ABSTRACT

While migrating existing software systems to Software Product Lines, finding out the functionalities in the software is critical. For maintenance activities like deleting or changing existing features, or adding new similar features, identifying and extracting functionalities from the software is significant. This paper describes a technique for creating mapping between the source code and functionalities implemented by it while exploiting the domain knowledge. The technique is based on the notion of function variables that are used by developers for expressing functionality in the source code. By tracking the known values of the function variables and evaluating the conditions that use them, the mapping is identified. Our technique makes use of static data flow analysis and partial evaluation, and is designed with automation perspective. After applying to few samples representing real-life code structure and programming practices, the technique identified precise mapping of the detailed program elements to functions.

Keywords

Functionality identification, function variables, partial evaluation

1. INTRODUCTION

Software Product Lines (SPL) is a collection of similar software systems constructed from a shared set of assets. Especially for software products, SPL are valued, as they enable increased productivity with reduced time-to-market, provide for lesser cost and high quality [3] by enabling strategic, planned reuse of assets among multiple products.

Legacy applications on the contrary are generally monolithic in nature and hence expensive and time consuming to maintain. Adding new features and fixing bugs consumes lots of resources. If such monolithic applications (especially products) are transformed to SPL aligned architectures, it would help reduce costs to maintain the application, and also reduce the time required to add or update features in the application. Performing such transformations are however non trivial. It requires knowledge about the structure of the software being transformed along with information about the various functionalities that are implemented. Finding mappings between the source code and the implemented functionalities is important for program comprehension and is of big help for a new developer to understand the software, especially while adding new feature or modifying existing feature [19].

Functionality mapping is also useful during defect fixing. With the availability of source code to functionality mapping, the developer can locate the failure point in the source code much faster, thus expediting the defect-fixing process [19].

Performing the migration consists of three main phases, first is detection phase, next is analysis phase and last is transformation phase. In detection phase, we extract information from the source code such information describes the existing code structure and architecture the various functionalities implemented in the source code, how are they structured into layers and library components. In the analysis phase, the information discovered from the previous phase is used to design and create new partitions that primarily segregate the functional features, adhere to layered-code principles and identify explicit library components. We then perform the various transformations on the source code using knowledge from the previous phases. All the transformations are aimed at migrating the monolithic legacy application to the layered, modular, functionally separated source code. Creating the Product Line architecture requires more analysis to identify commonalities and variabilities, which is beyond the scope of this paper.

In the detection phase, the primary requirement is to identify which program elements perform which functionality. This is akin to detecting core assets (functionalities) from existing monolithic system. Our technique to identify the functionality in the source code makes use of the variables that are used by the developers as a representation of domain knowledge. We employ partial evaluation to evaluate conditional checks in the source code that use such variables, and use the result to decide (statically) which program elements are exercised for the given functionality. The work in this paper is the detailing out of one of the critical steps in the application architecture discovery, described in an earlier paper [8].

2. RELATED WORK

Finding the initial location of functionality in the source code is known as Feature Location (FL). Different kinds of techniques have been explored for performing FL. Techniques using static analysis, dynamic analysis, or textual analysis have been tried. Few methods also use combination of these techniques.

Dynamic feature location collects information when system is in execution. Such information generally is in the form of execution traces. Software reconnaissance [20, 10] is one of the oldest dynamic method for FL. The system is executed on two different
scenarios; one where the feature is invoked and other where the feature is not invoked. By comparing the two execution traces, it is possible to identify the source code which got executed for that particular feature. Above method was extended by Eisen-berg [6] which is called as Dynamic Feature Traces [DFT]. Wong et al. [21] proposed a technique which uses execution slicing for performing the feature location. There are numerous other tech-niques which use dynamic analysis. All the dynamic techniques suffer from the drawback that the results depend on the quality of the test scenarios using which traces were collected. They also adversely impact the execution time of the software system. In industry, especially for legacy environments, it is very difficult to get access to execution (test) environment, making it extremely difficult to apply methods based on dynamic analysis.

Some techniques use static analysis. Chen and Rajlich [4] intro-duced a method based on Abstract System Dependence Graphs [ASDG] which is an abstraction of System Dependence Graphs [SDG]. It requires feedback from the user whether the current node (function or global variable) in the graph is relevant to the maintenance task (feature). Based on the choice, the method refines the graph, and the process is continued until the user finds all nodes related to the maintenance task. Robillard and Murphy [14, 13] introduced concept graphs which describe the features in terms of program elements and static relations between them. The user marks the initial program elements that are related to the feature, and the concept graph then identifies more elements that are potentially related to the feature, based on the structural dependencies. All these methods make use of only structural dependencies and do not use data dependencies. Also, they rely on user inputs and their precision depends heavily on the accuracy of the user inputs.

In addition, many other techniques use source code as texts and apply various text processing techniques for locating features. Some methods make simple use of grep [11], while others use complex algorithms of Information Retrieval [5, 1, 9, 12] and NLP [7, 15]. The techniques depend heavily on the appropriate naming of variables and use of comments in the source code. For old legacy applications, this can impose severe limitations on its accuracy.

Our method starts with a set of variables which are called information sinks. The variables are selected by developers and designers de-pending on their significance in the program. They are used as starting points to identify parts of the source code where its propagated values are used. Data flow analysis is employed to propagate the values. In our approach we start with a set of function variables - they are defined as variables which hold values that map to the functionality from the domain (user) perspective. Both approaches share similarity as they start with a predefined set of variables and then track their values for marking the source code. However, the propagation in our approach is based on control-flow due to condition checks that use the function variables. We also perform partial evaluation of the conditions appearing in the code to increase the accuracy of the marking. Partial evaluation allows us to add a flavor of dynamic analysis to the process. As we start with fine grained input (i.e., variables), the results are also finer grained than other feature location techniques.

The rest of the paper is organized as follows. Section 3 describes various ways in which functionality is represented in the source code. Section 4 explains the partial evaluation and expression evaluation algorithm. In Section 5, the high level algorithm to detect the source code for a feature is explained. Section 6 presents the results with an example. Section 7 describes the approach to automation. Finally, in Section 8, we present concluding remarks.

3. EXPRESSING FUNCTIONALITY IN SOURCE CODE

We have observed different ways in which the developer expresses the functionality in the source code. In business applications, the functionality is usually represented using function variables. The function variables hold values which correspond to various functionalities implemented in the source code. The values are usually discrete values and not range bound values. The values of the function variables represent the functionalities (features) in the domain which allows us to map the program elements to the functionality represented by the value.

The way the developer expresses functionalities in the source code also varies depending on the adopted architecture. In an event-driven architecture (e.g., event may be a click of a button on the screen) a specific subroutine may be called on the occurrence of the event. Function will have a predefined signature like ‘void action Performed (ActionEvent e)’. In this case there are no function variables.

In a MVC request-response based architecture, the developer specifies the subroutine (method) to be invoked for a particular action, through a configuration file. For example, in a Strut based ap-plication, the developer specifies the action class to be used for a screen. The action class has an execute method to implement the functionality. The action class and corresponding execute method is used for expressing functionality. In business applications written using the RPG programming language on iSeries platform (fairly popular in some European financial institutions), we have observed that the menu processing - including display of the menu and processing user input - is driven through a special menu pro-gram. A configuration file is used by the menu program to call appropriate functions for user requests. Thus, this configuration file gives initial mapping of the functionality to the source code.

In our observation, a large number of legacy systems make use of the functional variable paradigm. We intend to target such software systems that make use of functional variables. Based on the observations in multiple business applications, we assume that the values of the function variables are discrete values and are not range bound. In addition to function variables, we also make use of variables which take values depending on the value of function variable. Such variables are termed as derived variables. Derived variables indirectly represent domain functionality and hence are used for mapping functionality. A sample COBOL program below shows the function and derived variables

**Listing 1: Sample Program**

```plaintext
PROCEDURE DIVISION.
MAIN-PARA.
MOVE "ADD" TO FuncVAR1.
MOVE "UPDATE" TO FuncVAR2.
IF FuncVAR1 = ADD
   MOVE 10 TO DERVAR1
ELSE
   MOVE 100 TO DERVAR1
END-IF.
IF FuncVAR2 = UPDATE
   MOVE 50 TO DERVAR2
ELSE
   MOVE 500 TO DERVAR2
END-IF.
```

Our technique is closest to that proposed by Trifu [17, 18]. His approach consists of a set of variables which are called information sinks. The variables are selected by developers and designers depending on their significance in the program. They are used as starting points to identify parts of the source code where its propagated values are used. Data flow analysis is employed to propagate the values. In our approach we start with a set of function variables - they are defined as variables which hold values that map to the functionality from the domain (user) perspective. Both approaches share similarity as they start with a predefined set of variables and then track their values for marking the source code. However, the propagation in our approach is based on control-flow due to condition checks that use the function variables. We also perform partial evaluation of the conditions appearing in the code to increase the accuracy of the marking. Partial evaluation allows us to add a flavor of dynamic analysis to the process. As we start with fine grained input (i.e., variables), the results are also finer grained than other feature location techniques.
In the sample program 1, FUNCVAR1 and FUNCVAR2 are the two function variables which are known upfront along with the values they can take in the program. Here, FUNCVAR1 can take value either ADD or DELETE while FUNCVAR2 can take value either UPDATE or COMMIT. These values directly tell us about the functionality implemented in the source code. The two variables DERVAR1 and DERVAR2 are derived variables, since they take values depending on some value of the function variable. Hence, those values of the derived variables can be mapped to the corresponding values of the function variables and in turn to the domain functionality. These values of the function variables to which derived variable values are mapped are known as context of the derived variable. Once this mapping is established we can use derived variables for marking the source code.

3.1 FINDING DERIVED VARIABLES
The number of additional variables that take values depending upon the value of function variables can be very large and hence tracking all of them may be expensive. Ideally, we are interested in the variables which occur in some conditional expression. Following pseudo-code represents the algorithm for finding derived variables.

```
Listing 2: Pseudo-Code for finding derived variables
HashSet findDerivedVariables() {
    HashSet influencedVars = null;
    HashSet controllingVars = null;
    HashSet derivedVars = null;
    influencedVars = getAllInfluencedVars();
    controllingVars = getAllControllingVars();
    derivedVars = influencedVars INTERSECTION controllingVars;
    return derivedVar;
}
```

In the above pseudo-code 2, the subroutine `getAllInfluencedVars()` identifies all the variables that take values from any function variable, while `getAllControllingVars()` identifies all the variables that control some flow paths in the code. The Derived variables are the ones present in both the sets.

4. PARTIAL EVALUATION
Partial evaluation is a technique of partially executing a program with given or available input. Suppose there is a program $P_1$ which requires two inputs, $m1$ and $m2$ for computing the desired output. When only one of the inputs, say $m1$, is available, the program cannot compute the output, but can be partially evaluated with input $m1$ to get another program, say $P_1'$, which is a specialized version of the original program $P_1$. Partial evaluation is a program optimization technique, also known as program specialization, and $P_1'$ is known as a residual program.

Consider following example of a function for finding the power of a number.

```
Listing 3: An Example code
double power(n, x) int n;
double x;
{
double p = 1.0;
while (n > 0)
{
    if (n%2 == 0)
    {
        x = x * x;
        n = n / 2;
    }
    else
    {
        p = p * x;
        n = n - 1;
    }
}
return p;
}
```

Suppose we know that value of n = 5 and we need to find the value of function `power(5, x)`. We can now partially evaluate the `power()` function to obtain the following residual function.

```
Listing 4: Residual Function
double power(5, x) double x;
{
double p;
p = 1.0 * x;
x = x * x;
x = x * x;
p = p * x;
return p;
}
```

We use a derivative of such partial evaluation technique while marking the program elements with functionality. While performing the marking, we try to evaluate the condition expressions present in the decision blocks with the values of the variables that are found using constant propagation.

In a program, the decisions are made at the decision blocks, namely, IF, WHILE, SWITCH, and others. The evaluation of the conditions decides which path is taken and which block executed. If we can find the values of the variables involved in the conditional expression, then we can perform complete or partial evaluation of the condition expression and evaluate whether condition would be true or false. Having evaluated the condition, we can proceed to mark the program elements in the two or more paths appropriately based on the inferred value of the function variable.

4.1 Constant Propagation
For evaluating the conditional, the values of the variables in the condition expression must be known. For finding the values of these variables, we use constant propagation, which is a well known global data flow analysis. Constant propagation finds the variables which have constant values at a given program point, along all the execution paths reaching that point. The values are then propagated in the program. We make use of a variant of the constant propagation. Instead of having the assertion of the variable having constant value along all the paths which would be a very stringent condition, we collect all the variables having constant values along at least one path. By relaxing this condition of having constant value along all paths we increase our set of values which we use for the evaluation. However, this increases the imprecision of the result set. Results of constant propagation are used by the expression evaluator for evaluating the condition.

4.2 Expression Evaluation
Expression evaluation performs the job of evaluation of conditions that are encountered in the decision blocks of the source program. It returns the set of values that satisfies the conditional expression, while it uses data from constant propagation data-flow analysis. For a given condition, the expression evaluator first uses the result of constant propagation which lists the variables that have got constant values at that program point. With the constant values and knowledge of the values that the function variables can take, the evaluator tries to evaluate the condition. On successful evaluation, it returns a multiple value set, each consisting of one satisfying assignment to the condition. For unsuccessful evaluation, the evaluator will return a special status implying so. The results of the evaluator are used for marking the program elements. Below we provide the pseudo code of the algorithm for expression evaluator.

Listing 5: Expression Evaluation pseudo-code

```java
List evalElemExpr(expression)
{
    opr = get operator from expression;
    lhs = get lhs from expression;
    rhs = get rhs from expression;
    lhsVals = get all possible values of lhs;
    rhsVals = get all possible values of rhs;
    for each lVal in lhsVals
    {
        for each rVal in rhsVals
        {
            tValue = checkValues(lhs, rVal, opr);
        }
    }
    currExp = (lVal && rVal) | (lVal || rVal)
    return currExp;
}
```

The expression evaluator breaks down the expression into several elementary expressions until it encounters an expression which has an equality operator. For each elementary expression, the evaluator obtains the possible values of left hand side variable, right hand side variable and performs the evaluation. The expression evaluator also uses the results of constant propagation, i.e., possible constant values of variables. Evaluator makes use of the values to compute the truth value of the expression. The function checkValue() checks whether lhs and rhs values match. If they match, then that expression evaluates to true. The evaluator accumulates the truth values of the expressions to evaluate the composite expression. It remembers the logical operator involved in the expression so as to create the correct satisfying set.

For e.g., condition expression we want to evaluate is (FV1 == A & FV2 == B). The expression is broken down into two elementary expressions (FV1 == A), (FV2 == B). The individual elementary expressions are then evaluated to find out the truth values which make those expressions true. Since these expressions are connected by & operator, the corresponding values are then ANDed with each other.

5. HIGH-LEVEL ALGORITHM

Figure 1 shows the components which are involved in the technique.

Marking Query: This component performs the marking of program elements. It finds out derived variable values along with their contexts. When it encounters conditions, it invokes the expression evaluator to evaluate them. It uses the result of the expression evaluator to mark the program elements with the values of the function variables.

Function Marker: After the program elements are marked with the values of function variables, the marks need to be replaced by
the functionalities (names) which are represented by the values. This step may require manual intervention.

**Expression Evaluator:** - This component evaluates the conditions at decision points (IF, SWITCH etc) and returns the values that satisfy the given condition. The step makes use of the results from the constant propagation analysis.

**Constant Propagation:** - This component finds possible constant values of the variables at any given program point. It provides data to the Expression Evaluator for evaluating the conditions.

Our technique marks the program elements at the granularity of decision blocks. Every decision block has a condition check, and which block of source code executes for what values of variables in the conditional is decided by the evaluation of the conditions during program execution. The values that the (function) variables take depend on the functionality that is implemented in the source code. Hence, we can map the values of the function variables to the functionalities. When the marking query encounters jump or subroutine invocation, the marking context is continued to the jump point or the subroutine.

In business applications using legacy programming languages, we observed that the functionalities are implemented using the notion of function variables. The values of the function variables are pre-defined constants which are representation of the domain functions in the source code. Consider the following example.

### Listing 6: An Example code

```plaintext
1 IF FVAR1 = ADD
2   MOVE 10 TO DATA-VAR1
3   MOVE 20 TO DATA-VAR2
4   COMPUTE DATA-VAR3 = DATA-VAR1 + DATA-VAR2
5 END-IF
6 ELSE
7   PERFORM A THRU A-EXIT
8 END-IF
9 IF DATA-VAR1 = 10
10   PERFORM C THRU C-EXIT
11 END-IF

Assume that FVAR1 is a function variable which can take discrete values from the set {ADD, SUBTRACT}. This information is available as input. The code block in the THEN branch (line 3 to 6) executes when FVAR1 takes value ADD. The value ADD, therefore, can be mapped to the Addition functionality and we can conclude that the code block from line 3 to 6 executes for Addition functionality. The ELSE branch of IF...THEN executes when FVAR1 != ADD i.e., FVAR1 is SUBTRACT. We, therefore, can mark ELSE branch (line 8) as Subtraction functionality. This describes the overview of our technique.

Before marking the THEN branch we need to make sure that condition FVAR1 = ADD evaluates to TRUE. For this evaluation, we make use of constant propagation flow analysis to find the possible constant values of FVAR1. If ADD is one of the possible values then we conclude that the condition will evaluate to true for some execution. However, if ADD is not one of the possible values then we conclude that THEN branch is dead code and will not execute for any execution of the program.

To improve the precision of the technique, we also use derived variables for performing the marking. In the sample source code shown in listing 6, the variable DATA-VAR1 gets the value 10 from the context of variable FVAR1 having value ADD. We keep track of all such derived variables and their values along with their functional contexts and use them for marking the program elements. When the next decision block with condition DATA-VAR1 = 10 is encountered, it is known that DATA-VAR1 has value 10 and its functional context is Addition. This is used to mark the program element at line 12.

### 5.1 Algorithm

1. Mark all the statements as ALL functionality, this is an initialization step.
2. Find derived variables. Include all the variables which influence the values of the derived variables.
3. Variables which we want to track
   - {Interested Variables} = {Function Variables} + {Derived Variables}
4. foreach node in the CFG
   - if (statement is assignment statement)
     - a. Store the value of the variable in the form <Variable, its value, {Functionality}>
   - else if (statement is IF-THEN)
     - b. condExpr = Extract the condition expression;
     - c. valueSet = Constant Propagation();
     - d. resultSet = evaluateExpr (condExpr, valueSet);
     - e. if (resultSet is not empty)
       - Mark the Then-block statement with TRUE functionality context.
     - else if (resultSet is empty)
       - Mark the Else-block statement with FALSE functionality context.
     - else
       - Mark the Then-block statement with FALSE functionality context.
     - else
       - Mark the Else-block statement with TRUE functionality context.
     - else
       - Mark the block statement with functionality of its parent block.
   - end if

```
In the above algorithm, we traverse the control flow graph of the program taking appropriate action at every node that is encountered. For every assignment statement, we maintain the value for tracking derived variables and their values. For every conditional check, we call evaluateExpr() subroutine that makes use of constant propagation to evaluate the condition. The result of evaluateExpr() is used to appropriately mark the source code.

6. RESULTS

A sample COBOL program is shown in listing 7 on which we apply the algorithm and discuss the results.

Listing 7: An Example Program

```
1 2 PROCEDURE DIVISION
3   A000--MAIN-LOGIC.
4     MOVE 22 TO MOVE-VAR
5     MOVE 25200 TO STRT-AREA-TRN-TYPE.
6     IF STRT-AREA-TRN-TYPE = 25200
7       MOVE 526 TO DB-FUNCTION
8       PERFORM A THRU A-EXIT
9       ELSE
10      END-IF.
11     ELSE
12       MOVE 734 TO WA-OUTPUT-ST
13       PERFORM B THRU B-EXIT
14     END-IF.
15     MOVE 222 TO DB-FUNCTION
16     IF STRT-TRN-NO = 20040
17       MOVE MOVE-VAR3 = MOVE-VAR1 + MOVE-VAR1
18       MOVE UT0080-BINARY-DATE TO BRCL-REP-DATE
19       MOVE UT0080-BINARY-DATE TO WA-RESPOND-DATE
20       MOVE 444 TO DB-FUNCTION
21       END-IF.
22     IF DB-FUNCTION = 444 || STRT-TRN-NO = 20040
23       COMPUTE MOVE-VAR2 = TEMP-VAR1 + TEMP-VAR2
24       MOVE 0 TO BRCL-OTH-DATE
25       MOVE SPACES TO BRCL-OTH-SCH-NO
26       END-IF.
27     END IF.
28     IF STRT-TRN-ATT = 21443
29       PERFORM F THRU F-EXIT.
30     END-IF.
31     EXIT PROGRAM.
32     A.
33     MOVE 21443 TO STRT-TRN-ATT.
34     IF WA-ORIGIN-DATE > WA-RESPOND-DATE
35       MOVE 126 TO WA-OUTPUT-ERR-MSG-NO
36       MOVE 1 TO WA-MSG-ERR-FLAG.
37       MOVE 22040 TO STRT-TRN-NO.
38     END IF.
39     B.
40     MOVE 20040 TO STRT-TRN-NO.
41     MOVE SPACES TO DBIOMOD-STAT.
42     MOVE 22443 TO STRT-TRN-ATT.
43     COMPUTE MOVE-VAR3 = DB-FUNCTION + MOVE-VAR1.
44     IF TEMP-VAR2 = 10
45       MOVE 111 TO MOVE-VAR2.
46     END-IF.
47     C.
48     MOVE SPACES TO FCTX-RECORD-AREA.
49     MOVE TRST-INST-NO TO FCTX-INST-NO.
50     IF STRT-TRN-NO = 21040 || STRT-TRN-NO = 22040
51       DISPLAY "IN UPDATE OR DELETE"
52       MOVE 10 TO TEMP-VAR3.
53     END-IF.
54     D.
55     EXIT.
56     END-IF.
57     A-EXIT.
58     EXIT.
59     B-EXIT.
60     EXIT.
61     C-EXIT.
62     EXIT.
63     D-EXIT.
64     EXIT.
```

Table 1 shows the functional variables for the sample program along with their constant values. The constants are called as transaction codes and represent some functionality in the source code.

<table>
<thead>
<tr>
<th>Variables</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRT-TRN-NO</td>
<td>20040, 21040, 22040</td>
</tr>
<tr>
<td>STRT-TRN-ATT</td>
<td>21443, 22443</td>
</tr>
<tr>
<td>STRT-AREA-TRN-TYPE</td>
<td>25200, 26200</td>
</tr>
</tbody>
</table>

Table 1: Function Variables and Values

Table 2 shows the mapping between the transaction codes and respective functionality (name) in the source code. Transaction codes to functionality mapping which is decided at the time of design can be obtained from a system expert who is typically the architect or designer of the application. Even the developer is
likely to be able to provide the information.

<table>
<thead>
<tr>
<th>Function Variable Values</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>20040</td>
<td>ADD</td>
</tr>
<tr>
<td>21040</td>
<td>UPDATE</td>
</tr>
<tr>
<td>22040</td>
<td>DELETE</td>
</tr>
<tr>
<td>21144</td>
<td>CREDIT</td>
</tr>
<tr>
<td>22443</td>
<td>DEBIT</td>
</tr>
</tbody>
</table>

Table 2: Function variable values and functionalities

When the language parser and program analyzer processes the program, they generate control flow graph \([\text{CFG}]\) of the program. Program analyzer uses the CFG to compute constants and propagates them as far forward as possible.

Next, we discuss applying the algorithm to the sample program.

**Step 1**: All the statements from line (1...64) will have initial marking of ALL.

**Step 2**: Derived Variables = \(\{\text{DB-FUNCTION}, \text{TEMP-VAR2}\}\)

**Step 3**: Interested Variables = \(\{\text{STRT-TRN-NO}, \text{STRT-TRN-ATT}, \text{STRT-AREA-TRN-TYPE}, \text{DB-FUNCTION}, \text{TEMP-VAR1}, \text{TEMP-VAR2}\}\)

**Step 4**: Tracked Variables = \(\{\text{STRT-TRN-NO}, \text{STRT-TRN-ATT}, \text{STRT-AREA-TRN-TYPE}, \text{DB-FUNCTION}, \text{TEMP-VAR2}, \text{TEMP-VAR3}\}\)

**Step a**: This step will be applied at all the assignments from the source program. For Example, @line number 4,

- \(<\text{MOVE-VAR}, 22, \{\{\text{ALL}\}\}>\)

- \(<\text{MOVE-VAR}, 22, \{\{\text{ALL}\}\}>\)

- \(<\text{MOVE-VAR}, 22, \{\{\text{ALL}\}\}>\)

Two different tuples in the result set for variable DB-FUNCTION signifies two different contexts with two different values. In such a manner information will be accumulated at every assignment and propagated in the program.

**Step b, c, d, e, f**: This step will be executed at CFG node of a decision block. For Example, @line number 7

- \(\text{condExpr} = \text{DB-FUNCTION} = 444 \ || \ \text{STRT-TRN-NO} = 20040\)

- \(\text{valueSet} = \text{ConstantPropagation}()\)

- \(\text{result} = \text{result} = \{\text{<TRUE, STRT-TRN-NO, 20040>}, \text{<TRUE, DB-FUNCTION, 444>}\}\)

- As result == TRUE, the THEN code block will be marked with value 20040 OR 25200. 25200 appear because derived variable DB-FUNCTION’s value 444 is mapped to function variable value 25200.

Similarly, this process is continued, applying each step of the algorithms as the different nodes are encountered in the CFG.

**7. APPROACH TO AUTOMATION**

Mapping between code blocks and various functionalities is not a 1:1 mapping. It can be non-injective as one block can perform multiple functionalities. Performing such mapping becomes even more difficult in the presence of spaghetti paths and complex condition expressions that need to be evaluated during the analysis. Legacy programs are usually very large in size, which adds to the complexity. Therefore, having an automated technique for functionality marking is very crucial.

Our proposed technique is an application of static data and control-flow analysis and partial evaluation. It requires minimal manual intervention for the part where values of the function variables need to be mapped to functionalities they represent. Presently, we are working on the implementation of this technique. Scalability of the automated technique to handle industry size programs is an obvious concern. To start with, we are implementing this technique for procedural languages though we intend to make the automation independent of any programming language, which requires making the design devoid of programming language specificities. The limitation of procedural languages is primarily due to the underlying data-flow analysis tool [2], which supports only procedural languages at the moment.

One scalability aspect is the representation of the condition variables and their values. For the IF-THEN-ELSE decision block, we need to compute the \(\text{NOT}\) of a condition to represent the functionality in ELSE code block. Representing \(\text{NOT}\) of a condition that involves more than one function variable becomes very large since it means every other possible value. Another problem with the condition evaluation is that, all possible satisfying assignments to such expression increases exponentially with the number of variables involved in the condition as well as possible values of the variables involved. We are engineering our solution to address the issues.

Handling such evaluation and its satisfying assignment matches intuitively to Multi-valued Logic. In multi-valued logic, variables involved in function can take values from defined set of discrete values and it can have multiple values as output. However, in our case we have variables taking multiple values but output will always be binary \((0 \text{ or } 1)\) i.e., whether condition evaluates to TRUE or FALSE. We use multi-valued decision diagram \([16]\) for
representing the condition as it simplifies computing the logical NOT of a condition.

8. CONCLUSION
We have proposed a new technique to locate functionality (features) in the source code, based on a minimal knowledge of the representation of the domain functionality (functional variables) in the source code. The technique is able to identify mapping of detailed (statement-level) program elements to functionality, especially in legacy business applications. We have verified the feasibility of the technique using a representative case study that is derived from a real-life industry application. The technique has high feasibility of automation, which is work in progress. We intend to apply the automated technique to a number of representative, real-life business applications and confirm its usefulness in industry. In the future, we plan to integrate other known techniques in our automation, so as to provide automated detection of features in most of the business (software) applications.

9. REFERENCES