET IL Assembler*. Apress, 2014 (cit. on p. 9).

[Mic15] Microsoft. *.NET Compiler Platform ("Roslyn")*. Microsoft Open Technologies Website. 2015. URL: https://github.com/dotnet/roslyn/wiki (cit. on pp. 2, 13, 14).

[Ora15] Oracle. *Oracle Documentation*. Oracle Website. 2015. URL: http://docs.oracle.com/javase/7/docs/technotes/guides (cit. on pp. 1, 2).

[Prü13] Thomas Prückl. "Ein echter Pseudocode Compiler für .NET". Masterarbeit. Hagenberg, Austria: University of Applied Sciences Upper Austria, Software Engineering, June 2013 (cit. on p. 19).

[Sch11] Raphael Schober. "Supporting Modern Programming Paradigms via Code Transformations in the Common Language Infrastructure". Diplomarbeit. Hagenberg, Austria: University of Applied Sciences Upper Austria, Software Engineering, June 2011 (cit. on p. 6).

[Set96] Ravi Sethi. *Programming Languages – Concepts & Constructs*. 2nd ed. Murray Hill, New Jersey: Addison-Wesley, 1996 (cit. on p. 20).

[Wik15] Wikipedia. *.NET Framework Stack Overview*. Wikipedia Website. 2015. URL: https://upload.wikimedia.org/wikipedia/commons/thumb/d/d3/DotNet.svg/2000px-DotNet.svg.png (cit. on p. 6).

Acronyms

API	Application Program Interface.
AST	*Abstract Syntax Tree.*
ATG	*Attributed Grammar.*
CIL	Common *Intermediate Language.*
CLI	*Common Language Infrastructure.*
CLR	*Common Language Runtime.*
CLS	*Common Language Specification.*
COM	*Component Object Model.*
CTS	*Common Type System.*
DOM	Document Object Model.
EBNF	Extended Backus–Naur Form.
GC	*Garbage Collection.*
HTML	Hyper Text Markup Language.
IDE	Integrated Development Environment.
IL	*Intermediate Language.*
ISO	International Organization for Standardization.
JNI	*Java Native Interface.*
JVM	*Java Virtual Machine.*
VB	Visual Basic.
VES	*Virtual Execution System.*

Glossary

Abstract Syntax Tree A tree representation of the syntactic source code structure.

Attributed Grammar Formal grammar description of a specified programming language with attached attributes, which are evaluated within the *AST* nodes by a parser or compiler.

Coco/R Compiler Generator developed by the Johannes Kepler University in Linz (Austria).

Common Intermediate Language Microsoft's lowest-level human-readable programming language based on the *CLI*.

Common Language Infrastructure An open specification developed by Microsoft and standardized by the *ISO*.

Common Language Runtime Microsoft's virtual machine component for the .NET framework.

Common Language Specification Document describing how computer programs can be turned into *IL* code.

Common Type System A standard for type and type value memory representation within the .NET framework.

CompilationUnit Roslyn AST root class, representing a assembly solution.

Component Object Model Component Object Model Technology a Microsoft developed API for building and re-using software components, Windows services used by Microsoft Office products, Direct Show, etc..

EcmaScript Alternative name for the JavaScript programming language.

Garbage Collection Automatic handling for memory management, which can disposed unused objects from the heap.

GitHub Public git repository to commit, fork and contribute (open-source) software solutions.

IntelliSense Visual Studio code completion technology.

Intermediate Language Microsoft's lowest-level human-readable programming language based on the *CLI*.

IronPython An open-source implementation of the Python programming language integrated in the .NET framework.

JNBridge Framework to bridge code usage between Java and .NET technologies.

Java Native Interface Java standard Programming interface for using native written libraries in the *JVM*.

Java Virtual Machine Java Runtime Environment platform for executing Java code.

LL(1) grammar LL(1) grammar is used by so called LL parser, which in this case has one token look-ahead for recognizing a committed programming language.

NuGet Visual Studio package manager.

PostSharp Pattern-aware extension framework for C# and VB.

Reflection Framework abilities to examine and modify code structures and behaviors at runtime by using metadata.

Roslyn Microsoft's Compiler Platform.

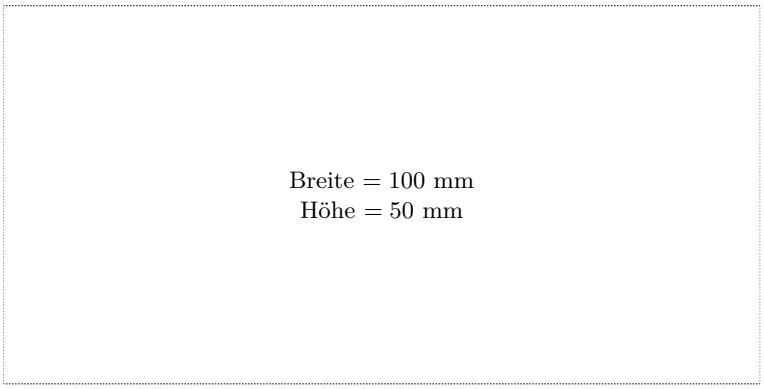
VSIX Visual Studio deployment extension unit.

Virtual Execution System Runtime system based on the *CLI*.

XCompilR Prototype for cross compiling languages based on an *ATG* for cross language interoperability, developed by Marius C. Dinu.

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —