Searching Source Code Using Code Patterns

Ken Nakayama
Tsuda College
2-1-1 Tsuda-machi, Kodaira-shi, Tokyo, Japan
e-mail: ken@tsuda.ac.jp

Eko Sakai
Otani University
Koyama-Kamifusacho, Kita-ku, Kyoto, Japan
e-mail: echo@res.otani.ac.jp

***Abstract*— Understanding source code is crucial for a software engineer. To efficiently grasp the semantics of source code, an experienced engineer recognizes semantic chunks and relations (code patterns) in source code as clues. If a rich repository of searchable code patterns, together with human understandable meanings, is available, comprehension of unfamiliar source code would be easier. However, explicitly defining a source code query even for a simple code pattern can be prohibitively complex for human. In this paper, a tool for search-by-example through abstract syntax tree is presented. A programmer gives sets of desired and undesired nodes, then the system presents some candidate nodes resembling desired ones. This kind of implicit definition by examples is suitable for constructing and revising a repository socially. The method is supervised incremental learning of decision trees. The proposed system uses a set of primitive attributes to reflect domain-specific knowledge safely and easily.**

Keywords—source code search; search by example; abstract syntax tree

I. INTRODUCTION

Observing common programming habit, an engineer seems to locate and loosely combine apparent semantic chunks of various granularity to understand source code. We call such an apparent semantic chunk a code pattern. Although formal correctness of this coarse understanding is not guaranteed, this often successfully outlines the structure of the source code, and thus it becomes good guidance to the engineer. Rich knowledge of code patterns is one of the primary sources of the strength of an experienced software engineer.

If code patterns in source code can be explicitly given by the author or other person who understands the code before, or automatically searched by a tool, understanding unfamiliar source code would be easier. Unfortunately, neither of them is realized. As for the former, comment lines embedded in source code are too ambiguous for automatic processing. As for the latter, defining syntactic search pattern for a code pattern would be too complicated for human, since a code pattern may appear with many minor variations.

We believe that software engineering community should share and socially evolve the "dictionary for code understanding" in the form of queries through source code. This is like a searchable code snippet. Contributors are expected to add and refine these searchable snippets together with description in natural language and example codes.

In this paper, a define-by-example framework for a code pattern is presented based on [1]. In the framework, each instance of a code pattern is represented as an anchored abstract syntax tree (AST). An anchored AST is a tuple

$$(t_1, t_2, \cdots, t_n; T) \tag{1}$$

where $(t_1, t_2, \cdots, t_n)$ is a tuple of AST nodes, or anchors, and T is the enclosing AST, say a `class`. A tuple of anchors provides a syntactically clear (i.e., tightly coupled with source code) chunk for semantic annotaion, while T indicates the maximum extent (context) of the interest in which possible relationships among $(t_1, t_2, \cdots, t_n)$ are searched for. The number of anchors n may be arbitrarily chosen by a user. A prototype system is presented, too.

In the next section, related work is presented. In Section III, a motivating example of the code pattern is presented. In Section IV, anchored abstract syntax tree is propsed as a means of representing instances of a code pattern, then a system which infers a candidate definition of a code pattern from user specified code pattern examples is presented. Section V outlines the algorithm for inferring a code pattern definition. Some experience with our prototype system is in Section VI, the conclusion in Section VII.

II. RELATED WORK

Comprehension of computer programs by human programmer is one of the most important activities in software engineering. There are a wide variety of researches in the literature. Dynamic analyses [2] try to understand runnable programs, while static approaches analyse source codes without running them. The essential difficulty of program comprehension exists in the broad gap of complexity between human and software. When coding a program, abstraction mechanisms such as libraries, code fragments (code snippets), or design pattern help a programmer. However, when reading a code in practice, a programmer can rely only on comments in source code and documents. Feature (concept) location systems may suggest possible locations of interest in source code, but a programmer still have to read the code line by line. Design pattern detection systems may report the meaning of the located code, but their targets are small number of well-known design patterns.

When searching through source code, conventional string search tools are not suitable, because they do not assume the syntax of a programming language. Some tools search source code based on its structure to overcome this limitation. For example, `sourcerer`[3] is a search engine for open source code. `sourcerer` allows a user to ask about implementation and program structure. They compare some ranking methods for the search result ordering.

A brief history of software reuse is presented in [4]. Signature matching [5] tries to search a function using types of its parameters and the return value as a query. There are extensions, such as, [6] using formal semantics, [7] using a specification language, [8] using contract-based specification

Copyright (c) IARIA, 2014.

matching, or [9] using a type system for specification description. All of these approaches are based on “static” information which is extracted from source code or specification without running it.

Another way of charactrising a function is specifying a set of expected (input, output) pairs as a query. Each candidate function is run against these inputs to check whether the output becomes the same with the specified ones. This is “dynamic” characteristics of a function. Researches based on this method include [10], [11], and [12]. Generally, however, signature or type matching alone seldom gives satisfactory search result.

While above approaches aim at reusing functions or methods, other techniques such as [13] or [14] are intended for reusing code fragments.

III. Code Pattern

As an example of code pattern, we will compare some implementations of memoization in Java language. Memoization [15] is a technique to avoid redundant computation of a referentially transparent function, at the cost of storing and recalling already computed results. Referential transparency requires a function to always return the same result for the same given arguments. Memoization is effective when computationally expensive function is called repeatedly. In the design pattern, the flyweight pattern is essentially similar with this technique.

The code in Fig. 1 illustrates a typical memoization. The method `get(P p)` returns the value of function `calc(P p)` either by actually calculating it or by recalling it from already calculated table. For this purpose, this method has a table `Map<P, V> values` which is used to keep already calculated values. If `get(P p)` is called with `p` for the first time, the value `calc(P p)` is added into the table with `p` as its key. If `get(P p)` is called again, this method returns the result by retrieving it from the table without calculating it again.

Although memoization is a quite simple technique, it may be implemented in various ways like in Fig. 2. Some characterizations of memoization should be given to a system, but it is practically impossible for a user to define a search pattern which matches all of these code patterns selectively.

```
abstract class Memoize<P, V> {

    Map<P, V> values = new HashMap<P, V>();

    public V get(P p) {
        if (! values.containsKey(p)) {
            values.put(p, calc(p));
        }
        return values.get(p);
    }

    public abstract V calc(P p);

}
```

Fig. 1: Example code of memoization in Java language.

```
if (! values.containsKey(p)) {      // Code A
  values.put(p, calc(p));
}
return values.get(p);
```

```
if (tab.containsKey(this.d))        // Code B
    return tab.get(this.d);
else {
    Node newNode = new Node(this.d,
            new P(this.d - 1, this.t),
            new P(this.d + 1, this.t));
    tab.put(this.d, newNode);
    return newNode;
}
```

```
if (memo.get(n) == null) {          // Code C
    memo.put(n,
            memoizedFibonacci(n - 1).
            add(memoizedFibonacci(n - 2)));
}
return memo.get(n);
```

```
Byte cb = (Byte) cache.get(nInt);   // Code D
if (cb == null) {
    byte b = runAlgorithm(n);
    cache.put(nInt, new Byte(b));
    return b;
} else {
    return cb.byteValue();
}
```

```
if (r[x][z] == -1) {                // Code E
    r[x][z] = compute(x, z);
}
return r[x][z];
```

Fig. 2: Examples of the memoization code pattern.

To cope with this difficulty, we have adopted a decompose-and-conquer strategy. Let us focus on the conditional expression of `if` statement as an example. The followings are from Fig. 2 and other codes:

```
if (! values.containsKey(p)) {...
if (tab.containsKey(this.d)) ...
if (built.containsKey(m)) ...
if (value == null) {...
if (memo.get(n) == null) {...
if (solved.get(newVal) != null) {...
if (cb == null) {...
if (rem[x][z] == -1) {...
if (c[i][j] == -1) {...
```

Some common patterns are observed. If the table is implemented as a `Collection` object, a predicate method `containsKey()` is used. If the result type is `int`, the value `-1` seems to represent the absence of the data in the table, while `null` is used for reference types (e.g., objects) after trying to get the value using `get()`.

We expect the code pattern would be something like the followings:

```
_X_.containsKey(_Y_)
_X_.get(_Y_) == null
_X_ = _Y_.get(_Z_); ...; _X_ == null
_X_[_Y_][_Z_] == -1
```

allowing `boolean` negation such as `!` or `!=`. Note that this pseudo notation is only for explanation here.

```
IfStatement     // root ASTNode of this part
  // conditional expression of the if-statement
  // ASTNode for condition expression
  EXPRESSION
    PrefixExpression
      > (Expression) type binding: boolean
      OPERATOR: '!'
      OPERAND
        MethodInvocation
          > (Expression) type binding: boolean
          > method binding:
                  Map<P,V>.containsKey(Object)
          EXPRESSION
            SimpleName
              > (Expression) type binding:
                      java.util.Map<P,V>
              > variable binding:
                      Memoize<P,V>.values
              IDENTIFIER: 'values'
          TYPE_ARGUMENTS (0)
          NAME
            SimpleName
              > (Expression) type binding:
                      boolean
              > method binding:
                  Map<P,V>.containsKey(Object)
              IDENTIFIER: 'containsKey'
          ARGUMENTS (1)
            SimpleName
              > (Expression) type binding: P
              > variable binding: p
              IDENTIFIER: 'p'
  THEN_STATEMENT        // then-part
    Block               // then-part is a block
      STATEMENTS (1)
  ELSE_STATEMENT: null  // no else-part
```

Fig. 3: AST for `if` statement of Fig. 1 (generated using [17] and adapted).

Once this code patter is defined, this can be used as a building block of other code patterns. Of course, these syntactic characteristics do not guarantee that this conditional expression is a constituent element of memoization. Although they just suggest it, the existence of multiple clues gradually increases the confidence of it.

IV. Code Pattern by Example: User Interaction

A. Code Pattern as Anchored AST

Source code that conforms to the syntax specification of the language can be represented as AST. Fig. 3 is an AST which corresponds to the `if` statement in Fig. 1. This AST is generated by Java development tools (JDT)[16] in Eclipse IDE. The root node `IfStatement` represents the whole `if` statement, which has three branches, namely, `EXPRESSION` (condition expression), `THEN_STATEMENT` (then-part), and `ELSE_STATEMENT` (else-part). In this example, `ELSE_STATEMENT` is `null` since the `if` statement has no else-part. Fig. 3 shows just an overview the AST for simplicity.

We represent an instance of a code pattern as an anchored AST. If a user wants to express a code pattern for the conditional expression in the above example, anchored AST's $(t_1; T)$ where t_1 is conditional expression such as

```
values.containsKey(p)
```

or

```
cb == null.
```

As for the latter one, if a user wants to give explicit hint for the constraint on variable `cb`, constraining AST node

```
Byte cb = (Byte) cache.get(nInt);
```

may be specified as an additional anchor t_2, resulting an anchored AST $(t_1, t_2; T)$.

B. User Interaction Model

A user interactively defines a new code pattern by giving positive or negative examples of a code pattern. Each example is an anchored AST. Negative example refers to an anchored AST which is not an instance of the code pattern. Typically, a negative example is obtained when the system mistakenly locate a false positive code pattern. In such a case, a user teaches that it is a negative example.

We will give the overview of this user interaction using our prototype system. The prototype system has been implemented as a plug-in of Eclipse IDE [18]. A plug-in can take advantage of functions for manipulating Java source code provided by Eclipse JDT [16]. Fig. 4 is the screen shot of the prototype system. In the window, a special pane for anchored AST registration and classification is placed on the right-hand-side.

Part (a) through (d) in Fig. 5 are screen snapshots when a user is defining an single anchored AST $(t_1; T)$. A user selects a code region corresponding to the intended AST node (part (a)), then add this either as a positive (part (b)) or as a negative (part (c)) example by clicking appropriate button. Added AST node is displayed in the pane with a character '`P`' or '`N`' indicating positive or negative, respectively.

For each code pattern, the system keeps positive example set S_+ and negative example set S_-. A user can add arbitrary number of positive examples to S_+. Once a user thinks that he has put enough positive/negative examples, the system infers a hypothetical definition of the intended code pattern from S_+ and S_-, the system infers the definition of a code pattern as a supervised discrimination learning on AST.

Then, the sytem searches through source code for a tuple of AST nodes that match the hypothetical definition. If such a tuple is found, it is presented to the user. The user judges whether the found tuple is an instance of the intended code pattern or not, and add it as either positive or negative example to the system. Updated S_+ and S_- are used for the next search. A user continues this interaction until he is satisfied with the inferred code pattern.

V. Classifying Attribute Vectors

A. Projecting Anchored AST onto Attribute Vector

The problem to solve is finding a common pattern in S_+ and not in S_-. Since anchored AST has complex structure to compare directly, we have decided to project an anchored AST onto a vector of attributes. Let us illustrate an attribute vector with a simple example. Suppose that we are interested in a

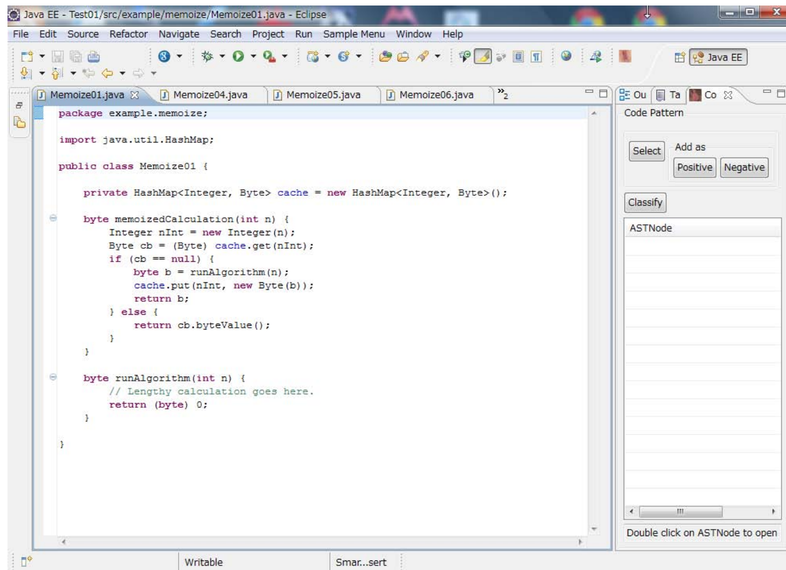

Fig. 4: A screen shot of the prototype system.

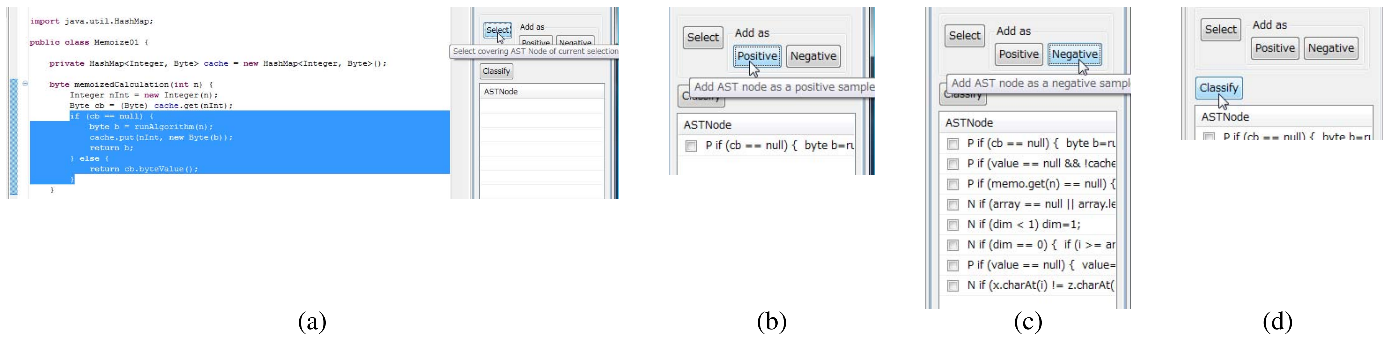

Fig. 5: (a) Select an AST node on source code, add it as a (b) positive or (c) negative example, repeat this to register some examples, then (d) classfy to compose a decision tree.

code pattern represented as anchored AST's with two anchors. That is,

$$S_{+},\ S_{\text{-}} \subseteq U^{(2)},\ S_{+} \cap S_{+} = \emptyset \tag{2}$$

where $U^{(2)} = \{(t_1, t_2; T)\}$ denotes the set of all anchored AST's with two anchors. And suppose that we have choosen a vector of four attribute functions $A = (a_1, a_2, a_3, a_4)$ where

$$a_i : U^{(2)} \mapsto \text{(various attribute value)}\,(i = 1, \cdots, 4) \tag{3}$$

as shown in Table I. Then, if t_1 is the `if`-statement

TABLE I: ATTRIBUTE FUNCTIONS (EXAMPLE).

$a_1(t_1, t_2; T)$	`true` if t_1 is an `if`-statement; `false` otherwise
$a_2(t_1, t_2; T)$	`true` if t_2 is an `if`-statement; `false` otherwise
$a_3(t_1, t_2; T)$	Method name if t_2 is a method invocation; `null` otherwise
$a_4(t_1, t_2; T)$	`true` if t_2 is under t_1 in AST T; `false` otherwise

```
if (! values.containsKey(p)) {
    values.put(p, calc(p));
}
```

and t_2 is "`put(p, calc(p));`" in t_1, $A(t_1, t_2; T)$ would be an attribute vector $(\texttt{true}, \texttt{false}, \texttt{"put"}, \texttt{true})$.

B. Generating Attribute Vector

An attribute function vector A is generated by recursively combining predefined primitive attribute functions. Table II is an excerpt of the list of primitive attribute functions used in the current prototype system. There are primitive functions too that have multiple inputs not shown in this table.

`ASTNode` is a class type which represents AST node in JDT. Current implementation of primitive attribute function returns `null` when its imput(s) are illegal or nonsense. For instance, `paIfThenStatement` returns the AST node of then-part if the input is `if`-statement and then-part exists, but it returns `null` otherwise.

An attribute function vector A for an anchored AST with one anchor $U^{(1)}$ is shown in Fig. 6. Line `0` is the given AST node t_1. After that line, generated attributes follow. For example, line `1` means that this at-

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-354-4

TABLE II: PRIMITIVE ATTRIBUTE FUNCTIONS (EXAMPLE).

Function name	Input type	Output type
paAstNodeType	(ASTNode)	Class
paIfExpression	(ASTNode)	Expression
paIfThenStatement	(ASTNode)	Statement
paBlockStatements	(ASTNode)	List<Statement>
⋮	⋮	⋮

tribute is defined as application of primitive attribute function `paAstNodeType()` to the value of line `0`, that is t , and the type of value is `java.lang.Class` . Similarly, line `2` uses another primitive attribute function `paIfExpression()` , which returns another AST node of class `org.eclipse.jdt.core.dom.Expression` (subclass of ASTNode). If line `0` is not an instance of `if`-statement, the value would be `null`. The value type of line `2` is obtaind as an attribute at line `11`.

By applying generated attribute function vector A to an anchored AST, an attribute value vector is obtained. Fig. 7 shows an example of attribute value vector. This is logically single vector (wrapped to fit the page width). `NA` denotes the value which is not used for constructing a decision tree. For example, if a value is an instance of `ASTNode`, it is used as an input of other attributes, but is not directly used as an attribute for classification.

C. Discrimination by Constructing Decision Tree

From attribute value vectors projected from S_+ and S_-, a decision tree is constructed for classification. A decision tree is a tree-structured cascade of decisions in which each node represents a decision on a certain attribute at a time. Current prototype system uses J48 in [19]. Fig. 8 is an example of constructed decision tree. `N0`... are nodes, and `N0->N1`... are transitions labeled with a condition. For example, At the root node `N0`, primitive attribute functions `paIfExpression()`, `paInfixExpressionRightOperand()`, and `paAstNodeType()` are applied to the given `ASTNode` (`param0`) in this order. `paIfExpression()` assumes the given `ASTNode` is an `if`-statement, and returns its condition expression. Then, `paInfixExpressionRightOperand()` assumes the condition is infix expression, and returns its right-hand-side operand. Finally, `paAstNodeType()` extracts its node type. Depending on the type, one of `N0->N1`, `N0->N2`, ..., `N0->N5` is chosen. If the decision flow reaches `N1`, that indicates that the given tuple is classified as `positive`.

VI. Experience with Prototype System

A. Searching a Tuple Through Source Code

A decision tree can classify a given tuple of `ASTNode`. Current prototype system does not have efficient search mechanism yet. It uses brute-force exhaustive search by enumerating all combination of `ASTnode`'s in the given source code. This works when applied to small tuple and source code, improvement is necessary.

B. Current Set of Primitive Attribute Functions

The primitive attribute functions currently implemented in our prototype system are in two types, namely, (1) function which gives some value describing a feature of the given `ASTnode`, (2) function which traverses to another `ASTnode` from the given `ASTnode`.

Since most of type (2) follow parent-to-child relationship in AST, structurally similar code pattern can be defined. However, anchored AST's that have different structures to each other are recognized as different ones. For instance, `x = 3;` and `{ x = 3; }` are semantically equivalent, but the existence of an extra block prevents the intended recognition. In the same way, an operator such as equivalence operator == is commutative (if there is no side effect), but current system cannot generalize `x == null` and `null == x`. Rather, the system just enumerate these two patterns as distinct ones.

Another example is `ASTnode` traverse by following data flow or control flow. When the AST node at `if`-statement is given as an anchor in the next code, the variable `cb` should be treated as if it is `cache.get(nInt)`.

```
Byte cb = (Byte) cache.get(nInt);
if (cb == null) {
```

This can be realized if a primitive function exists which traverse AST following data flow backward.

To improve such situations, more sophisticated primitive functions should be introduced.

VII. Conclusion

A framework for incremental definition of a code pattern has been presened. In the framework, a code pattern is represented as an anchored AST by a user. An anchored AST is then projected onto an attribute vector for classification using a decision tree. An attribute function vector is generated by recursively combining primitive attribute functions. A prototype system is implemented as a plug-in of Eclipse IDE. Inferred decision tree if good for exact exact search.

Future improvements include enriching primitive attribute functions, reducing redundant or nonsense attributes.

References

[1] K. Nakayama and E. Sakai, "Source code navigation using code patterns," IEICE technical report. Life intelligence and office information systems, vol. IEICE-113, no. 479, mar 2014, pp. 23–28, (In Japanese).

[2] B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," Software Engineering, IEEE Transactions on, vol. 35, no. 5, 2009, pp. 684–702.

[3] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi et al., "Sourcerer: a search engine for open source code supporting structure-based search," in Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications. ACM, 2006, pp. 681–682.

[4] S. P. Reiss, "Semantics-based code search," in Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on. IEEE, 2009, pp. 243–253.

[5] A. M. Zaremski and J. M. Wing, "Signature matching: A key to reuse," vol. 18, no. 5. ACM Press, 1993, pp. 182–190.

[6] E. J. Rollins and J. M. Wing, "Specifications as search keys for software libraries." in ICLP. Citeseer, 1991, pp. 173–187.

```
0: Given parameter: value type class org.eclipse.jdt.core.dom.ASTNode
1: Generated attribute: paAstNodeType(0)-->class java.lang.Class
2: Generated attribute: paIfExpression(0)-->class org.eclipse.jdt.core.dom.Expression
3: Generated attribute: paIfThenStatement(0)-->class org.eclipse.jdt.core.dom.Statement
4: Generated attribute: paIfElseStatement(0)-->class org.eclipse.jdt.core.dom.Statement
5: Generated attribute: paInfixExpressionOperator(0)-->
                    class org.eclipse.jdt.core.dom.InfixExpression$Operator
6: Generated attribute: paInfixExpressionLeftOperand(0)-->class org.eclipse.jdt.core.dom.Expression
7: Generated attribute: paInfixExpressionRightOperand(0)-->class org.eclipse.jdt.core.dom.Expression
8: Generated attribute: paMethodInvocationBoundMethod(0)-->class java.lang.String
9: Generated attribute: paMethodInvocationArguments(0)-->interface java.util.List
10: Generated attribute: paBlockStatements(0)-->interface java.util.List
11: Generated attribute: paAstNodeType(2)-->class java.lang.Class
  (some lines omitted)
20: Generated attribute: paIfExpression(7)-->class org.eclipse.jdt.core.dom.Expression
21: Generated attribute: paIfThenStatement(2)-->class org.eclipse.jdt.core.dom.Statement
  (the rest are omitted)
```

Fig. 6: Combination of primitive attribute functions (example).

```
NA,class org.eclipse.jdt.core.dom.IfStatement,NA,NA,NA,null,null,null,null,null,null,
class org.eclipse.jdt.core.dom.InfixExpression,class org.eclipse.jdt.core.dom.Block,
class org.eclipse.jdt.core.dom.Block,null,null,null,null,null,null,null,null,null,null,null,
null,null,null,null,null,null,"==",null,null,null,null,NA,null,null,null,null,NA,null,null, ...
```

Fig. 7: Attribute values for decision tree construction (example).

```
J48() result:
digraph J48Tree {
N0 [label="paAstNodeType(paInfixExpressionRightOperand(paIfExpression(param0)))" ]
N0->N1 [label="= class org.eclipse.jdt.core.dom.NullLiteral"]
N1 [label="positive (3.0)" ]
N0->N2 [label="= class org.eclipse.jdt.core.dom.NumberLiteral"]
N2 [label="negative (2.0)" ]
N0->N3 [label="= class org.eclipse.jdt.core.dom.MethodInvocation"]
N3 [label="negative (1.0)" ]
N0->N4 [label="= class org.eclipse.jdt.core.dom.InfixExpression"]
N4 [label="negative (1.0)" ]
N0->N5 [label="= class org.eclipse.jdt.core.dom.PrefixExpression"]
N5 [label="positive (1.0)" ]
}
```

Fig. 8: Attribute values for decision tree construction (example).

[7] D. Hemer and P. Lindsay, “Supporting component-based reuse in care,” in Australian Computer Science Communications, vol. 24, no. 1. Australian Computer Society, Inc., 2002, pp. 95–104.

[8] J.-J. Jeng and B. H. Cheng, “Specification matching for software reuse: a foundation,” in ACM SIGSOFT Software Engineering Notes, vol. 20, no. SI. ACM, 1995, pp. 97–105.

[9] C. Runciman and I. Toyn, “Retrieving re-usable software components by polymorphic type,” in Proceedings of the fourth international conference on Functional programming languages and computer architecture. ACM, 1989, pp. 166–173.

[10] S.-C. Chou, J.-Y. Chen, and C.-G. Chung, “A behavior-based classification and retrieval technique for object-oriented specification reuse,” Software: Practice and Experience, vol. 26, no. 7, 1996, pp. 815–832.

[11] R. J. Hall, “Generalized behavior-based retrieval,” in Proceedings of the 15th international conference on Software Engineering. IEEE Computer Society Press, 1993, pp. 371–380.

[12] S. Thummalapenta and T. Xie, “PARSEWeb: a programmer assistant for reusing open source code on the web,” in Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. ACM, 2007, pp. 204–213.

[13] S. Paul and A. Prakash, “A framework for source code search using program patterns,” Software Engineering, IEEE Transactions on, vol. 20, no. 6, 1994, pp. 463–475.

[14] G. Little and R. C. Miller, “Keyword programming in java,” Automated Software Engineering, vol. 16, no. 1, 2009, pp. 37–71.

[15] D. Michie, “Memo functions and machine learning,” Nature, vol. 218, no. 5136, 1968, pp. 19–22.

[16] The Eclipse Foundation, “Eclipse Java development tools (JDT),” http://eclipse.org/jdt/ [retrieved: July, 2014].

[17] ——, “AST View in eclipse Java development tools (JDT),” https://eclipse.org/jdt/ui/astview/index.php [retrieved: July, 2014].

[18] ——, “Eclipse integrated development environment,” http://www.eclipse.org/ [retrieved: July, 2014].

[19] Machine Learning Group at the University of Waikato, “Weka 3: Data Mining Software in Java,” http://www.cs.waikato.ac.nz/~ml/weka/ [retrieved: July, 2014].