DOI: 10.15514/ISPRAS-2022-34(3)-3

Diff tool for comparing .NET assemblies in the Rider IDE

V.I. Miroshnikov, ORCID: 0000-0002-6218-9406 <vladislaw.miroshnikov@gmail.com>
St. Petersburg State University,
7/9, University Embankment, Saint Petersburg, 199034, Russia

Abstract. A .NET developer occasionally needs to compare compiled programs or assemblies, e.g., when updating versions of third-party libraries or when working with their own binary files. However, the existing tools have some significant drawbacks, for example they don't support comparison of .NET Core assemblies. In this paper we reviewed different types of .NET assemblies and, taking into account their structure, developed and integrated into Rider IDE our own Assembly Diff tool which considers the disadvantages of the existing tools and expands the comparison possibilities. We presented several variants of comparison tool presentation and implementation and chose the most functional one in the form of a comparison tree, for which we developed and described special algorithms allowing to take into account semantic features of .NET types.

Keywords: Assembly Difference; assembly diff tool; .NET assembly diff; Rider; assembly comparison; Exe/Dll diff; comparing compiled assemblies

For citation: Miroshnikov V.I. Diff tool for comparing .NET assemblies in the Rider IDE. Trudy ISP RAN/Proc. ISP RAS, vol. 34, issue 3, 2022, pp. 31-46. DOI: 10.15514/ISPRAS–2022–34(3)–3

Инструмент для сравнения .NET сборок в интегрированной среде разработки Rider

В.И. Мирошников, ORCID: 0000-0002-6218-9406 <vladislaw.miroshnikov@gmail.com>
Санкт-Петербургский государственный университет,
199034, Россия, г. Санкт-Петербург, пр. Университетский, 7/9

Аннотация. Разработчику .NET иногда требуется сравнить скомпилированные программы или сборки, например, при обновлении версий сторонних библиотек или при работе с собственными бинарными файлами. Однако существующие инструменты имеют ряд серьёзных недостатков, например, они не поддерживают сравнение сборок .NET Core. В данной работе мы рассмотрели различные типы сборок .NET и, учитывая их структуру, разработали и интегрировали в Rider IDE собственный инструмент Assembly Diff, который учитывает недостатки существующих инструментов и расширяет возможности сравнения. Мы представили несколько вариантов представления и реализации инструмента сравнения и выбрали наиболее функциональный в виде дерева сравнения, для которого разработали и описали специальные алгоритмы, позволяющие учитывать семантические особенности типов .NET.

Ключевые слова: Assembly Difference; assembly diff tool; .NET assembly diff; Rider; сравнение сборок; Exe/Dll diff; сравнение откомпилированных сборок

Для цитирования: Мирошников В.И. Инструмент для сравнения .NET сборок в интегрированной среде разработки Rider. Труды ИСП РАН, том 34, вып. 3, 2022 г., стр. 31-46. DOI: 10.15514/ISPRAS–2022–34(3)–3

1. Introduction

The .NET [1] platform – is Microsoft's software platform for developing various applications. One of the distinguishing features of this platform is the ability to develop applications using several programming languages, e.g., C#, F#, Visual Basic .NET, thanks to Common Language Runtime

(CLR). Currently, there are two different platforms from Microsoft: *.NET Framework* and .NET Core/.NET. .NET Framework is an older version and supports only the Windows operating system, which differs it from .NET Core/.NET that is already cross-platform and corresponds to modern industry standards. Thus, according to Stack Overflow research [2], for 2021 .NET Core/.NET and .NET Framework are among the top three most popular frameworks.

One of the basic, structural and functional units of the .NET platform is assembly, which is used for version control, application deployment and configuration. **.NET assembly** – is a collection of .NET types (classes, interfaces, etc.) and resources (JSON, XML files, etc.) assembled to work together to form a logically functional unit. During the .NET development process, it is often necessary to compare the versions of the compiled programs. For example, sometimes this need arises when updating third-party libraries or debugging problems with dependency resolution of .NET applications, and sometimes – it can be our own binary files, for which we want to understand which version of code corresponds to any of the previously published binary files.

However, there are various ways to compare .NET assemblies. For example, a developer can use any decompiler to get the source code and then apply a text comparison tool. Nevertheless, such solutions are not very convenient because they don't allow to compare assemblies directly without additional tools. There are also “complete applications” providing comparison of assemblies, but they have own significant drawbacks, for example, lack of support for comparing .NET Core/.NET assemblies.

One of the existing development environments for the .NET platform is **Rider** [3] – JetBrains cross-platform development environment, which is one of the “most loved” [4] IDEs according to the mentioned above Stack Overflow research of 2021. In addition to the existing functionality in Rider, this paper proposes extending the capabilities to work with the .NET platform, namely – adding a tool for comparing compiled .NET assemblies. This should take into account the weaknesses of the existing solutions and improve the comparison capabilities. In this paper we consider various ways of comparing assemblies and also offer the implementation of one of them in the Rider IDE. We also need to consider the technical challenges of the task, which is not only about searching for source code differences provided in the C# language. For example, due to the large number of structural units and language features in .NET assemblies it is necessary to search and display “semantic” differences, both in the metainformation of the assembly and in the resources and code, which will be discussed in more detail in this paper.

This paper is organized as follows. Section 2 includes the different types of .NET assemblies, provides the structure and contents of the assembly, gives an overview of existing solutions and the Rider architecture required for the implementation. Section 3 provides the first version of the implementation based on a directory comparison, and discusses the alternative implementation as a comparison tree. Section 4 describes the final implementation based on the assembly diff tree, and gives various algorithms for constructing the tree and comparing types. Section 5 gives general conclusion of the paper and further plans.

2. Background

Assemblies can be created from one or more source code files. For simplicity, they can be thought of as archives, which have a specific structure and in particular contain source code data. Typically, a compiled assembly is presented on disk as a file with the extension .exe or .dll.

2.1 Types of .NET assemblies

According to Microsoft's official .NET [5] documentation, assemblies are divided into two types:

- Executable assemblies – assemblies that are represented as an executable file with the .exe extension. For the sake of brevity, this type of assembly is called an “Exe-assembly”;
- Assemblies that are presented as a dynamic link library file – files with the extension .dll.

Hereafter, for the sake of brevity, this type of assembly is called a “Dll-assembly”.

There is also a division of assemblies into single-file and multi-file assemblies. A single-file assembly is the simplest type of assembly, with all its contents placed inside a single *.exe or *.dll file. Multi-file assembly, on the other hand, consists of a set of .NET modules, which are deployed as a single logical unit and are provided with single version number. Such assemblies can be created, for example, using command line compilers. One of the main advantages of this type of assemblies is that they allow combining modules written in different .NET compatible programming languages. A multi-file assembly can also be represented as a file with the extension *.exe or *.dll.

.NET assemblies also satisfy with the ECMA-335 standard [6]. This standard is a specification of common language infrastructure and .NET platform, defining architecture of .NET runtime system, arrangement of different libraries, structure, and types of assemblies, description of intermediate language CIL and others.

Let's take a closer look at Exe and Dll assemblies, as well as their various subtypes.

a) Exe-assembly

According to ECMA-335 standard [6], the distinguishing features of Exe-assemblies are follows:

- Possibility of “direct” invocation – the user can directly run a .NET application with the .exe extension;
- Availability of its own address space and memory area – since the Exe-assembly is executable, it can be run as a separate operating system process with its own address space and memory area.

As described in Section 1, there are two main .NET platforms: .NET Framework and .NET Core/.NET. Each of these platforms provides a different implementation of Exe-assemblies:

1) .NET Framework

In this platform there is a single and so called “classic” type of Exe-assembly. As a result of compilation, we get one file with the extension .exe, which remains to be run by the user. The limitation in the form of dependencies on other assemblies should be taken into account.

2) .NET Core/.NET

- Native host assembly – compilation results in both a file with extension .exe and a dynamic library file with extension .dll. There is a restriction though: an Exe-application cannot be started without a corresponding dynamic library file, because the DLL containing the application source code is loaded during the startup.
- Single File assembly (or so-called standalone assembly) – an assembly presented as a single file, allows to combine all files that application depends on, resources, other assemblies into a single assembly. This type of assembly makes it much easier to deploy and distribute the application, but the size of such a file will be large, as it will include the runtime and platform libraries.

b) Dll-assembly

In accordance with the ECMA-335 standard mentioned above [6], the distinguishing feature of this type of assembly is that it cannot be “directly” launched, so the DLL-assembly has no address space or memory area of its own. The assembly is dynamic in the sense that it can only be loaded or mounted in some other process, such as a console or web application. A Dll-assembly is also called a class library, because it actually contains the source code, but has no “entry point” of its own.

It's worth noting that there is only one type of DLL-assembly in .NET Framework and

in .NET Core/.NET.

To summarize the overview of the different types of .NET assemblies, we also declare that a comparison of all the types described above should be supported in the diff tool.

2.2 Structure and content of the .NET assembly

According to the official documentation from Microsoft [7], any .NET assembly, both Exe and Dll, consists of the following:

- Assembly manifest – a key component of the assembly, without which the source code cannot be run. It includes name, version, a list of all assembly files, a list of references to other assemblies, and a list of references to types used by the assembly. Manifest allows the system to identify all the files included in an assembly, map references to types, resources, and assemblies to their files, and manage version control.
- Type metadata – used to define the location of types in an application file;
- Application code in the intermediate CIL language. The assembly file does not contain source code in C# or any other .NET compatible language, but instead contains IL code (a.k.a. bytecode) – the language of the .NET virtual machine.
- Assembly Resources. These can be various JSON, XML files as well as images and audio files.

Additionally, it is worth noting that there is such a thing as format of .NET assembly [8] – binary file format.

The format is fully defined and standardized in the ECMA-335 specification [6]. All .NET compilers and runtime environments use this format. For example, it defines that the assembly is processor- and operating system-independent.

This format makes the assembly diff tool platform-independent in the sense that there is no need for any checks on which processor or operating system a particular assembly was derived. Thanks to the unified assembly format, it is possible to compare any assemblies, even if one of them was derived, for example, from Windows and the other from macOS.

2.3 Existing solutions

The existing solutions can be divided into two groups:

- Solutions that allow to compare assemblies using several tools. This type of solutions are based on using some .NET decompiler (Ildasm, IlSpy, dotPeek) to get the C# source code and then applying any diff tool for text comparison.
- Complete solutions that allow to compare assemblies directly without using any third-party tools. These full-fledged diff tools provide both decompilation and source code difference display. However, many of them have some significant disadvantages. Some, for example, do not support comparison of .NET Core/.NET assemblies, which makes such tools not very relevant these days. Others do not support viewing decompiled C# code and only allow user to see added/removed types.

In our assembly diff tool, we take many of these disadvantages into account and offer solutions to fix them.

2.4 Rider architecture

As stated earlier, Rider [3] – a cross-platform development environment for the .NET platform from JetBrains. To understand the inner workings and architecture of the Rider environment, we refer to the corresponding article [9] published in the journal CODE Magazine.

Thus, the Rider consists of two main operating system processes running in parallel:

- Frontend process – responsible for rendering the user interface, based on the IntelliJ platform [10] and runs on a Java Virtual Machine.

- Backend process – a process without user interface, responsible for code analysis, code inspections, formatting and other logic. Based on the ReSharper [11], which is an extension to the Visual Studio development environment from JetBrains and runs on a .NET virtual machine.

One of the key points of the Rider IDE is that the two processes "communicate". For this purpose, a specially developed "Reactive Distributed" protocol [12], presented as an open-source library, is used. This protocol ensures the principle of reactivity: it provides entities that can react to changes through an event subscription mechanism. It also provides consistency in the state of the system at any given time, and importantly, it allows to work with distributed systems. This protocol enables the exchange of information between two processes.

The Rider architecture is shown schematically in Fig. 1.

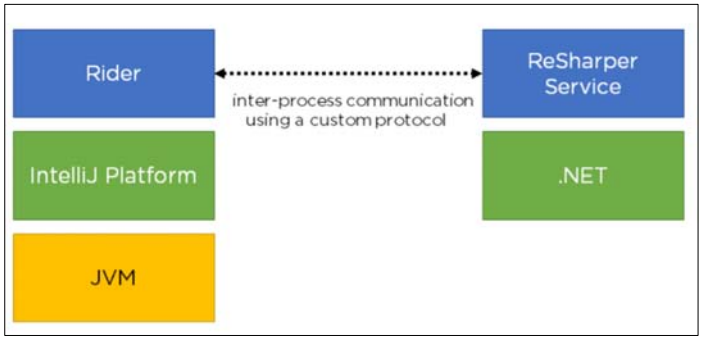

Fig. 1. Architecture of the Rider (from the related article [9])

3. Assembly diff methodology

3.1 Project decomposition

Assembly diff is decomposed into three key subtasks:

- "Disassemble" the assembly and find objects that are different and contain source code. This subtask is due to the fact that an assembly can consist of several source files and has a certain structure described in Section 2.2. Therefore, among various metadata tables, assembly resources and other contents it is necessary to find source code objects.
- Decompile found objects to C# code, as they will be represented in intermediate IL code in the assembly.
- Calculate and display the difference represented as C# code.

Thus, taking the Rider architecture into account, the following decision has been made:

- "Reading" the assembly and finding source code objects, as well as the decompilation process into C# code should be performed on the backend process and implemented in C# using the ReSharper capabilities.
- Difference calculation and display should be done on the frontend process and implemented in Kotlin/Java using the capabilities of the IntelliJ platform.
- The exchange of information between the two processes must be provided by the protocol described in Section 2.4.

3.2 Implementation based on directory comparison of the IntelliJ platform

The IntelliJ platform already includes the ability to compare arbitrary directories and archives. In the IntelliJ IDEA IDE, it is also possible to compare JAR files, which are a kind of analogue of .NET

assembly. JAR files are archives that contain source code, manifest files, and various resources such as audio files. The JAR file diff tool shows the difference between the decompiled versions of Java classes within them.

Based on the existing capabilities of the IntelliJ platform, we decided to make the first version of the .NET assembly diff tool, presented in Fig. 2. There is a table for the user that contains the .NET types – classes, interfaces, enumerations, etc. When you click on a row, you can see the difference of the decompiled C# code.

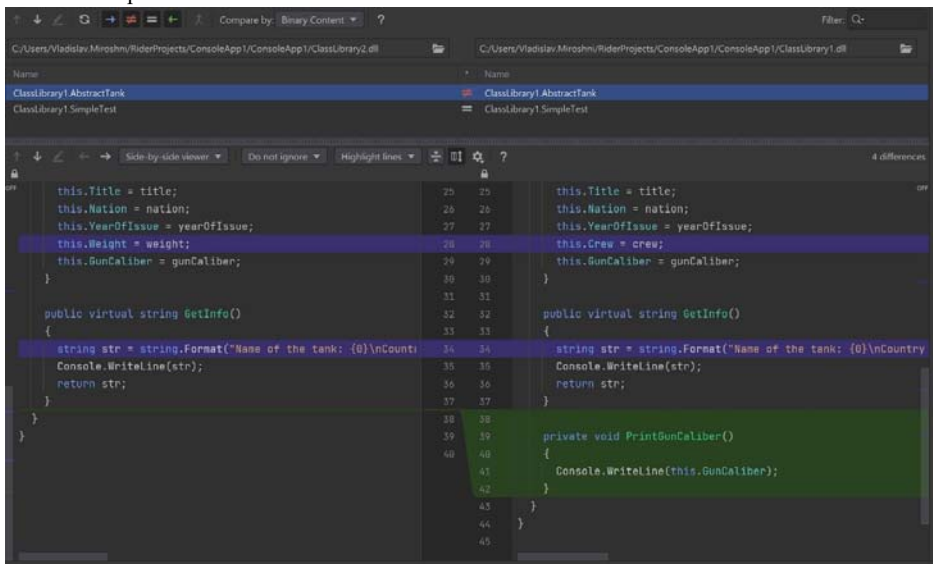

Fig. 2. Assembly diff tool based on directory comparison

However, the implementation of the diff tool is not limited to the one method presented above. We suppose it's appropriate to consider an alternative option. To do this, Assembly Explorer Rider subsystem should be reviewed.

3.3 Assembly Explorer subsystem

Assembly Explorer – an existing subsystem of the Rider IDE, which provides a wide range of opportunities to explore the contents of assemblies. An assembly is represented as a tree with assembly name, version, and platform in the root node, e.g., .NET Framework v.4.8 or .NET Core v5.0.

In the tree itself, the user can see the following information when the root node is expanded:

- Metadata – this subtree contains all metadata information, various metadata tables, special type tokens and more;
- References – this subtree contains information about the dependencies of the assembly, i.e., which other assemblies, packages, and modules the current assembly refers to;
- Resources – this subtree contains various resources, e.g., audio files, images, XML documents. By double-clicking on a resource node, you can view its contents.
- Namespaces and nested types – this subtree is responsible for the source code. You can view classes, its internal parts, such as methods, fields, constructors and other. Double-clicking on a node decompiles the source code into C#, opens in a new window and navigates through the document.

An example of an assembly tree is shown in Fig. 3.

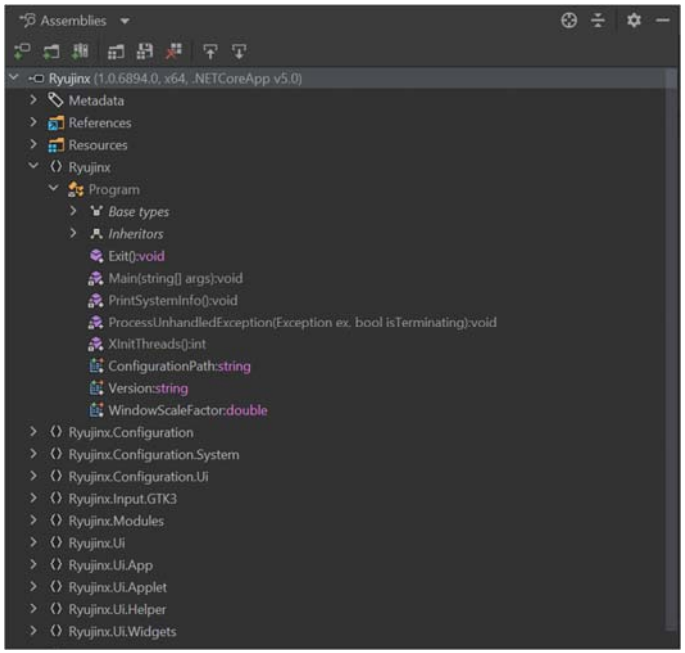

Fig. 3. Assembly tree in Assembly Explorer

It is also worth noting that this subsystem is implemented in other products of the JetBrains .NET ecosystem – ReSharper and dotPeek.

Considering the capabilities of the Assembly Explorer subsystem, we made the following key decision – to use a tree view in the diff tool implementation for the following reasons:

- The assembly diff tree view greatly extends the functionality of the tool in contrast to the version based on directory comparison. Such a tree view allows to compare not only source code differences, but also resources and references from other assemblies.
- Comparison of assemblies in the tree view will also allow to extend the scope of application of this tool. Integration with Assembly Explorer subsystem in future will allow to add assembly comparison functionality in other products – ReSharper and dotPeek. For this reason, choosing a method based on directory comparison will significantly reduce the scope of this functionality and possible work scenarios, as it will only allow comparing assemblies in Rider.

4. Implementation based on assembly diff tree

4.1 General operating principle

By choosing to represent the difference of the assemblies as a tree, we have implemented this approach.

Assembly diff tool works as follows:

1) When a user requests to compare two assemblies on the frontend, the frontend assembly tree is initialized, as well as various additional components. At the same time, an asynchronous request is sent to the backend to “read” assemblies and the “internal backend” tree is initialized, which

is created separately for each request. When parsing assemblies, among the various meta-information and other contents, objects are searched and divided into 3 subtrees: a subtree of references, resources and assembly types. This involves matching appropriate pairs to merge into a single tree node (e.g., the same method with a changed implementation from one assembly version to another), and also applies various algorithms, such as semantic .NET type comparison, which will be described in Section 4.3. The data found is sent to the frontend process using the protocol described in section 2.4 and displayed as a final assembly diff tree. It is worth noting that it is thanks to the reactive protocol that data consistency is ensured at any time between the frontend and backend trees.

2) By double-clicking on the tree node responsible for the .NET type (class, field, method, etc.), a request is sent to the backend, the desired backend tree node is found and then the IL code is translated into C# using the decompiler and the decompiled code is forwarded back to the frontend. Navigation is also provided: the frontend also receives the required parameter to which the carriage should be moved in the corresponding window.
3) The frontend calculates the C# code difference using Myers and LCS [13, 14] algorithms, shifts the carriage, and displays it to the user with the C# syntax highlighting.
4) A specially created protocol model in Kotlin DSL is used to “communicate” between processes. This model describes containers/protocol types that contain some data (such as decompiled class code or assembly paths). These protocol types are then serialized and deserialized when the two processes “communicate”.

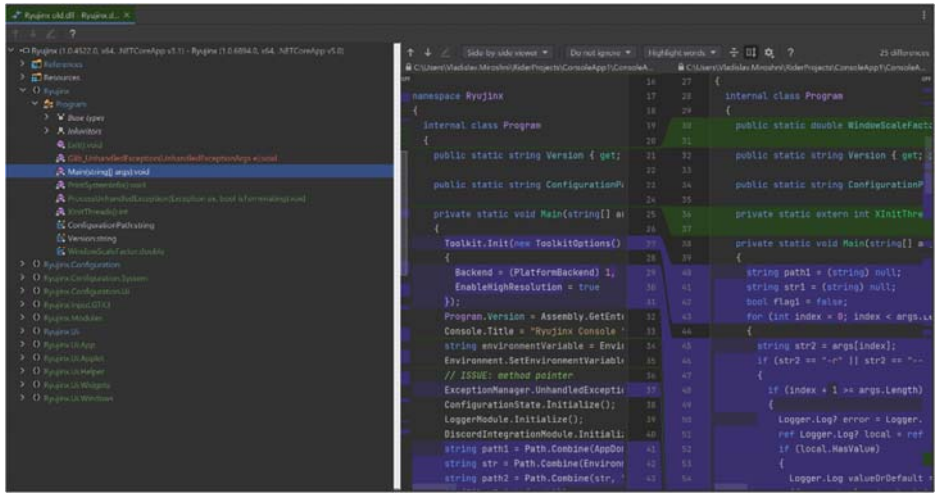

Fig. 4. Assembly diff tool work example

Additionally, it's worth mentioning that this diff tool supports comparison of both .NET Framework and .NET Core/.NET assemblies, including Exe and Dll assemblies and their subsets described in Section 2.1. This fact distinguishes this tool from existing solutions.

An example of the assembly diff tool is shown in the Fig. 4.

Consider the structure and principle of operation of individual subtrees.

4.2 Algorithm of assembly diff tree construction

The assembly diff tree is a key component in the diff tool, it allows to view difference in resources, references and .NET types themselves, as well as C# code differences.

On the backend, a separate *AssemblyDiffTreeHost* instance is created when an appropriate request is received. This instance is responsible for building and updating the tree.

The general algorithm for constructing a tree is as follows:

1) Once the pair of assembly paths is obtained, the assemblies are loaded and a root tree node is created, containing information about the two assemblies, differences in name, versions, platforms, and architecture on which the assembly was obtained.
2) After the assemblies are loaded among the various meta-information and other content, so-called “entry points” are searched and a level 1 of the tree is created from the root node of level 0. Level 1 of the tree contains a references node, a resource node and nodes representing containers for .NET types – in this case namespace nodes. The tree is thus divided into 3 subtrees, which are processed and built independently in asynchronous mode.
3) The following is true for each level of the tree: globally there are three groups of entities: nodes, providers and presenters. Each level of the tree is built as follows: for each node the corresponding provider is triggered, which builds a subtree of the next level, calculating children and creating the necessary instances of nodes. It is possible to say that provider of n-1 level collects nodes of n level of the tree. For example, when you visit node *ClassDiffNode* the corresponding provider will be called, which will spawn children, i.e., the next level containing types of the given class: methods, fields, events, nested classes and so on. Thus, an assembly diff tree is built by applying a breadth-first search (BFS) to traverse the tree in order of levels. At each level of the tree it is necessary to search and match the entity of the old assembly with the new one and merge it into one tree node (in case there is no pairing, the entity is considered as deleted or added). For each of the resource, references and types subtrees, its own matching algorithms are applied.
4) Presenters are used to compose the presentable item of the node: generating text, setting styles and colours, and selecting a suitable icon. The node data is then wrapped in a special protocol container and sent to the frontend to be unpacked and displayed to the user.

Consider the operation of the individual subtrees:

a) References subtree (Fig. 5)

This subtree contains nodes of two types: *AssemblyReference* (a dependency on another assembly, e.g., *System.Runtime*) and *ModuleReference* (these can be references to some third-party library). All references of old and new assemblies are merged and further grouped by name without regard to version or additional tokens. In this way, each found old-new reference pair will be merged into one node and for it the version difference will be calculated to show to the user. If there is no pair for any reference, it is considered as added or deleted.

b) Resources subtree (Fig. 6)

This subtree contains various resource nodes: images, XML, JSON files, etc. Similar to the references subtree, the resources are merged and grouped by name, then pairs are searched and merged. In this way the user will be able to identify which resource files have been added or removed from one version to another.

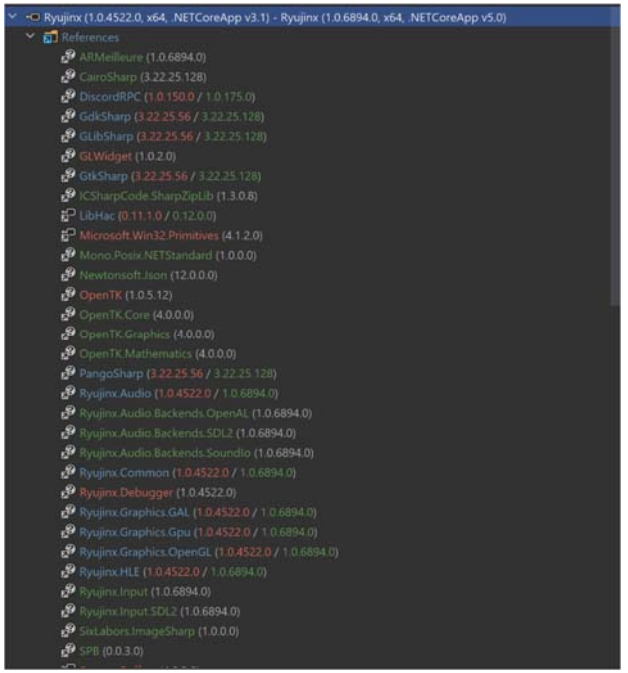

Fig. 5. Example of Reference subtree

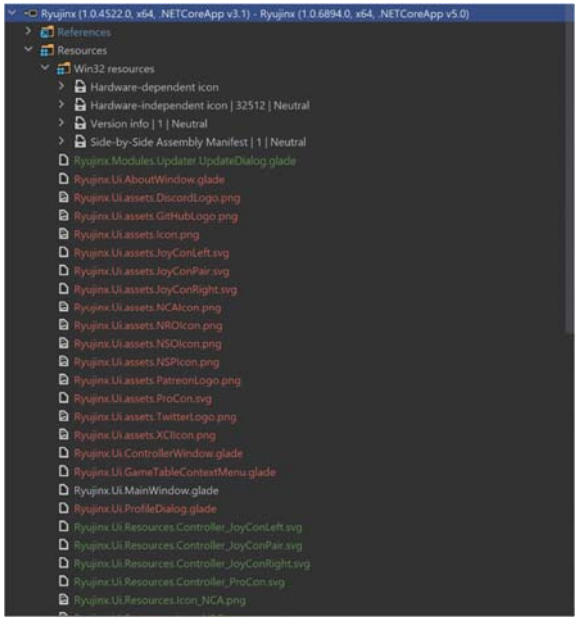

Fig. 6. Example of Resources subtree

c) Assembly types subtree (Fig. 7)

This subtree contains namespaces and nested types: classes, interfaces, enums, etc. with their own nested types.

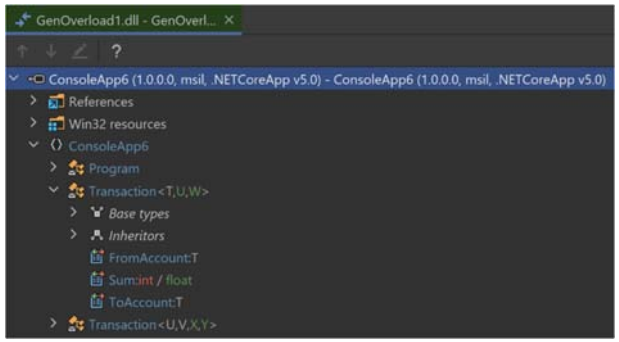

Fig. 7. Example of Assembly types subtree (including signature diff)

When constructing this subtree, the logic and general approach changes somewhat. The general algorithm is as follows:

1) The first stage is to search for namespaces from both assemblies and group them by the unique CLR *QualifiedName*, noting that no two namespaces with the same FQN can be in the same assembly. Thus, we will have two groups: the first will represent namespaces with no pairing, i.e., deleted or added namespaces. The second group will have old-new namespace pairs, meaning that we have found the right pair to merge into a single node in the tree. To determine the state of such a node (modified/unmodified) our Quick Check method first compares the number of nested types for a namespace pair. For example, if the number of classes inside the old and new namespaces does not match, then the node is considered to be changed. If the Quick Check fails, a Full Check is performed: for both namespaces *CLRName* of nested types is collected and compared as sets. There is one extreme case, where the namespaces do not differ in their subtypes and the difference is only at deeper levels (e.g., the method body within the class has changed). In this case, the state of the namespace node will be dynamically updated with a special trigger.
2) At the second stage, for each namespace, subtrees are constructed independently of each other. This level will contain the "children" of namespaces – classes, interfaces, structures, enums and records. It will also group by type *CLRName* and have two groups. The first group will contain unpaired nodes, i.e., nodes meaning that the type was removed or added depending on its location (from the old assembly or from the new one). The second group will have pairs to merge into a single tree node. The modified/unmodified state of the node is defined as follows:
   a. First, we apply a Quick Check for the two types by comparing their signatures. For example, if a class was *public* in the old assembly and *private* in the new assembly, then that node is identified as the changed node.
   b. If the signatures match, a Full Check is performed – a rendering of the IL code is performed for the two types. For example, for two classes we will generate all their IL code and then compare it to determine the state of the node. We believe that in this case the IL code rendering is the key point in determining the node state. We could consider another approach and generate C# code for the classes, but this approach would be slower compared to IL code rendering, which is what we have chosen to achieve the fastest performance in tree construction.

3) When constructing the next levels, we work with "children" of classes, interfaces, etc., i.e. fields, methods, properties, nested classes, etc. The general logic here is partly the same as in point 2: for each type, e.g., class, we collect its old and new nested types from assemblies and also group them by *CLRName*. However, here we have 3 groups:
   a. The first group consists of types for which there are no other types with the same name, i.e., they are treated as deleted or added depending on their location.
   b. The second group consists of pairs, i.e., we have found pairs of types with the same name. In this case the logic changes: first we need to determine if both types belong to some single assembly. If they do, it means that the types don't need to be merged into one tree node, they are two different types and two different nodes. This is possible because of the overloading mechanism in the .NET platform: the types in a given tree level can have the same name. For example, the old assembly had two overloaded methods named *Start* in the *Fabric* class, but the new assembly has no such methods in the same class. In this case, the two methods are considered deleted and treated as different tree nodes. If, however, each of the two types belongs to a different assembly, we have a merge into one node and state detection is performed. A Quick Check – a signature comparison – is also performed first. If this check fails, a Full Check – IL code rendering – is performed. But there are peculiarities here, depending on the .NET type being checked. For example, if we have a property, then in addition to rendering the IL code we need to find and render the IL code of the corresponding *Getter* and *Setter*, because they are located separately from the property in the assembly. Similar behaviour for events, where we need to search for the corresponding *Adder* and *Remover*. A method deserves special attention: if a method is an iterator and *yield return* is used in it or if a method is marked with *async* keyword, a special *StateMachine* class is generated in a compiler-generated class in IL code. In such a case to understand whether the method has changed or not, besides rendering the IL code of the method itself, it is necessary to search among various metadata and *CustomAttributes* and then render this *StateMachine*. The situation is similar when using lambda expressions: in this case, separate compiler-generated methods are created during compilation, and sometimes even entire classes if there is a closure.
   c. The third group contains tuples of 3 or more types with the same name. Here will only be types that can be overloaded in .NET: methods and operators. This group is of the greatest interest, including research interest. Here we immediately face the problem of type matching: suppose we have three overloaded methods *Start* with different signatures in the old assembly, and the new assembly has three overloaded methods *Start* with different signatures in the same class. The question arises: how exactly do we map these methods for further merging into one node? The problem could be solved by using a special token given to each type at compile time. If it were unique and didn't change, we could accurately find the right pairs of methods by this token, but it changes when recompiling. In this case we have to choose some other approach. To do that, we developed a special algorithm for semantic matching and comparison of .NET types, which we will discuss next.

Also at each level, after the providers are completed, the view for the node is computed using the appropriate node presenter. Particularly noteworthy is the generation of text with colour definitions. Thus, for nodes at each level of this subtree (excluding the namespace level as an unnecessary case), a signature difference calculation algorithm is applied. This algorithm computes and identifies parameter addition/removal, parameter type or return value changes, and generic parameter changes. The signature difference will be displayed to the user in the tree and he will not need to look through the decompiled code additionally.

We consider this tree-constructing algorithm to be unique, as it provides wide functionality in exploring differences of assemblies. It is worth noting that existing solutions do not have this kind

of tree-constructing behaviour and use approaches that, in our opinion, are not the most accurate. For example, some existing solutions consider all types with different signatures as different. In this case, assume there is a method in the old assembly and assume that in the new assembly only one parameter has been added to the signature in that method, but the body of the method itself has not changed. Then there will be two different nodes in the tree: the method with the old signature will be considered completely removed, and with the new signature completely added, despite the fact that the implementation of the method has not changed at all. However, in this case we suppose it's more correct to show the user a single node in the tree with the state changed, showing the addition of a parameter to the signature.

4.3 Algorithm of semantic comparison of .NET types

Suppose the following situation. An old assembly has a class with two overloaded methods:

```
class OverloadListing
{
  public void Add(int id, int age)
  {
    // Method body
  }
  // Other methods ...
  public void Add(string name, string
      email)
  {
    // Method body
  }
}
```

The new assembly has the same class with two overloaded methods, but these methods have different signatures:

```
class OverloadListing
{
  public void Add(string name, string[]
      emails)
  {
    // Method body
  }
  // Other methods ...
  public void Add(int id, float age)
  {
    // Method body
  }
}
```

We don't consider methods with different signatures as different and we don't think it's correct to show two deleted methods and two added methods in the tree in this case. In the case of several overloaded methods whose signatures have changed in the new assembly, we believe it is most correct to match methods with the most similar signatures. Thus, for the old method it's necessary to find the closest new method, taking into account the semantic differences.

When implementing such an algorithm, it is most important to propose an appropriate metric to determine the proximity of the signatures. In this example, it is obvious that the signature from the old assembly *public void (int, int)* is closer to the signature of *public void (int, float)* than to *public void (string, string[])*, because in the first case the type of one parameter has changed and in the second two have changed. Our metric works as follows: for the chosen pair of .NET types we create a *similarityCounter* that accumulates data about the proximity of signatures. Then multiple checks are performed: comparison of *CLRName* for two types, generic parameters (if exists), comparison of access modifiers, keywords, return value types, also compared types of passed parameters, their number, name of parameters, presence of additional modifiers *ref/in/out*, performed

CLRTypeConversion – determination of data types affinity (for example, we determine that type *float* is closer to type *double* or *int* than to *string*) and other checks. Each check, if passed, has its own weight by which the *similarityCounter* is incremented. For example, for some keywords there are the following checks:

```
public int
    CompareByMostCommonParams(Element
    first, Element second)
{
    //...
    if (first.IsStatic ==
        second.IsStatic)
        similarityCounter++;

    if (first.IsVirtual ==
        second.IsVirtual)
        similarityCounter++;
    //other keywords checks (override,
        sealed, abstract, etc.)
}
```

Using this metric, we compare all signatures from the first assembly with all signatures from the second assembly and use a greedy algorithm: sort all possible pairs and take the closest pair of signatures, then the closest of the remaining ones, and so on. If we have 3 overloaded methods in the old assembly and 5 in the new one, we'll find only 3 mappings, and 2 methods will be considered as added. It is quite possible that there are pairs with the same proximity value (e.g., there were signatures (*int*) and (*double*) and became (*string*) and (*object*)), then we can choose the matching arbitrarily. Comparing method bodies and rendering IL code is unnecessary in this algorithm, thus reducing the overhead.

This algorithm is in some sense a heuristic – we can define other proximity metrics too. We do not consider our chosen metric to be the only true one and leave the ability for research in this area.

5. Conclusion and Future work

Assembly Diff tool is a tool designed and integrated into the Rider IDE for comparing compiled .NET assemblies and provides extensive functionality for exploring assembly differences. The tool includes not only differences in .NET types but also references and resource differences and presents them in a convenient diff tree. The Assembly Diff tool considers and fixes the shortcomings of existing tools, expands the number of supported .NET assembly types, and uses specially designed algorithms in the comparison tree to compute signature differences and account for semantic features in type mapping. The Assembly Diff tool is used in Rider with a special assembly diff action (including the Ctrl+D shortcut) and is also integrated with the version control subsystem, allowing users to view assembly diffs in the Commit tab.

However, in the future we plan to integrate the Assembly Diff tool into other products: ReSharper and dotPeek, and to integrate with the NuGet Rider subsystem. This would allow users to conveniently explore the differences between the versions of the downloaded third-party libraries.

References

[1]. .NET platform from Microsoft. URL: https://dotnet.microsoft.com/, accessed 19-March-2022.
[2]. Stack Overflow frameworks and platforms survey for 2021 year. URL: https://insights.stackoverflow.com/survey/2021#most-popular-technologies-misc-tech, accessed 19-March-2022.
[3]. JetBrains Rider IDE. URL: https://www.jetbrains.com/ru-ru/rider/, accessed 19-March-2022.
[4]. Stack Overflow most loved collaboration tools survey for 2021 year. URL: https://insights.stackoverflow.com/survey/2021#most-loved-dreaded-and-wanted-new-collab-tools-love-dread, accessed 19-March-2022.

[5]. .NET assemblies' types according to Microsoft documentation. URL: https://docs.microsoft.com/en-us/dotnet/standard/assembly/, accessed 20-March-2022.
[6]. Standard ECMA-335: Common Language Infrastructure (CLI), 6th edition, June 2012. URL: https://www.ecma-international.org/wp-content/uploads/ECMA-335_6th_edition_june_2012.pdf, accessed 20-March-2022.
[7]. .NET Assembly contents according to Microsoft documentation. URL: https://docs.microsoft.com/en-us/dotnet/standard/assembly/contents, accessed 21-March-2022.
[8]. .NET Assembly format according to Microsoft documentation. URL: https://docs.microsoft.com/en-us/dotnet/standard/assembly/file-format, accessed 24-March-2022.
[9]. C. Woodruff and M. Balliauw. Building a .NET IDE with JetBrains Rider. CODE Magazine, 2018, November/December.
[10]. JetBrains IntelliJ Platform. URL: https://www.jetbrains.com/ru-ru/opensource/idea/, accessed 23-March-2022.
[11]. JetBrains ReSharper. URL: https://www.jetbrains.com/ru-ru/resharper/, accessed 23-March-2022.
[12]. Reactive Distributed communication framework. URL: https://github.com/JetBrains/rd, accessed 23-March-2022]
[13]. E.W. Myers. An o(nd) difference algorithm and its variations. Algorithmica, vol. 1, 1986, pp. 251-266.
[14]. J.W. Hunt and M.D. Mcilroy. An algorithm for differential file comparison. Computing Science Technical Report, vol. 41, Bell Laboratories, 1976, 9 p.

Information about the author / Информация об авторе

Vladislav Igorevich MIROSHNIKOV – a student and a researcher at St. Petersburg State University. His research and professional interests include IDE development, code refactoring, power consumption research of Android devices.

Владислав Игоревич МИРОШНИКОВ – студент и исследователь Санкт-Петербургского государственного университета. В сферу научных и профессиональных интересов входит разработка IDE, код-рефакторинг, исследование энергопотребления Android устройств.