

Cross-Language Clone Detection

Nicholas A. Kraft, Brandon W. Bonds, and Randy K. Smith
Department of Computer Science
The University of Alabama
Tuscaloosa, AL 35487, USA
{nkraft, bbonds, rsmith}@cs.ua.edu

Abstract

Code duplication is a common software development practice that introduces several similar or identical segments of code, or code clones. In addition, there is currently a trend towards the use of multiple languages in the development of software systems. While there has been much work on clone detection and increased interest in studies of multi-language software systems, there have been no studies of code clones that span multiple languages, which we term cross-language code clones. In this paper we describe an approach for cross-language clone detection. We then introduce a tool, which is based on the CodeDOM library that is included with the Microsoft .NET Framework, to demonstrate the existence of cross-language clones in a real software system that contains both C# and Visual Basic.NET source code. Because our clone detection algorithm operates on a tree structure, other tree-based clone detection algorithms could be substituted in the implementation of our tool.

1. Introduction

Code duplication is a common software development practice that introduces several similar or identical segments of code, or *code clones*, and has many forms: language construct recurrence, pattern or paradigm adherence, framework reuse, and copy-paste replication. There are some advantages to duplicating code; developer productivity can be improved through the use of copy-paste to quickly replicate functionality. However, there are also important disadvantages to duplicating code; program comprehensibility, maintainability, and correctness can all be adversely affected by code duplication practices, particularly copy-paste replication. For example, when a change is to be made to a duplicated piece of code, a developer must determine whether the change should be made to all duplicate instances (clones) of that code. If the developer makes an

incorrect determination, or accidentally misses a duplicate instance, then a bug is introduced.

Because code clones can have a negative impact on system quality, techniques and tools to eliminate code clones have been proposed. For example, techniques for automatic refactoring have been described [3, 5] and tools to guide manual refactoring [12] and to allow meta-level refactoring [4] have been developed. However, studies have shown that some code clones are inherent and that their elimination is not always desirable [15, 16]. Therefore, additional techniques, such as linked editing [38], are needed to mitigate the unnecessary maintenance costs associated with code clones [11].

Irrespective of the benefits or drawbacks attributed to a code clone or a category of code clones, techniques and tools to detect code clones are needed, and much work on clone detection has been performed. These techniques and tools use as inputs a variety of program representations, including source code [1, 2, 8, 14, 25], parse or abstract syntax trees [5, 9, 13, 19, 20, 37, 39], abstract semantic graphs [28], and program dependence graphs [17, 22]. Furthermore, these techniques and tools use a variety of matching approaches, including string or token comparison [1, 2, 8, 14], metric comparison [28], hashing [5], subgraph isomorphism [17, 22], feature vectors [19, 13], frequent item sets [25, 39], suffix trees [20, 37], and structural abstraction [9]. Direct comparisons and evaluations of some of these techniques and tools are provided in the literature [6, 33].

While several of the aforementioned clone detection techniques can be adapted to multiple languages [32] and some of the aforementioned clone detection tools accept more than one source language (for example, CCFinder [14] can detect code clones in Java, C, C++, C#, Visual Basic, and COBOL programs), all of the techniques and tools focus on detecting same-language code clones, that is, code clones for which each associated code segment is written in the same source language. However, there is currently a trend towards the development of heterogeneous soft-

ware systems in which multiple languages are used. The Microsoft .NET Framework [29], which leverages a common byte code format and virtual machine to allow programs written in different languages to interact, is a central actor in this trend. While this trend has caused increased interest in studying multi-language software systems [18, 27, 26, 30, 35], there are currently no approaches for detecting code clones that span multiple languages or studies of such clones.

In this paper we describe an approach for detecting code clones that span multiple languages, which we term *cross-language code clones*. Our approach is complementary to other clone detection techniques, as we are focused primarily on issues associated with detecting cross-language code clones and not on the mechanics of clone detection. We also introduce a tool, C2D2, that implements cross-language clone detection for the .NET Framework. We then apply C2D2 to two subsystems of a real, open-source software system, MonoTM [31]. Our results demonstrate the existence of cross-language code clones, specifically code clones that span the C# and Visual Basic.NET languages.

The rest of the paper is organized as follows. In Section 2 we provide background information about the libraries that we use to implement our cross-language clone detection system, C2D2, and in Section 3 we provide an overview of C2D2. In Section 4 we describe a case study and report our results. Finally, we discuss related work in Section 5 and conclude in Section 6.

2. Background

In this section we provide background information about the Code Document Object Model (CodeDOM) and NRefactory. Both are libraries that we use to implement our cross-language clone detection system, C2D2.

2.1. CodeDOM

The Microsoft .NET Framework includes the Code Document Object Model (CodeDOM) library, which provides a language-independent metamodel for representing source code [7]. CodeDOM exists primarily to allow developers of programs that emit source code, such as the Visual Studio.NET forms designer, to emit source code in multiple programming languages from a common representation. A CodeDOM graph is a graph of CodeDOM nodes that represent the logical structure of source code and can be used both to generate source code in any language for which an implementation of the `ICodeGenerator` interface is provided.

CodeDOM is similar to other metamodels for abstract syntax trees [10, 21] but is more similar to the Dagstuhl

Middle Metamodel (DMM) [23] and the Common Meta-Model (CMM) [36]. Like the DMM and the CMM, CodeDOM provides classes that represent many common object-oriented language constructs, including namespaces, type declarations (classes, structs, and enumerations), methods, fields, exceptions, and control statements. Also like the CMM, CodeDOM provides classes that represent statements and expressions, and is extensible. The expressiveness of CodeDOM is limited by the necessity of providing the option to generate code in any language in the .NET-family. Thus, CodeDOM can express only constructs that are available in all .NET languages.

Language constructs that are not directly supported by CodeDOM can be represented using a generic node, or “snippet” node. However, in many cases, source code can be restructured to avoid the use of snippets. For example, the switch statement is not present in all .NET languages, but can be restructured as a series of `if` statements. Such a restructuring ensures that the desired functionality can be achieved across all .NET languages.

2.2. NRefactory

Microsoft implements the interfaces in the CodeDOM library that allow a developer to populate a CodeDOM graph programmatically, but they do not implement the interfaces that would allow a developer to populate a CodeDOM graph from existing source code. The SharpDevelop IDE [34] includes the NRefactory library, which provides parsers for both C# and Visual Basic.NET; each of these parsers builds an internal abstract syntax tree (AST) representation of the input program. The `CodeDomOutputVisitor` class of NRefactory traverses the internal AST to produce a CodeDOM graph that represents source code that is semantically equivalent to the source code provided as input to the parser. However, because the expressiveness of the CodeDOM is limited, the source code generated from the produced CodeDOM graph may not be syntactically identical to the original source code.

3. C2D2

In this section we present our cross-language clone detection system, C2D2, including an overview of the system, a description of changes we made to NRefactory, and the details of our clone detection algorithm.

Figure 1 illustrates an overview of the C2D2 system, which takes as input one or more C# or Visual Basic.NET source files and produces as output a listing of detected clones. The source files, which are shown in the upper left of the figure, are read by the convertor component, which is shown towards the lower left of the figure. The convertor component converts each source file to a CodeDOM graph

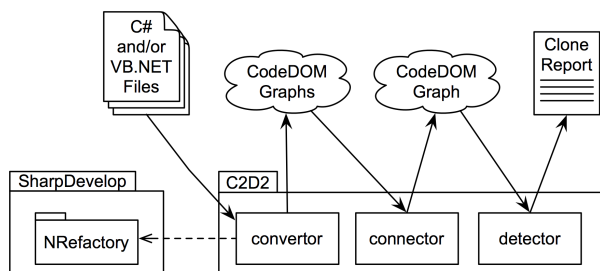


Figure 1. Overview of C2D2. *Dashed lines represent “use” dependencies. Solid lines represent data flow. Tabbed files are provided by the user. Clouds represent internal data structures. Lined files are generated by the system.*

by passing it to the NRefactory library, which is shown in the lower left of the figure. The connector component, which is shown in the lower middle of the figure, connects the collection of CodeDOM graphs produced by the converter component to form a unified CodeDOM graph. The detector component, which is shown in the lower right of the figure, detects clones in the unified CodeDOM graph produced by the connector component. Output of the detector component is a clone report that lists the detected code clones.

3.1. Changes to NRefactory

We made several changes to the NRefactory library to fix bugs and to provide new functionality. With these changes, the CodeDOM graphs produced by the NRefactory parsers are usable for clone detection. Our corrections and additions to the NRefactory code are located primarily in the `CodeDomOutputVisitor` class. For example, in C# the `using` keyword can be followed by either an expression or a statement; however, only the expression case was handled by the version of NRefactory that we downloaded (version 2.2.0.2532). We added code to handle the statement case.

The parsers provided by NRefactory annotate some (but not all) AST nodes with line and column numbers, but do not propagate this information to the produced CodeDOM graphs. However, CodeDOM allows graph nodes to be annotated with a dictionary of user data via the `UserData` property. We leveraged this property to add line and column numbers (along with other auxiliary data provided by the NRefactory lexers and parsers) to the nodes in the CodeDOM graph.

Our efforts to correct and enhance the NRefactory library are ongoing. Despite our efforts, there remain C# constructs that are either not properly parsed or not prop-

erly translated to CodeDOM; we are not yet able to obtain a CodeDOM graph for NRefactory due to these deficiencies. Nonetheless, our initial efforts allow the CodeDOM graphs produced by our modified version of NRefactory to be used for our feasibility study.

3.2. Clone Detection Algorithm

We utilize a hybrid token/tree-based algorithm for clone detection. Tree-based detection algorithms tend to be slower and more complex than token-based algorithms, so we wished to use a token-based algorithm to complete our feasibility study of cross-language clone detection. However, our input structure is a tree, not a token stream; thus, we created a hybrid token/tree-based algorithm.

The first step in our algorithm is to traverse the tree that underlies the unified CodeDOM graph and to create and store a string token for each leaf node in the tree. We perform the traversal depth-first to allow each interior node in the tree to access the tokens of its children during the traversal. An interior node stores the tokens of its children in a list and in prefix order. After traversal of the tree is complete, the list in the root node contains the token for each node in the tree, and so on down the tree. During the traversal, we also store each tree node in a list of nodes of the same type; one list exists for each CodeDOM node type.

Our matching algorithm is based on the Levenshtein distance algorithm [24]. The Levenshtein distance between two strings is the minimum number of character insertions, deletions, or substitutions required to transform one of the strings into the other string. We adapt this algorithm to work on lists of tokens representing CodeDOM nodes (recall that each CodeDOM node in the tree, except for a leaf node, contains the list of tokens for its children), which in turn represent code segments. We determine the minimum number of token insertions, deletions, or substitutions required to transform one list of tokens into the other list of tokens. We store our results in a data structure that stores the number of tokens cloned and the percentage matching (cloning) between two lists of tokens (code segments).

We only apply our matching algorithm on CodeDOM nodes of the same type. This is equivalent to only applying the algorithm on AST nodes of the same type, or to only attempting to match code segments of the same syntactic form. By intelligently applying our matching algorithm we reduce the complexity of the clone detection process and greatly improve performance. In addition, we eliminate the possibility of detecting a clone that crosses a block boundary; thus, we retain a key advantage that tree-based clone detection approaches have over token-based approaches.

Our matching algorithm takes four parameters. The first parameter specifies a minimum number of tokens that a list may contain to be considered, and the second parameter

# of Clones	Total	Percentage Matching					Tokens Matched						
		30–40	40–50	50–60	60–70	70–100	0–10	10–20	20–30	30–40	40–50	50–60	60–70
	8,029	7,347	361	188	10	0	805	4,328	2,590	278	25	3	0

Table 1. Cross-Language Clones. *The number of cross-language clones detected by C2D2, categorized by percentage matching and by tokens matched.*

```

static public void EmitInt (ILGenerator ig, int i)
{
    switch (i){
    case -1:
        ig.Emit (OpCodes.Ldc_I4_M1);
        break;
    ...
    default:
        if (i >= -128 && i <= 127){
            ig.Emit (OpCodes.Ldc_I4_S, (sbyte) i);
        } else
            ig.Emit (OpCodes.Ldc_I4, i);
        break;
    }
}

Shared Sub EmitLoadI4Value(ByVal Info As EmitInfo, \
                          ByVal I As Integer, \
                          ByVal TypeToPushOnStack As Type)
    TypeToPushOnStack = \
        Helper.GetTypeOrTypeBuilder (TypeToPushOnStack)
    Select Case I
    Case -1
        Info.ILGen.Emit (OpCodes.Ldc_I4_M1)
    ...
    Case SByte.MinValue To SByte.MaxValue
        Dim sbit As SByte = CSByte(I)
        Info.ILGen.Emit (OpCodes.Ldc_I4_S, sbit)
    Case Else
        Info.ILGen.Emit (OpCodes.Ldc_I4, I)
    End Select
    Info.Stack.Push (TypeToPushOnStack)
End Sub

```

Figure 2. Example Cross-Language Clone. *A portion of a cross-language clone detected by C2D2. The left side of the figure lists a C# code segment from `mcs/constant.cs` and the right side of the figure lists a Visual Basic.NET code segment from `vbnc/source/Emit/Emitter.vb`. The full clone spans lines 805–855 of `mcs/constant.cs` and lines 1384–1414 of `vbnc/source/Emit/Emitter.vb`, and has a percentage matching of 58 (69 of 118 tokens matched).*

specifies the corresponding maximum number. These parameters can be used to reduce the number of matches attempted by the algorithm; please note that these parameters specify the numbers of tokens a list may contain to be considered by the algorithm and not the numbers of tokens that a clone may contain. The third parameter specifies a minimum percentage matching. This parameter can be used to filter clones that do not reach the specified percentage of similarity. Finally, the fourth parameter specifies whether to attempt to detect same-language clones (in addition to cross-language clones).

4. Case Study

In this section we describe a case study used to evaluate the feasibility of our approach. We use the C# and Visual Basic.NET compilers from Mono [31], version 1.2.6, as the test case. We chose these compilers because they are open-source and are part of a large, multi-language software system. Furthermore, as compilers for a common back end (the .NET runtime environment) they are likely to have common functionality in their code generators.

The C# compiler, `mcs`, consists of 38 C# files that contain 49,216 lines of non-blank, un-commented, non-

preprocessed source code. The Visual Basic.NET compiler, `vbnc`, consists of 422 Visual Basic.NET files that contain 52,161 lines of non-blank, un-commented, non-preprocessed source code. Thus, our test case totals 460 source files and 101,377 lines of source code. Before running C2D2 on the test case, we configured it as follows: minimum number of tokens (50), maximum number of tokens (200), minimum cloning percentage (30), and attempt to detect same-language clones (no). We performed all experiments on a workstation with an AMD Athlon™ 64 X2 3800+ processor and 1 GB RAM; the operating system is Microsoft Windows XP Professional SP2 32-bit. Using the above configuration and the test case as input, the total execution time for C2D2 is 21 minutes and 23 seconds (wall clock time).

Table 1 summarizes the results of the execution. A total of 8,029 cross-language clones were detected; however, 7,708 of the detected clones had a percentage matching of less than 50, leaving only 198 clones with a percentage matching of 50 or greater. Of those 198 clones, only 10 have a percentage matching of 60 or greater and none have a percentage matching of 70 or greater. At first glance, these results do not appear to demonstrate the existence of meaningful cross-language clones. Yet, the example clone illustrated

in Figure 2 shows that percentage matching is a potentially misleading metric for cross-language clones detected using our algorithm. Figure 2 illustrates a cross-language clone that consists of a segment of C# code and a segment of Visual Basic.NET code. If it were not for the stack manipulation performed in the Visual Basic version of the code then the functionality would be identical. This example clone is a *type 3* clone [6], i.e., a clone in which identifiers have been changed and statements have been changed, added, or removed (added in this case). However, the percentage matching for the clone is only 58% (69 of 118 tokens matched), which might seem low.

Despite the discovery of the cross-language clone illustrated in Figure 2 and others, when compared to the results of other clone studies, the number of detected clones with a significant percentage matching appears to be low for the amount of code included in the test case. There are several possible explanations for this. First, as noted above, percentage matching seems to be a misleading metric for evaluating cross-language clones detected using our algorithm. Second, Mono simply might not have many cross-language clones; this seems likely given that different projects within Mono often have disparate development teams. However, this would indicate that study of additional multi-language software systems is warranted. We plan to undertake such studies in the future. A third possibility is that C2D2, while capable of finding some cross-language clones (as demonstrated above), might miss other cross-language clones. This could be due to the clone detection algorithm employed. Again, this would indicate that further study, including applying more advanced tree-based clone detection techniques, is warranted.

5. Related Work

In this section we discuss related work, which we divide into two categories: (1) studies of clones and (2) studies of multi-language software systems.

5.1. Studies of Clones

Many techniques and tools for clone detection exist. These techniques and tools operate on a variety of program abstractions, including strings [1, 2, 8], tokens [14, 25], trees [5, 9, 13, 19, 20, 37, 39], and graphs [17, 22, 28]. Because our approach detects clones on a tree structure, other tree-based clone detection algorithms could be substituted for the one we present.

Studies investigating the presence of code clones in single-language software systems have found significant amounts of duplicated code within these systems. These studies, known as clone coverage studies, suggest that it

is not uncommon to have over 20% cloned code in a software system. For example, CCFinder detected almost 30% cloned code in version 1.3.0 of the JDK [14], and CP-Miner detected over 22% cloned code in version 2.6.6 of the Linux kernel [25]. In addition, one study reported over 59% cloned code in a COBOL payroll application [8].

5.2. Cross-Language Studies of Multi-Language Software Systems

There is currently a trend towards the development of heterogeneous software systems in which multiple languages are used [18]. This trend has caused increased interest in studying multi-language software systems. Topics of interest include modeling, usability, tool support, as well as maintenance processes, which would include clone detection and evolution.

Linos, et al. [27] and Moise and Wong [30] present studies of cross-language dependencies found in software systems written in C/C++ and Java. Strein, et al. [35] present an approach to cross-language program analysis for refactoring and a prototype tool that handles both C# and Visual Basic.NET; however, they do not leverage CodeDOM for their implementation. Finally, Linos, et al. [26] present an approach to computation of metrics on MSIL (Microsoft Intermediate Language). Their ultimate goal is to determine whether computation of metrics at the MSIL level is as effective as computation of metrics directly on source code.

6. Conclusions and Future Work

The current trend towards the use of multiple languages in the development of software systems introduces new challenges for software comprehension and software maintenance. Many techniques and tools for clone detection, elimination, and removal have been described in the literature, and some of this literature addresses the problem of applying these techniques and tools to software systems written in a variety of languages. However, none of these techniques or tools have been applied to multi-language software systems with a focus on detecting code clones that span multiple languages.

In this paper we introduced an approach for detecting code clones that span multiple languages, which we termed *cross-language code clones*. Our approach complements other clone detection techniques, as it is focused on issues of detecting cross-language code clones and not on the mechanics of clone detection. We described an approach for cross-language clone detection and presented a tool, C2D2, that implements cross-language clone detection for the .NET Framework. Our experimental results demonstrate the existence of cross-language code clones, specifically code clones that span C# and Visual Basic.NET.

As future work we propose to enhance the usability and interoperability of C2D2 by producing output files that can be read by existing clone visualization systems, to integrate more advanced tree-based clone detection techniques into C2D2, and to perform more extensive studies of multi-language software systems.

References

- [1] B. S. Baker. On finding duplication and near-duplication in large software systems. In *WCRE*, pages 86–95, July 1995.
- [2] B. S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SICOMP*, 26(5):1343–1362, Oct. 1997.
- [3] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Partial redesign of java software systems based on clone analysis. In *WCRE*, pages 326–336, October 1999.
- [4] H. A. Basit, D. C. Rajapakse, and S. Jarzabek. Beyond templates: A study of clones in the STL and some general implications. In *ICSE*, pages 451–459, May 2005.
- [5] I. D. Baxter, A. Yahin, L. M. de Moura, M. Sant’Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM*, pages 368–377, November 1998.
- [6] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *TSE*, 33(9):577–591, Sept. 2007.
- [7] Code Document Object Model. <http://msdn2.microsoft.com/library/system.codedom.aspx>.
- [8] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *ICSM*, pages 109–118, August/September 1999.
- [9] W. Evans, C. Fraser, and F. Ma. Clone detection via structural abstraction. In *WCRE*, pages 150–159, October 2007.
- [10] R. Ferenc, S. E. Sim, R. C. Holt, R. Koschke, and T. Gyimothy. Towards a standard schema for C/C++. In *Proceedings of the 8th Working Conference on Reverse Engineering, Stuttgart, Germany*, pages 49–58. IEEE Computer Society, Oct. 2001.
- [11] R. Geiger, B. Fluri, H. C. Gall, and M. Pinzger. *Relation of Code Clones and Change Couplings*, volume 3922 of *LNCIS*, pages 411–425. 2006.
- [12] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. ARIES: Refactoring support environment based on code clone analysis. In *SEA*, pages 222–229, November 2004.
- [13] L. Jiang, G. Mishnerghi, Z. Su, and S. Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *ICSE*, pages 96–105, May 2007.
- [14] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. *TSE*, 28(7):654–670, July 2002.
- [15] C. Kapser and M. Godfrey. “Cloning considered harmful” considered harmful. In *WCRE*, pages 19–28, October 2006.
- [16] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. *SEN*, 30(5):187–196, Sept. 2005.
- [17] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *SAS*, pages 40–56, July 2001.
- [18] K. Kontogiannis, P. Linos, and K. Wong. Comprehension and maintenance of large-scale multi-language software applications. In *ICSM*, pages 497–500, September 2006.
- [19] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *JASE*, 3(1/2):77–108, July 1996.
- [20] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *WCRE*, pages 253–262, October 2006.
- [21] N. A. Kraft, B. A. Malloy, and J. F. Power. An infrastructure to support interoperability in reverse engineering. *Information and Software Technology*, 49(3):292–307, Mar. 2007.
- [22] J. Krinke. Identifying similar code with program dependence graphs. In *WCRE*, page 301, October 2001.
- [23] T. C. Lethbridge, S. Tichelaar, and E. Ploedereder. *The Dagstuhl Middle Metamodel: A Schema for Reverse Engineering*, volume 94 of *Electronic Notes in Theoretical Computer Science*, pages 7–18. Elsevier B.V., May 2004.
- [24] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.
- [25] Z. Li, S. Myagmar, and Y. Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *TSE*, 32(3):176–192, Mar. 2006.
- [26] P. Linos, W. Lucas, S. Myers, and E. Maier. A metrics tool for multi-language software. In *SEA*, pages 209–218, November 2007.
- [27] P. K. Linos, Z. Chen, S. Berrier, and B. O’Rourke. A tool for understanding multi-language program dependencies. In *IWPC*, page 64, May 2003.
- [28] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *ICSM*, pages 244–253, November 1996.
- [29] Microsoft .NET Framework. <http://msdn.microsoft.com/netframework/>.
- [30] D. L. Moise and K. Wong. Extracting and representing cross-language dependencies in diverse software systems. In *WCRE*, pages 209–218, November 2005.
- [31] Mono. <http://www.mono-project.com/>.
- [32] M. Rieger. *Effective Clone Detection Without Language Barriers*. PhD thesis, University of Bern, Switzerland, 2005.
- [33] C. K. Roy and J. Cordy. Scenario-based comparison of clone detection techniques. In *ICPC*, June 2008. To appear.
- [34] SharpDevelop IDE. <http://www.sharpdevelop.net/>.
- [35] D. Strein, H. Kratz, and W. Löwe. Cross-language program analysis and refactoring. In *SCAM*, pages 207–216, September 2006.
- [36] D. Strein, R. Lincke, J. Lundberg, and W. Löwe. An extensible meta-model for program analysis. *IEEE Transactions on Software Engineering*, 33(9):592–607, Sept. 2007.
- [37] R. Tairas and J. Gray. Phoenix-based clone detection using suffix trees. In *ACMSE*, pages 679–684, March 2006.
- [38] M. Toomim, A. Begel, and S. L. Graham. Managing duplicated code with linked editing. In *VL/HCC*, pages 173–180, September 2004.
- [39] V. Wahler, D. Seipel, J. Wolff, and G. Fischer. Clone detection in source code by frequent itemset techniques. In *SCAM*, pages 128–135, September 2004.