2016 IEEE/ACM 13th Working Conference on Mining Software Repositories

A Dataset of Simplified Syntax Trees for C#

Sebastian Proksch, Sven Amann, Sarah Nadi, Mira Mezini
Software Technology Group
Technische Universität Darmstadt, Germany
{proksch, amann, nadi, mezini}@st.informatik.tu-darmstadt.de

ABSTRACT

In this paper, we present a curated collection of 2833 C# solutions taken from Github. We encode the data in a new intermediate representation (IR) that facilitates further analysis by restricting the complexity of the syntax tree and by avoiding implicit information. The dataset is intended as a standardized input for research on recommendation systems for software engineering, but is also useful in many other areas that analyze source code.

1. INTRODUCTION

A multitude of source-code-based recommendation systems for software engineering (RSSE) exist. Examples include systems for code completion [5], code search [6], or snippet mining [1]. A recurring challenge for every new RSSE is to find suitable projects that can be used as input. The recent rise of platforms like Github, Bitbucket, or SourceForge make it easy to find vast amounts of source code that can be used for the above research. However, open-source repositories cannot be directly used in analyses, because some effort is required in order to prepare them, e.g., resolving dependencies and making them compile. Additionally, it is necessary to think of how the results of a batch analysis of many repositories are stored such that it is easy to access them later in a structured way. In short, using and analyzing a large number of open-source projects is usually non-trivial.

This paper introduces a dataset that contains the source code of 360 C# repositories in a simplified form. The dataset is primarily meant for use in research on source-based RSSE and to answer questions that target correct usage of *application programming interfaces* (API), but it could also be used in related areas, e.g., anomaly detection. Instead of spending time on assembling a dataset and making all sources compile, researchers can simply use our curated dataset for their research. Reusing an existing dataset has the added benefit of improving comparability and reproducibility of the results.

Additionally, most research in the field focuses on Java code. We have observed that there are not many C# datasets available. According to the TIOBE index, C# is among the most commonly used programming languages, so we believe that it is important to close this gap.

We surveyed a selection of 23 source-based RSSE and designed an intermediate representation (IR) of source code that contains all the information necessary to satisfy the requirements of the underlying recommender techniques. The details of the survey are published as an online appendix to this paper on the artifact page [2].

The IR also provides a means to bridge the gap between programming languages. While the current dataset contains only C# code, we designed the IR to support both Java and C# such that tools can be build once to support both languages. This also opens the opportunity to study the effect of programming language on RSSE.

2. DATASET CREATION

A C# repository contains one or more *solutions*, a C# specific construct that represents a collection of several related *projects*. Our dataset is created from a set of 360 Github repositories that contain 2833 solutions. We transformed the source code found in each project into our IR. The dataset represents more than 42.8M lines of code extracted from ~360K classes. It contains usages for more then 560 unique APIs (excluding all versions of the `mscorlib`, which is always used), out of which 52 APIs are used in at least 20 different projects (82 in 10, 123 in 5). The full statistics can be found on the artifact page.

Project Selection. We used the Github API to automatically find repositories. The number of available repositories is huge, so we restricted the search to a specific set of commonly used API types that were relevant to a recommendation system we were building. This list of types is included in our artifact page. We added all returned repositories to our dataset and transformed the contained solutions. A checkout script is provided that clones all included repositories.

The solutions contained in the selected repositories cover a wide variety of project types, including exemplary repositories that mainly contain tutorials for different parts of an API, small applications, and large-scale projects with many committers. The project domains in use are also quite diverse. For example, our selection includes web service clients, applications for machine learning, games, and applications with a graphical user interface. The selected repositories consist of a diverse mix of project sizes, the artifact page includes a graphical presentation of the distribution.

Used Tooling. The tool that we built to perform the transformation is based on ReSharper, a very common plu-

MSR'16, May 14-15, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4186-8/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2901739.2903507

gin for the Visual Studio IDE. In ReSharper, it is possible to open many different C# project types and resolve external NuGet dependencies. We implemented a runner that finds all solutions in a folder, opens them one by one, and then processes each API type found in the solution. We then traverse the abstract syntax tree and perform our transformation. Our tool is open source and can easily be used by others to transform additional solutions.

Data Organization. For easier distribution of the dataset, we combined everything needed into a single downloadable archive. The archive contains one folder per analyzed repository. Each folder contains .zip files, one for each analyzed solution of the repository. The zip file contains several files where each file contains the *Javascript object notation* (JSON) of each type declaration found in a project of the solution. Our organization in the file system makes it easy to identify the Github repository from which this data was created, as well as the path to the solution file within the repository.

After downloading the above archive file, using the data is straightforward. We provide bindings in both Java and C# that can be used to read and write the IR. Additionally, we provide utility functions that make it easy to find, read, and write the solution archives. Therefore, dataset users do not need to handle the file structure of the data themselves. We also provide code examples that explain how to read the dataset, traverse the IR with a visitor, access information, or perform processing steps.

3. DATA REPRESENTATION

The dataset is provided in an IR that we have developed. It is a simplified version of the original source code that still contains all information needed in source-based RSSE. Different source-based recommendation systems have a variety of requirements on the input data. Based on our survey, we found that a representation that is close to source code works best. In addition, we designed the IR in a way that makes it easy to analyze later on. For example, we expand nested expressions and require explicit references everywhere.

Our intermediate representation consists of two parts. We first introduce our naming scheme that allows us to uniquely refer to source-code elements in object-oriented programs. For example, whenever we refer to a method, the reference does not only contain the name, but also the declaring type, the signature of the methods and also versions of all involved types. We then present the full IR language that encodes type declarations, including a simplified version of the syntax tree and information about the involved types.

Naming. Part of our IR is a name representation for types and members that preserves fully-qualified typing information. The name grammar is shown in Figure 1. While the notation is inspired by the *extended Backus-Naur-Form* (eBNF), we did not intend the grammar to be usable for the automatic creation of a parser, but to make it easy for the reader to understand the data representation. We use “(...)” to group information and denote multiplicities in the commonly used way, i.e., “(...)?” is 0 or 1, “(...)∗” is 0 or more, and “(...)+” is 1 or more.

```
// basic non-terminals
Id = ... // an arbitrary string
Num = ... // a positive integer value
// types
Type = ‘?’ | <TypeParameter> | <DelegateType> | <ArrayType> |
    <RegularType>
TypeParameter = <Id>
...
RegularType = (<TypeQualifier> ‘:’)? <ResolvedType> ‘,’ <Assembly>
TypeQualifier = ‘e’ | ‘i’ | ‘s’ // enum, interface, struct
ResolvedType = (<Namespace>)? <TypeName> (‘+’ <TypeName>)∗
Namespace = (<Id> ‘.’)+
TypeName = <Id> (<GenericPart>)?
GenericPart = ‘’’ <Num> ‘[’ <GenericParam> (‘,’ <GenericParam>)∗ ‘]’
GenericParam = ‘[’ <TypeParameter> (‘->’ <Type>)? ‘]’
Assembly = <Id> (‘,’ <AssemblyVersion>)?
AssemblyVersion = <Num> ‘.’ <Num> ‘.’ <Num> ‘.’ <Num>
// member names
Member = (‘static ’)? ‘[’ <Type> ‘] [’ <Type> ‘].’ <Id>
Field = <Member>
Method = <Member> <GenericPart> ‘(’ (<Param> (‘,’ <Param>)∗)? ‘)’
Param = (<ParamModifier>)? ‘[’ <Type> ‘]’ <Id>
ParamModifier = ‘opt ’ | ‘out ’ | ‘params ’ | ‘ref ’ | ‘this ’
...
```

Figure 1: Name Grammar

The syntax of the naming scheme is driven by the idea to fully-qualify all identifiers and to encode all typing-related information about a code element in its name. For example, as can be seen from the grammar, a type does not only contain the full namespace information; it also includes the name of the dependency (*assembly*), including its version; as well as information about generic type parameters.

Consider the type “`i:data.IList, collections, 1.2.3.4`”. It refers to the `IList` type defined in the `data` namespace. The prefix `i:` denotes that this is an interface type, other supported kinds are enums and structs. If the prefix is omitted, then the type refers to a class. When a type is defined in a dependency, then both the name of this dependency (in this case `collections`) and the version is used. If the referenced type is instead defined in the current solution, then the name of enclosing project is used and the version is not set (e.g., “`T,P`”: type `T` in project `P`). The notation can also encode array types and delegate types. Both are omitted for brevity, but can be found in the full grammar on the artifact page.

We also support types with generic type parameters. Consider the type “`T’2[[G1],[G2->T2,P]],P`”, which has “`’2`” generic type parameters, of which `G1` is not bound and `G2` is bound to “`T2,P`”. Additionally, we also support nested classes. The example “`n.T+NT,P`” refers to the class `NT` nested in `T`.

This type information is used for other code elements as well. Our grammar introduces a syntax for member names and lambda names, but the following examples are restricted to method names for brevity. A method is represented as “`[RT,P] [DT,P].M()`” in our notation. This refers to a method `M` that is defined in the type “`DT,P`”, returns an object of type “`RT,P`”, and has no parameters. Method parameters are listed between the parentheses: ...`M([T3,P] p)` Like all class members, methods can include the `static` modifier. They can also contain additional modifiers for parameters.

Full Language. The source code we collect is represented in our newly developed IR that represents a simplified syntax tree of the source code. We present a part of the grammar in Figure 2 that uses the naming scheme as a building block. The IR is very similar to JSON, so we slightly adapt the eBNF notation to increase the readability: instead of using terminals for the curly braces, we use the construct {a:<A>} to represent an object that has a property *a* of type *A*. The construct <A>∗ is used to express the storage of a collection of <A>. We leave out some non-terminals for brevity, but include the full grammar on the artifact page.

The language is designed to capture information about a single type declaration. The root element is an <IR> node that contains both the <TypeShape>, a structure that captures information about the type system, and the simplified syntax tree of a <TypeDecl>.

In the <TypeShape>, information about the type hierarchy is stored. This includes references to all extended classes and implemented interfaces, but also more detailed information about the methods declared in the class. A <MethodHierachy> stores the local method name, but captures the information if a method was overridden and -if sowhich one. A method might be overridden multiple times in a given type hierarchy. We store the name of the “super” method and the name of the method that originally introduced the signature.

In the <TypeDecl>, the contents of the class are captured in a simplified form. The representation is inspired by the language specification of Java and C#, but provides several simplifications that make it easier to analyze. The two most important ideas are that 1) the IR does not allow the nesting of arbitrary expressions in the syntax tree, it requires the source-code to be normalized. And 2), it also does not allow any implicit information (like optional `this` references). This has several implications on the transformation. For example, the transformation has to handle nested method calls like `m1(m2())`. In this case, our transformation introduces a temporary variable, assigns the nested invocation `m2()`, and passes the variable as a parameter to `m1`.

The IR also enforces some unification of the source code. Consider, the example `if(isX())...`. Some developers want to assign `isX()` to a variable first, while others will not. The IR enforces the former style by only allowing <SimpleExpr> in conditions, i.e., <ConstantExpr> or <ReferenceExpr>. Note that this is not possible in loops (e.g., `while(isX())...`), because the loop condition is evaluated multiple times. You could include the condition before the loop and at the end of the loop, but this would introduce duplication. The IR avoids this problem by introducing the concept of <LoopHeaderBlockExpr>, which can contain multiple statements.

```
IR = {typeShape:<TypeShape>, type:<TypeDecl> }
// type shape
TypeShape = {th:<TypeHierarchy> mh:<MethodHierarchy>∗}
TypeHierarchy = {elem:<Type>, extends:(<TypeHierarchy>)?,
  implements:<TypeHierarchy>∗}
MethodHierarchy = {elem:<Type>, super:(<Type>)?, first:(<Type>)?}
// declarations
TypeDecl = {events:<EventDecl>∗, fields:<FieldDecl>∗,
  methods:<MethodDecl>∗, properties:<PropertyDecl>∗}
FieldDecl = {name:<Field>}
MethodDecl = {name:<Method>, body:<Statement>∗}
...
// statements
Statement = <Block> | <Assignment> | <BreakStmt> | <ContinueStmt> |
  <EventSubscriptionStmt> | <ExpressionStmt> | <GotoStmt> |
  <LabelStmt> | <ReturnStmt> | <ThrowStmt> | <UnknownStmt> |
  <VariableDecl>
Assignment = {ref:<AssignableRef>, expr:<AssignableExpr>}
ExpressionStmt = {expr:<AssignableExpr>}
VariableDecl = {id:<VariableRef>}
...
// blocks
Block = <DoLoop> | <ForEachLoop> | <ForLoop> | <IfElseBlock> |
  <LockBlock> | <SwitchBlock> | <TryBlock> | <UncheckedBlock> |
  <UnsafeBlock> | <UsingBlock> | <WhileLoop>
IfElseBlock = {cond:<VariableRef>, then:<Statement>∗, else:<Statement>∗}
WhileLoop = {cond:<LoopHeaderExpr>, body:<Statement>∗}
...
// expressions
SimpleExpr = <ConstExpr> | <ReferenceExpr> | <UnknownExpr>
ConstExpr = {value:<Id>}
ReferenceExpr = {ref:<Reference>}
LoopHeaderExpr = <SimpleExpr> | <LoopHeaderBlockExpr>
LoopHeaderBlockExpr = {body:<Statement>∗}
AssignableExpr = <SimpleExpr> | <BinaryExpr> | <CastExpr> |
  <CompletionExpr> | <IfElseExpr> | <IndexAccessExpr> |
  <InvocationExpr> | <LambdaExpr> | <TypeCheckExpr> | <UnaryExpr>
InvocationExpr = {ref:<VariableRef>, method:<Method>,
  parameters:<SimpleExpr>∗}
...
// references
Reference = <AssignableRef> | <MethodRef> | <EventRef>
AssignableRef = <VariableRef> | <IndexAccessRef> | <FieldRef> |
  <PropertyRef> | <UnknownRef>
VariableRef = {id:<Id>}
FieldRef = {ref:<VariableRef>, name:<Field>}
...
```

Figure 2: Excerpt of the IR Grammar

4. HOW TO USE

We now describe how we have made use of the current dataset as well as how we foresee future researchers using it.

Previous Uses. We have recently used the presented dataset and our IR to create a platform for modular development of source-based recommender systems. The IR allowed us to build general analysis modules such as a points-to analysis and an inlining component that are not specific to an approach and, therefore, easily reusable.

As a use case, we reimplemented an existing recommendation system that builds on top of the IR [4]. We then used the described dataset to train the recommender. Using it provided us easy access to a large number of solutions and was also a means for creating reproducible results.

Dataset. The dataset and the data format is tailored to the needs of source-based recommendation systems. Our dataset as well as future data sets encoded in our new IR serve as standardized input to build and evaluate RSSE. Our current data set can be used in a wide range of approaches; for example, to build intelligent completion engines such as call or snippet completion, code search engines, anomaly detectors, or simply to gather source code statistics.

However, the data contained in the dataset is very general and is not limited to RSSE. We believe it is useful for any research that involves source code, as long as the code information needed is contained in our IR.

Transformation. The tool we created for the source-code transformation is open source. Since the tool can analyze arbitrary solutions that exist locally, researchers can use it to create their own additional datasets as well

Another use case is researchers integrating our transformation into their Visual Studio integrated RSSE. They can use our transformation as a preprocessing step that transforms the current code under edit into the IR. This solves many lower-level recurring code analysis tasks such that they can concentrate on writing more sophisticated analyses on top.

Intermediate Representation. The IR itself is a contribution that could be applied by others to represent source code. For example, a data set of StackOverflow posts may need to store code snippets that are part of the post. These code snippets could be stored in our IR such that the same analyses and processing tools can be applied on them as on transformed source code from other sources.

The IR also serves a means to design cross-language analyses that investigate the differences between APIs and their usages in different programming languages. Having the IR as common ground opens the possibility to have a single toolchain to process and evaluate programs originally written in different programming languages. We are currently working on such an IR transformation for Java.

5. EXTENDING THE DATASET

We foresee multiple ways in which the dataset could be extended or further improved.

Intermediate Representation. The dataset could be extended by supporting more code elements in the IR. Examples of this include visibility modifiers, comments, and attributes. In the same way, the transformation could be extended to cover more cases. For example, support for the syntactic sugar of textual C#'s Language-integrated Query (LINQ) expressions could be added.

Dataset. An obvious way to extend the dataset is to use more repositories. This could include a larger number of repositories from Github, as well as from other platforms such as Bitbucket or SourceForge.

We did not explicitly aim for creating a dataset that is representative for different factors such as project size, complexity, domain, etc. Thus, future work could introduce a more structured way of selecting the included repositories and apply a measure to quantify the representativeness of the dataset similar to that sketched in our previous work [3].

6. LIMITATIONS

The dataset is meant to be used for research on source-based RSSE. While we tried to provide a dataset that covers as many applications as possible, it also has some limitations.

Transformation. Our transformation tool is built on top of ReSharper. We use this framework to open the solutions and for the automatic dependency resolution. Unfortunately, it is restricted to Visual Studio solutions and it does not support all possible C# project types. This includes for example solutions that make use of the most recent .NET version or some Windows Phone projects.

Our transformation works on the ReSharper AST that contains resolved typing information. We transform many language constructs into our IR, but supporting the complete language specification of C# was beyond the scope of our immediate goals. We prioritized the language constructs and focused on supporting the most common ones, while ignoring less-commonly used ones for the time being. For example, we do not transform the syntactic sugar of textual LINQ expressions and we currently ignore the contents of `unsafe` blocks. Unsupported constructs will be included in the IR as unknown expressions or unknown statements. We continue the work on the transformation and plan to support missing language features with every new release.

Additionally, the IR, by design, makes it necessary to expand nested expressions in the source code. For example, it is not possible to nest a method invocation as a parameter of another method invocation in the IR. While the normalization of these cases makes it a lot easier to analyze the code, the current transformation just assigns nested expressions to new temporary variables. This might change the semantics of the code, for example, it breaks short-circuit evaluation. Please note that this is not a conceptual limitation of the IR, but simply a matter of engineering effort to improve and extend the transformation.

Repository Selection. Our current selection of repositories covers a wide variety of applications and project sizes. The repositories are only selected from GitHub. Additionally, no meta data is available in the dataset that makes it possible to select a representative subset of repositories and no representativeness guarantees exist for the dataset. For example, it might be that the selected repositories are biased towards a specific domain or contain more research projects than industrial open-source projects. However, we believe that this does not limit the use of the dataset unless the target research problem explicitly requires an equal distribution of different domains, project size, etc.

Data Format. We do not include all information of the original source code in our IR and leave out code elements that we did not deem required after surveying the 23 RSSE. For example, this includes visibility modifiers, attributes, or comments. We found that existing RSSE do not leverage this information. If future approaches require the presence of these elements, adding them is only a matter of extending the grammar and considering the elements in the transformation.

7. CONCLUSION

We presented a dataset containing the source code of 2833 C# solutions taken from Github. The source code of these solutions is encoded in a newly developed intermediate representation that resembles a simplified syntax tree. This IR facilitates further automated code analysis and we provide bindings to read and process the dataset in both C# and Java. The dataset can be used for research on RSSE, like intelligent code completion, code search, or snippet mining.

8. ACKNOWLEDGMENTS

The work presented in this paper was partially funded by the German Federal Ministry of Education and Research (BMBF) within the Software Campus projects KaVE and Eko (both grant no. 01IS12054). The authors assume responsibility for the content.

9. REFERENCES

[1] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Graph-based Pattern-oriented, Context-sensitive Source Code Completion. In *International Conference on Software Engineering*. IEEE, 2012.

[2] Online Appendix. http://www.st.informatik.tu-darmstadt.de/artifacts/sst/.

[3] S. Proksch, S. Amann, and M. Mezini. Towards standardized evaluation of developer-assistance tools. In *International Workshop on Recommendation Systems for Software Engineering*. ACM, 2014.

[4] S. Proksch, J. Lerch, and M. Mezini. Intelligent Code Completion with Bayesian Networks. *ACM Transactions on Software Engineering and Methodology*, 2015.

[5] V. Raychev, M. Vechev, and E. Yahav. Code Completion with Statistical Language Models. In *Conference on Programming Language Design and Implementation*. ACM, 2014.

[6] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending api usage patterns. In *ECOOP–Object-Oriented Programming*. Springer, 2009.