nameCollector: Extracting Source Code Identifiers Along with their Types and Context

John A. Sipahioglu
Department of Computer Science
Kent State University
Kent, OH, USA
jsipahio@kent.edu

Joshua A. C. Behler
Department of Computer Science
Bowling Green State University
Bowling Green, OH, USA
jbehler@bgsu.edu

Sophia Testa
Department of Computer Science
Kent State University
Kent, OH, USA
stesta4@kent.edu

Ali F. Al-Ramadan
Department of Computer Science
Kent State University
Kent, OH, USA
aalramad@kent.edu

Michael J. Decker
Department of Computer Science
Bowling Green State University
Bowling Green, OH, USA
mdecke@bgsu.edu

Michael L. Collard
Department of Computer Science
The University of Akron
Akron, OH, USA
collard@uakron.edu

Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent, Ohio, USA
jmaletic@kent.edu

***Abstract*—The work presents nameCollector, a lightweight tool built using the srcML infrastructure that collects all identifiers along with their data type, position, and syntactic category from source code. Identifier names are an essential component for program comprehension, code reviews, assessing software quality, and other maintenance activities. However, collecting identifier names from source code with contextual and type information is a non-trivial problem. Some compilers give access to this information, but few standalone tools exist that can easily collect the identifiers. nameCollector is currently capable of extracting identifiers along with their syntactic category, type information, and location from C, C++, C#, Java, and Python systems. nameCollector is scalable and can be used on single files and large systems. The tool is aimed at generating output for researchers to use in downstream analysis.**

Demo Video: https://youtu.be/_t76d5pMc3I

Keywords—program comprehension, identifier names, static analysis, srcML

I. Introduction

Conducting research on software systems often requires gathering all the programmer-defined names for analysis or processing. Investigations on program comprehension, refactoring, renaming, and naming standards are all concerned with how developers name classes, types, functions, and local variables.

However, collecting all the programmer-defined identifiers in a system is somewhat problematic. A researcher needing this information must custom-build a parser or, if available, attempt to use a compiler's API. Either option requires quite a lot of work and is difficult to get all the correct information necessary. Only declarations are of interest; therefore, simple parsing is generally insufficient. Extracting only the declarations of identifiers requires knowledge of the abstract syntax information of the source code. This is compounded if multiple programming languages are being studied. Methods for extracting identifiers tend to be either narrow and focused on a single programming language or are specific to certain workflows and not applicable for general use.

Studies on identifier names have implemented a variety of approaches to collect names from systems. Abebe et al. [7] and Newman et al. [8] leverage the srcML infrastructure (as does our work) to extract identifier names from software systems. Butler et al. [9] use their own custom parser to collect names from Java systems. Beniamini et al. [10] analyze identifiers from systems implemented in several programming languages, including C, Java, PHP, JavaScript, and Perl. To extract them, they implement separate parsers for each language, leveraging existing parsers or compiler output options. Malpohl, Hunt, and Tichy [11] use ANTLR, a parser generator, to construct the abstract syntax tree for Java programs to extract. Clang can be used to extract identifiers from C and C++ source code by walking the AST it generates. Roslyn (C#) and Java/Eclipse JDT both have APIs that also provide access to the abstract syntax tree with semantic information. However, using these APIs requires creating custom tooling for each language. To the best of our knowledge, no opensource publicly available tool currently exists to extract all identifier names along with such things as type and location information for multiple languages.

To address this need for research infrastructure (and support our own needs for such a tool) we present nameCollector, a lightweight tool that statically collects all programmer-defined identifier names in a software system. nameCollector is built using the srcML infrastructure [1, 2]. The srcML format is an XML representation that embeds abstract syntax information with the source code. For each identifier name, nameCollector also extracts the data type (i.e., statically resolved), position in the file, and syntactic category. The syntactic category is the context of an identifier, describing if it is a function name, class name, local variable name, global, etc. nameCollector currently extracts names from C, C++, C#, Java, and Python (i.e., languages supported by srcML) without the need for the code to be compliable. The tool is also very scalable and works on large systems with reasonable runtimes. Additionally, it can work in conjunction with another tool, Stereocode [3], to collect the

stereotype information for functions and classes. Stereotypes [4–6] are simple abstractions that describe the high-level behavior and roles of functions and classes.

While nameCollector does not solve any new technical problem nor use some novel approach, it does provide a general purpose tool for software engineering researchers to easily collect all identifier names in a file/system. The tool is well tested and thus provides users with high quality output. The architecture for nameCollector is now described along with implementation details. This is followed by a discussion of how to use the tool and various applications. The tool is opensource (GPL3) and freely available for download on GitHub[1], and a Docker container available for easy use.

II. Architecture

nameCollector is written in C++ and leverages the srcML infrastructure [1, 2] to collect identifier names and context. First, the system's source code is converted to the srcML format, an XML representation of the source code that includes abstract syntax information. Then, srcSAX, a SAX2 parser framework for srcML, is leveraged to parse the srcML format. As certain XML elements are encountered, srcSAX issues event callbacks. We overload a number of these events to realize nameCollector's specific task. nameCollector outputs all the programmer-defined identifier names, along with their data type (as statically defined), position in the file, and syntactic category (e.g., class, local variable, function, etc.) as either a text report or a CSV file. Fig. 1 provides an overview of nameCollector's pipeline.

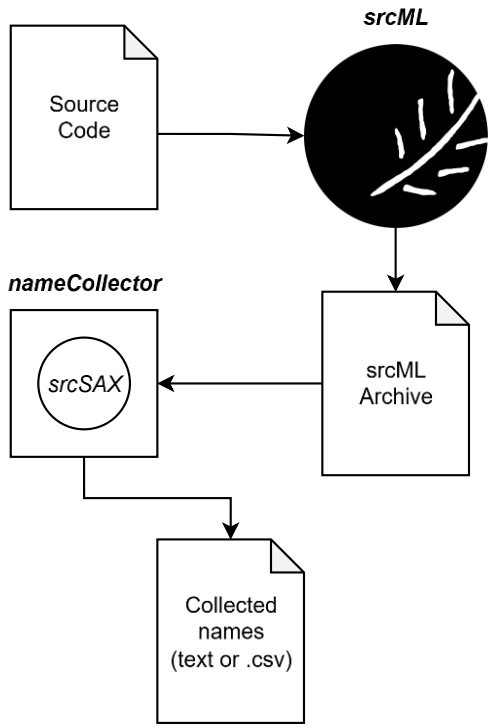

Fig. 1. Overview of identifier extraction using nameCollector

[1] https://github.com/srcML/nameCollector

A. srcML Infrastructure

srcML (www.srcML.org) is an infrastructure supporting source code analysis, exploration, and manipulation. srcML is opensource (GPL3) and freely available on GitHub (github.com/srcML). The srcML format is an XML representation of the source code that embeds abstract syntax information within the source code. This allows for XML parsing and transformation tools, such as XPath, XSLT, RelaxNG, and SAX2, to be used to query and manipulate the source code. srcSAX (also available on github.com/srcML) is an efficient, event-driven API for processing srcML. It has similar characteristics of SAX2 (Simple API for XML, version 2) in that it is low-memory and highly scalable to large srcML documents.

TABLE I. Syntactic Categories of Identifiers Collected by nameCollector over the Languages Supported.

Syntactic Category	Description	Language
Function	Function and function declaration names	All
Constructor	Constructor names	C++, C#, Java
Destructor	Destructor and finalizer names	C++, C#
Class	Class names	C++, C#, Java, Python
Interface	Interface names	C#, Java
Struct	Struct names	C, C++, C#
Union	Union names	C, C++
Enum	Enum names	C, C++, C#, Java
Typedef	Typedef names	C, C++, Python
Label	Lable names (for goto)	C, C++
Field	Names of class, struct, union, interface field	All
Local	Local variable names (function scope)	All
Global	Global variable names	All
Macro	Macro names	C, C++
Namespace	Namespace names and import aliases	C++, C#, Python
Parameter	Function parameter names	All
Function-parameter	Parameter names of function pointers	C, C++
Template-parameter	Template parameter names	C++
Property	Property names	C#
Event	Event names	C#
Annotation	Annotation names	Java

B. Extracting Identifiers and Context

nameCollector uses srcSAX to parse source code in the srcML format. A single srcML document can contain one or multiple source code files in the srcML format. Each file is represented as a `unit` element that specifies the original filename and language. Multiple files are wrapped in a single `unit`, called an archive. This allows a single document to contain source code from an entire system, potentially written in multiple languages (e.g., C++ and Python). At a high level, nameCollector collects names from the following syntactic categories: functions (free and member functions), variables

(fields, locals, parameters, etc.), and programmer-defined data types (classes, structs, etc.). The full list of categories and their descriptions are described in Table I. Of note, nameCollector only collects names where they are declared, and not when they are used.

When the start tag of an element is encountered, nameCollector checks if the element is one of the syntactic categories. If it is, nameCollector adds the element to a stack of open elements. nameCollector also maintains a stack of type elements to collect the data type for each identifier (type information is not collected for Python). It continues parsing until reaching a name tag. If the open element is a type, the name is recorded as the data type of the identifier currently being processed. Otherwise, the name element is part of the identifier's name. Scoped identifiers, such as `String::split()`, shown in Fig. 3, have only the local name collected. The namespace or scope is collected separately when/if it is declared.

When a name is encountered, nameCollector begins saving the content of the element. Upon reaching the end tag of a name, nameCollector stops collecting text and checks the element stack to find the most recently opened element. This is done to ensure the correct syntactic category is applied to the identifier. For example, a variable declared at the top level of a class is a field. However, a variable declared in a function that is within a class is a local variable. nameCollector immediately outputs an identifier once it is collected to minimize memory usage.

Collecting identifiers from source code written in Python adds additional challenges. While functions and classes have explicit syntax that denotes their declaration (`def`, `class`), variables do not have specific syntax for their declaration like the C family of languages. In Python, assigning a value to an identifier for the first time is a variable's declaration. Additionally, Python does not have explicit syntax for specifying the type of a variable. Type hints exist to be used as annotations for IDE plugins and to improve readability, but do not affect behavior by default. Therefore, Python does not have type information collected for variable and function identifiers. To collect Python variable identifiers, nameCollector will process the uses of names in expressions and collect the first occurrence of a name as an L-value. An additional problem in Python name collecting is the scoping and shadowing of variables. Traditional statements such as if- and for-statements, which have their own scope in C-Family languages, do not have their own scope in Python. This means a variable that is first assigned within such a statement is available outside of that statement. When collecting Python names, nameCollector keeps a stack of variable scopes in order to accurately collect the correct first-assignment of an identifier.

nameCollector provides two output formats. The default output format is a text report that provides a sentence describing each identifier collected. The other output format is a comma-separated values (CSV) report. The text report is designed to be human-readable for manual review by users, testing, and can also be used for LLM/Generative AI toolchains. The CSV report allows for nameCollector's output to be used as input to other tools. Fig. 3 provides an example C++ class with the text output generated by nameCollector.

C. Testing

As a part of nameCollector's development, we developed a robust test suite to verify nameCollector's accuracy. The test suite is designed to verify that nameCollector handles all syntactic variations. For example, in C and C++, the keyword `struct` can be used in multiple contexts. Fig. 2 shows the different variations for the `struct` keyword. On line 1, a struct named `Circle` is defined. The name `Circle` is collected as the name of a struct. On line 3, an anonymous struct is defined, and aliased as `Square`. nameCollector does not collect anything for this struct, as there is no name associated with the struct itself, but it will collect the alias `Square` as a typedef name. Line 5 defines a struct named `Rectangle`, as well as two instances of `Rectangle`, `rect1` and `rect2`. The identifier `Rectangle` is collected as the name of the struct, while `rect1` and `rect2` are global variables with type `Rectangle`. This demonstrates how there are many variations of common syntactic features, and statically identifying the identifier names for them all is non-trivial. The test suite is invaluable in identifying edge cases in source code syntax and supports future additions to the tool.

nameCollector is readily scalable and compares to the time of compilation, thus very usable for large systems. We demonstrate this by collecting names from the entire Linux kernel release 7.0.9. This version consists of 63,446 source code files, the majority of which (63,085), are written in C. The rest are Python and C++. In total, 12,263,095 names are collected in 425 seconds (~7 minutes), which is approximately 28,800 names per second. Timing data is collected on an ASUS laptop with a 4.7 GHz processor.

```
1.  struct Circle {/* ... */};
2.
3.  typedef struct {/* ... */} Square;
4.
5.  struct Rectangle {/* ... */} rect1, rect2;
```

Fig. 2. Syntactic variations of the struct keyword and declared names.

III. How to Use nameCollector

nameCollector is a command-line tool that builds and runs on Linux or Mac. A Windows build is future work. To use nameCollector, srcML must be first installed. srcML has installers available for multiple operating systems on www.srcML.org. nameCollector also depends on srcSAX, which needs to be built from source, and is available on GitHub (github.com/srcML). nameCollector must also be built from source (CMake), and is available on GitHub as well. nameCollector supports a srcML document with a single or multiple source code file(s) as input. The srcML client is used to convert source code to srcML. It is recommended that the srcML client outputs the position data, but names can still be collected without it. nameCollector produces a warning if position data is not available. Below is an example of how a system can be converted to the srcML format and run through nameCollector:

```
srcml --position linux/ -o linux.xml
nameCollector -i linux.xml -o linux.csv --csv
```

```
1.  constexpr int MAX_LENGTH = 256;
2.  #define NULL '\0'
3.  typedef std::vector<String> StringVec;
4.
5.  class String {
6.  public:
7.      String() {
8.          str[0] = NULL;
9.      }
10.
11.     String(const char* c_str) {
12.         int len = strlen(c_str);
13.         for (int i = 0; i <= len; ++i) {
14.             str[i] = c_str[i];
15.         }
16.     }
17.
18.     const char* to_c_str() { return str; }
19.
20.     void append(const String& s) {
21.         int start = length();
22.         int end = start + s.length();
23.         for (int i = start; i <= end; ++i) {
24.             str[i] = s.str[i - start];
25.         }
26.     }
27.
28.     String& operator+=(const String& rhs) {
29.         append(rhs);
30.         return *this;
31.     }
32.
33.     int length() const { return strlen(str); }
34.
35.     StringVec split(char c) const;
36. private:
37.     char str[MAX_LENGTH];
38. };
39.
40. StringVec String::split(char c) const {
41.     StringVec parts;
42.     int start = 0;
43.     int end = start;
44.     while (end != length()) {
45.         if (str[end] == c) {
46.             char* s = new char[end - start + 1];
47.             int i;
48.             // ...
49.         }
50.         ++end;
51.     }
52.     return parts;
53. }
54.
55. String concat(const String& s1, const String& s2) {
56.     String result = s1;
57.     result.append(s2);
58.     return result;
```

```
MAX_LENGTH is a constexpr int global in C++ file:
String.cpp:1:15
NULL is a macro in C++ file: String.cpp:2:9
StringVec is a std::vector<String> typedef in C++ file:
String.cpp:3:29
String is a class in C++ file: String.cpp:5:7
String is a constructor in C++ file: String.cpp:7:5
String is a constructor in C++ file: String.cpp:11:5
c_str is a const char* parameter in C++ file:
String.cpp:11:24
len is a int local in C++ file: String.cpp:12:13
i is a int local in C++ file: String.cpp:13:18
to_c_str is a const char* function in C++ file:
String.cpp:18:17
append is a void function in C++ file: String.cpp:20:10
s is a const String& parameter in C++ file:
String.cpp:20:31
start is a int local in C++ file: String.cpp:21:13
end is a int local in C++ file: String.cpp:22:13
i is a int local in C++ file: String.cpp:23:18
operator+= is a String& function in C++ file:
String.cpp:28:13
rhs is a const String& parameter in C++ file:
String.cpp:28:38
length is a int function in C++ file: String.cpp:33:9
split is a StringVec function in C++ file:
String.cpp:35:15
c is a char parameter in C++ file: String.cpp:35:26
str is a char field in C++ file: String.cpp:37:10
split is a StringVec function in C++ file:
String.cpp:40:19
c is a char parameter in C++ file: String.cpp:40:30
parts is a StringVec local in C++ file: String.cpp:41:15
start is a int local in C++ file: String.cpp:42:9
end is a int local in C++ file: String.cpp:43:9
s is a char* local in C++ file: String.cpp:46:19
i is a int local in C++ file: String.cpp:47:17
concat is a String function in C++ file: String.cpp:55:8
s1 is a const String& parameter in C++ file:
String.cpp:55:29
s2 is a const String& parameter in C++ file:
String.cpp:55:47
result is a String local in C++ file: String.cpp:56:12
```

Fig. 3. The left shows a String class, in C++, with identifier declarations highlighted and bolded. Right shows the text output from nameCollector.

The `--position` flag indicates that position data is recorded in the resulting srcML archive. The `-i` flag is used to specify the input file, and the `-o` flag specifies the output file name. The `--csv` flag tells nameCollector to output the results as CSV rather than the default text report. Additionally, nameCollector accepts piped input from the standard input stream:

```
srcml --position linux/ | nameCollector -o linux.csv --csv
```

nameCollector also supports the collection of stereotypes for functions and classes. Stereotypes provide a high-level overview of a function or class's role and responsibility in a system. See [4] and [5] for a description of function stereotypes and [6] for class stereotypes. To collect stereotypes with nameCollector, the stereotypes must first be determined and added to the input srcML archive as an attribute in the srcML.

To do this, Stereocode [3] can be run on the srcML archive, automatically identifying and embedding this information in the archive. Running nameCollector on this new srcML archive allows for the collection of stereotype information along with the identifier names. nameCollector can be extended to collect other custom attributes for future investigations.

IV. Applications

nameCollector is most typically used to support downstream analysis of identifiers of software systems. One of the advantages of collecting the syntactic category for each name is that researchers can easily analyze specific kinds of names. Thus, all local variables or type names can be examined together. This is very convenient for investigating how developers use various naming styles for each category of identifier. The type of the identifier is also very useful. For example, if a local variable is Boolean, one may use a different naming convention than if it is an integer. This provides the researcher with a more complete picture of the identifier's context within the program.

We are currently using nameCollector as part of a sophisticated naming analyzer for freshman computer science programming courses. The file location information, generated by nameCollector, provides the means for generating compiler-like warnings about poorly named identifiers to students. For example, "the local variable foo on line 53 does not follow the standards".

nameCollector can be used for supporting compliance with naming standards during code reviews. Many organizations have guidelines/style guides for identifier names that may be language-specific. Adherence to these guidelines are one aspect of code reviews. Opensource projects also document naming and style conventions for contributors. nameCollector can be used to collect names added to a system to ensure they comply with the project's standards.

V. Conclusion and Future Work

We present nameCollector, a scalable, automated tool for the collection of programmer-defined identifier names in software systems. nameCollector is implemented using the srcML infrastructure and srcSAX, offering it the ability to collect identifiers from C, C++, Java, C#, and Python source code. In addition to extracting names, it also includes data type, position, and syntactic categories along with each identifier. It can be used in conjunction with Stereocode to collect stereotype information with identifiers as well. The scalability of nameCollector is demonstrated by extracting approximately 28,800 names per second from the Linux kernel.

One limitation of nameCollector is that it depends on srcML for input. This means that nameCollector can only parse the languages currently supported by srcML. This does mean, however, that nameCollector will handle new languages that are added to srcML with minimal changes. This is demonstrated by the recent addition of Python to srcML. Existing syntactic categories, such as functions and classes, are immediately supported since srcML provides a unified AST across languages. To support Python, changes only had to be made to collect variable identifiers since Python does not have traditional declaration statements.

Future work for nameCollector includes adding support for collecting name changes. The srcDiff [12] format is an extension of the srcML format that includes syntactic differencing information along with the original srcML. srcDiff provides fine-grained delta identification, meaning that a name change only impacts the name element. This makes it possible to efficiently detect and report identifiers that have undergone a name change using nameCollector. Another modification is the addition of multi-threading. Currently, nameCollector is only single-threaded, thus its performance scales only with CPU clock speed. Multi-threading can improve performance.

Acknowledgment

This work was supported in part by a grant from the US National Science Foundation: CNS 20-16465/16452.

References

[1] M. L. Collard, M. J. Decker, and J. I. Maletic, "Lightweight Transformation and Fact Extraction with the srcML Toolkit," in *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, 2011, pp. 173–184.

[2] M. L. Collard, M. J. Decker, and J. I. Maletic, "srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration," in *2013 IEEE International Conference on Software Maintenance*, 2013, pp. 516–519.

[3] A. F. Al-Ramadan, J. A. C. Behler, M. J. Decker, N. Dragan, M. L. Collard, and J. I. Maletic, "Stereocode: A Tool for Automatic Identification of Method and Class Stereotypes for Software Systems," in *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2024, pp. 898–902.

[4] N. Dragan, M. Collard, and J. Maletic, "Reverse Engineering Method Stereotypes," in *2006 22nd IEEE International Conference on Software Maintenance*, 2006, pp. 24–34.

[5] N. Dragan, M. L. Collard, and J. I. Maletic, "Using Method Stereotype Distribution as a Signature Descriptor for Software Systems," in *25th IEEE International Conference on Software Maintenance (ICSM'09)*, 2009, pp. 567–570.

[6] N. Dragan, M. L. Collard, and J. I. Maletic, "Automatic Identification of Class Stereotypes," in *IEEE International Conference on Software Maintenance (ICSM'10)*, 2010, pp. 1–10.

[7] S. L. Abebe, S. Haiduc, A. Marcus, P. Tonella, and G. Antoniol, "Analyzing the Evolution of the Source Code Vocabulary," in *Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on*, 2009, pp. 189–198.

[8] C. D. Newman, R. S. AlSuhaibani, M. L. Collard, and J. I. Maletic, "Lexical Categories for Source Code Identifiers," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 228–239.

[9] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Relating identifier naming flaws and code quality: An empirical study," in *2009 16th working conference on reverse engineering*, 2009, pp. 31–35.

[10] G. Beniamini, S. Gingichashvili, A. K. Orbach, and D. G. Feitelson, "Meaningful Identifier Names: The Case of Single-Letter Variables," in *Proceedings of the 25th International Conference on Program Comprehension*, 2017, pp. 45–54.

[11] G. Malpohl, J. J. Hunt, and W. F. Tichy, "Renaming Detection," presented at the 15th IEEE International Conference on Automated Software Engineering (ASE '00), 2000, pp. 73–80.

[12] M. J. Decker, M. L. Collard, L. G. Volkert, and J. I. Maletic, "srcDiff: A Syntactic Differencing Approach to Improve the Understandability of Deltas," *J. Softw. Evol. Process*, vol. 32, no. 4, p. e2226, Apr. 2020.