A formalism to automate mapping from program features to code

Jean-Christophe Deprez & Arun Lakhotia
University of Louisiana at Lafayette
Center for Advanced Computer Studies (CACS)
Phone: (337) 482-{5062, 6766}
Fax: (337) 482-5791
E-mail: {jxd7803, arun}@cacs.louisiana.edu

Abstract

How does one locate the segments of code that implement a particular feature? Wilde and Scully (WS) pioneered the use of execution traces to map program features to code. Using their technique, to locate the implementation of a particular feature, a program is executed with two sets of inputs; one set invokes the feature of interest and the other set does not. Operations such as set-difference and set-intersection, amongst others, are then applied on the execution traces to obtain answers for various questions related to a feature and its implementation. Previous researchers have automated the tasks of computing the execution traces and performing operations on the execution traces. We present a formalism to automate the most time-consuming aspect of this approach for locating code, namely, the partitioning of the input-sets into invoking and non-invoking sets. A collection of input-sets is partitioned using feature syntax, a grammar of the program's input annotated with feature names. An input-set is placed in the invoking set if and only if its parse tree is annotated with that feature.

WS' technique solely applies set operations on the execution traces of inputs, in our technique, we also apply the set operations among the set of features used by these inputs. By doing so, we can precisely determine the features whose implementation is identified when applying the operations on the execution traces.

1 Introduction

When modifying a software system, one often needs information on the code segments responsible for implementing a particular feature [1, 2]. This information need may be expressed by the following types of questions:

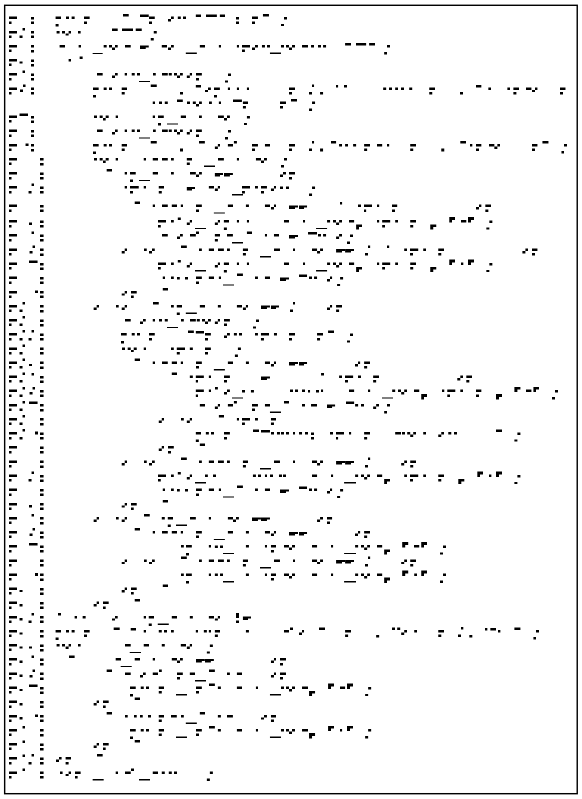
- Which segments of code participate in implementing feature *f*?
- Which segments uniquely implement feature *f*?
- Which segments are indispensable to the implementation of feature *f*?

Wilde and Scully (WS) pioneered a technique to compute answers to such questions using information collected from executing an instrumented version of a program [3, 4]. WS's technique may be expressed as executing the following three steps. A programmer with the intention of locating the implementation of a feature *f* first identifies a set of *invoking* input sets, which exhibit feature *f*, and a set of *excluding* input sets, which do not exhibit feature *f*. Second, the programmer executes the program with all the input sets to create their execution traces. In the last step, the programmer selects one of the methods developed by WS to map the feature *f* to the code segments implementing it. For example, WS proposed to answer, respectively, the three questions above by collecting the segments of code belonging to:

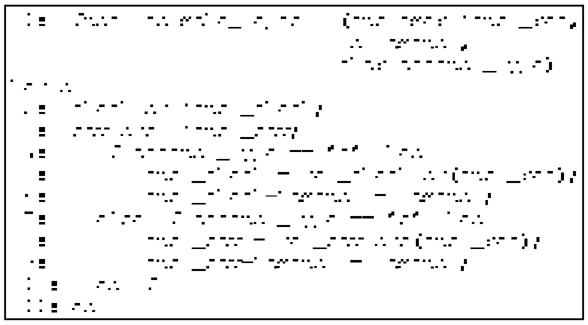
- The execution traces of any *invoking* input sets.
- The execution traces of any *invoking* input sets not belonging to the execution traces of any *excluding* input sets ($I_{invoking} - I_{excluding}$).
- All the execution traces of the *invoking* input sets.

This technique for identifying relations between program feature and code has also gained credence as evidenced by its use by other researchers. Reps *et al.* independently developed a similar approach for finding code segments that implement date related computation in order to solve the year 2000 problem [5]. Wong *et al.* developed three heuristics to locate the code that uniquely implements a particular feature [6]. They incorporated these heuristics in χSuds, a tool that provides the necessary support for instrumenting programs, managing test cases, and collecting execution traces from instrumentation. Several experiments have also proven the usefulness of such different techniques. WS and Wong *et al.* have conducted experiments in which programmers used their techniques to gain knowledge of an unfamiliar program. In both cases,

0-7695-0656-9/00 $10.00 © 2000 IEEE



```
[illegible]
```



```
[illegible]
```

Figure 1: Example of an implementation for the ATM program. On the top, the function *main* of the program, and on the bottom, the function *make_deposit* that implements the deposit operation.

the programmers could rapidly locate the implementation of features, and the implementers/experts of the system validated the knowledge acquired by the programmers.

According to WS, "The most time-consuming step, and the one most difficult to automate, is the preparation and running of test cases." In this paper, we propose a formalism to automate these tasks.

More specifically, we present:

- A formal definition for program features based on the *feature syntax* of a program. *Feature syntax* links a context free grammar that specifies the syntax of the input of a program to the features of the program.
- A method to automatically identify the features invoked by an input set.
- A method to determine precisely the features whose implementation is identified by a particular set operation on specific input sets.

Due to the coupling between features, it is likely that some answers will not be precise. That is, the implementation of other features than the one of interest may also belong to the answer. We develop a mechanism to identify such imprecision, and we illustrate its use in Section 4.

The rest of this paper is organized as follows. Section 2 illustrates a current technique to map features to code. Section 3 presents *feature syntax*, a formal definition of *program feature* and a method to automatically identify the feature used by input sets. Section 4 illustrates our method to map features to code. In Section 5, we discuss the expressiveness and limitation of our work. Section 6 relates our work to other efforts. The conclusions and future work appears in Section 7.

2 Overview of a current technique to map features to code

In this section, we first present the specifications of a bank ATM and a possible implementation for the ATM program in Figure 1. Then, we explain a scenario of program maintenance where a programmer utilizes a technique that operates on execution traces in order to locate the code that uniquely implements a specific feature of the ATM program.

2.1 Bank ATM example

The following constraints describe the operational requirements of a simple bank ATM:

1. A customer should be automatically prompted for a PIN.
2. After an operation, the customer must have the opportunity to start another operation.
3. At any point of an operation, the customer must be able to cancel the current operation, and s/he should be asked if s/he desires to continue with another operation.
4. After the insertion of a PIN, the customer should be proposed these three sorts of operations: make a deposit, make a withdrawal, or check one account balance.
5. After the operation is chosen, the customer should be asked the account on which to perform the operation: checking or savings.
6. In the case of a withdrawal, the customer should also be asked to enter a positive number that represent the amount to withdraw from the selected account.

Furthermore, if the withdrawal is done on the checking account then the amount must be equal or less than \$300. The amount should be deducted from the corresponding account.
7. In the case of a deposit, the customer should be able to slide bills of \$5, \$10, \$20, \$50, and \$100 in the ATM. The corresponding account should be credited.
8. In the case of a balance operation, the balance of the corresponding account should be displayed in the screen.
9. Once the series of operations is terminated, the customer should be asked if s/he desires a ticket. On a positive answer, a ticket with the balance information of all accounts that have been affected should be printed.

2.2 Scenario of software maintenance for the ATM

Let us assume that the ATM does not operate correctly when customers deposit money in their savings accounts. After several complaints, the bank management asks a programmer to investigate their ATM source code. From the description of the problem, the programmer knows that the faulty code is probably located in the implementation of the feature '*deposit in savings account*'. Instead of dealing with the entire code of the ATM program, the programmer can use the following steps to identify the implementation of the feature '*deposit in savings account*':

1. Identify or build input sets that *invoke* the feature '*deposit in savings account*',
2. Identify or build input sets that *do not invoke* the feature '*deposit in savings account*',
3. Execute the instrumented program with each input set to create its execution trace
4. Classify each execution trace in the *invoking* category if its input set invokes the feature '*deposit in savings account*' or in the *non-invoking* category if it does not.
5. Apply a method that operates on execution traces to map the feature '*deposit in savings account*' to code. For example, WS' method to locate the implementation unique to a feature uses the operation $\cup_{invoking} - \cup_{excluding}$.

The current implementations of such techniques by WS, by Wong *et al.*, and by Reps *et al.* automate steps 3 and 5; a programmer must manually perform the other steps. When performing step 5, a programmer must also decide which set operations to apply on the execution traces. Different operations identify different aspects of the implementation of a feature, for example, the program components always executed by a feature or the program components uniquely involved in the implementation of a feature. Thus, to locate feature '*deposit in savings account*', suppose that a programmer has identified two input sets that simulate the following scenarios:

- t_1: A customer enters a PIN, deposits \$100 in a savings account, and does not require a ticket.
- t_2: A customer enters a PIN, deposits \$100 in a checking account, and does not require a ticket.

Using an instrumented version of a program, it is possible to collect the execution trace of an input set. An execution trace contains information about the components of a program executed by the program with a specific input set. Components of programs can be statements, branches, functions, procedures, files, etc. Let us define a function that computes the components exercised by an input set.

Definition of *execution trace*: Given an instrumented program p and an input set t, the *execution trace* defined by t on p, denoted $p(t)$, is the set of components of program p exercised by the execution of p with the input set t. Program components can either be statements, condition branches, procedures, files, etc.

For this paper, we define the components at the statement level. Let us assume that a statement is denoted by i where i is unique for each statement, and p denotes the instrumented version of the bank ATM program. Then, Figure 2 shows the execution trace $p(t_1)=\{i \mid i = m_0..m_5, m_7..m_{10}, m_{12}..m_{13}, m_{15}, m_{18}, d_1..d_3, d_7..d_{10}\}$, and Figure 3 shows the execution trace $p(t_2)$. In Figure 2,Figure 3, andFigure 4, the code not executed by the two input sets t_1 and t_2 is replaced by ellipses.

The final step consists of operating on the execution traces to locate the implementation of the feature '*deposit in savings account*'. WS propose to apply the operation $\cup_{invoking} - \cup_{excluding}$ on the execution traces to identify the code uniquely involved in the implementation of a particular feature. In this case, the input set t_1 exhibits the feature '*deposit in savings account*' therefore it is an invoking input set, whereas input set t_2 does not hence it is an excluding input set. Figure 4 shows the result of $p(t_1)$ - $p(t_2)$. A programmer could use the segments of the code shown in Figure 4 as a starting point to locate the defect in the ATM program.

3 Features of a program and features of input sets

In this section, we present a method to automatically determine the features used by an input set based on its syntax. Since a program defines a notation for a language, its syntactic properties can be expressed using a grammar. Using the grammar of a program, we can parse its input sets. Therefore, if our definition of a program's features is linked to its input grammar, then the features invoked by that input set can be determined by parsing it.

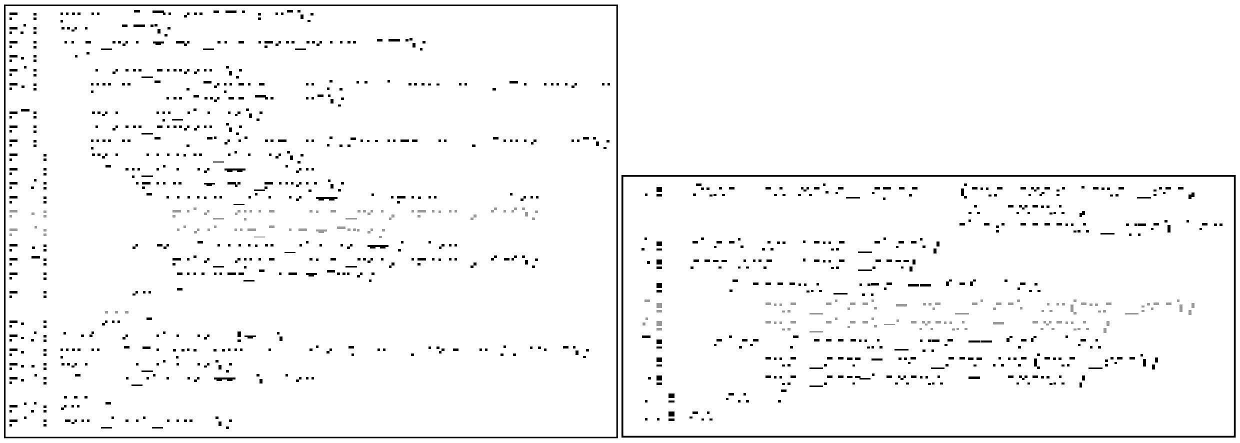

[illegible]

Figure 2 : Execution trace of the input sets t1, p(t1). The shaded code is not executed by t1.

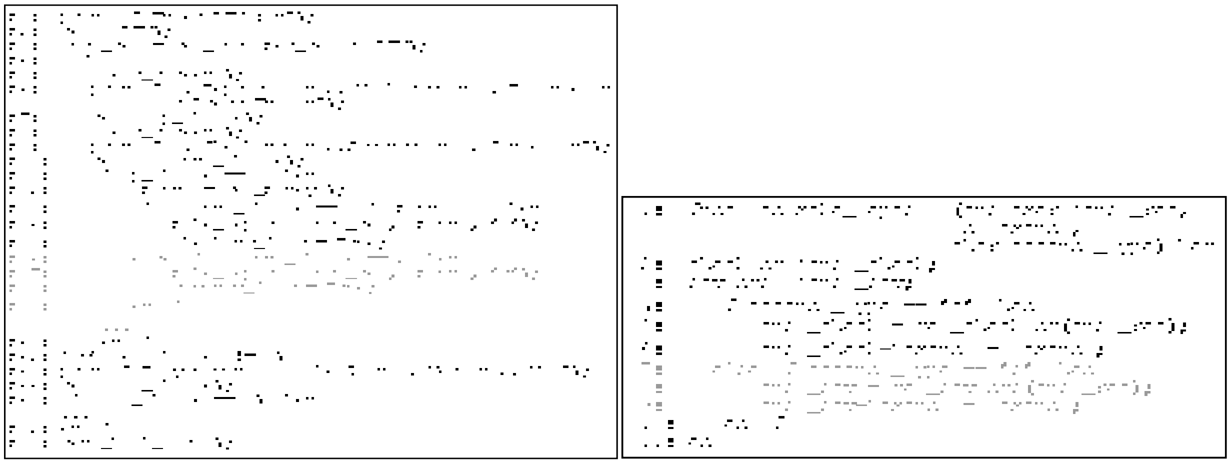

[illegible]

Figure 3 : Execution trace of the input sets t2, p(t2). The shaded code is not executed by t2.

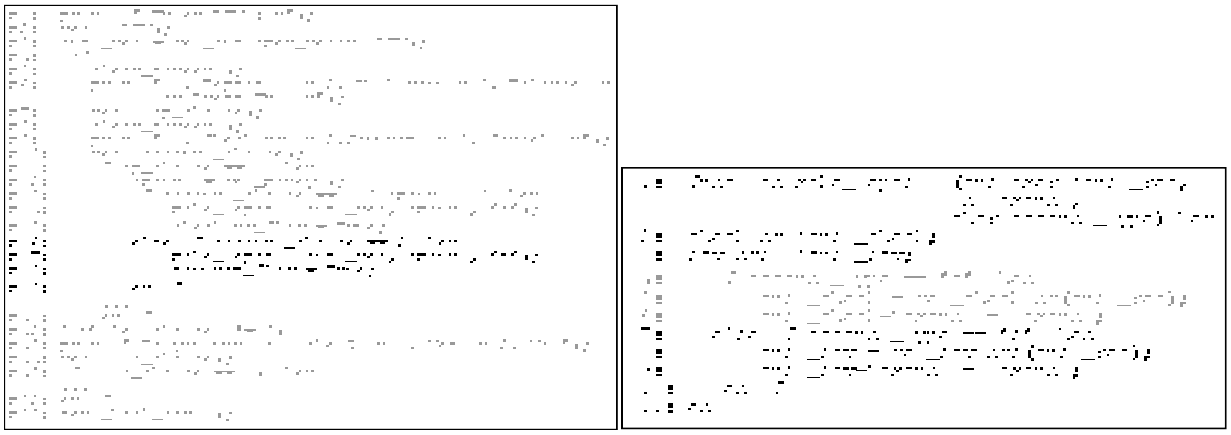

[illegible]

Figure 4 : By subtracting the execution trace p(t2) from the p(t1), we show the implementation only use by the feature "deposit in savings account".

Additionally, the grammar of a program's input (including user interface) can be compiled from the software specifications or through external observations of the behavior of a program, and without any knowledge of the source code. In this paper, we define a notation, called *feature syntax*, to express the syntax and the features of a program. *Feature syntax* consists of two parts, the first part is a context free grammar describing the inputs of the user interface, and the second part creates the links between the grammar and the features of the program

Feature Syntax for the bank ATM

```
Grammar                                        Features and their mapping to the grammar φ(F)

[illegible] ::= [illegible]                    [illegible]
[illegible] ::= [illegible] | Withdraw [illegible] | Deposit [illegible] |    [illegible]
      Balance [illegible] | Finished           [illegible]
[illegible] ::= Checking [illegible] | Savings [illegible] | Abort    [illegible]
[illegible] ::= Checking [illegible] | Savings [illegible] | Abort    [illegible]
[illegible] ::= Checking | Savings | Abort     [illegible]
[illegible] ::= [illegible] Finished           [illegible]
[illegible] ::= [illegible] | Finished         [illegible]
[illegible] ::= $5 | $10 | $20 | $50 | $100    [illegible]
[illegible] ::= [illegible] Finished           [illegible]
[illegible] ::= Yes | No                       [illegible]
[illegible] ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 |    [illegible]
      8 | 9 | 0                                [illegible]
                                               [illegible]
```

Figure 5 : Example of *feature syntax* created from the requirements of the bank ATM. The *grammar* column represents the BNF grammar that specifies the syntax of the input sets to the ATM program, and the *feature* column specifies the links between features and elements of the grammar. The superscripts found in the "Features" section (right column) help identify a particular member. The superscript is needed to distinguish between members with a similar syntax, for example, Abort appears on the right-hand side of several production rules. Thus, to differentiate between the different occurrences, we superscript a member with the term to which it belongs.

The rest of this section is organized as follows. We first define *feature syntax* and *program feature*. Then, we present the *feature syntax* derived from the specifications of the bank ATM example. Finally, we show how to determine the features invoked by an input set.

The following defines some notations to refer to the different components of a context free grammar. We introduce new notations because the standard BNF notations [7] do not provide means to refer to the internal parts of a production rule. In our case, we need to refer to non-terminals and also to the different options on the right hand side of production rule. In our notation, we refer to an option as a *member*.

Notations: Given a BNF grammar *G*,

- *Term(G)* denotes the set of all the terms in the grammar *G*,
- *Member(G)* denotes the set containing each options on the right hand side of productions for every term of grammar *G*,
- *Element(G)* denotes the union of the terms and members of the grammar *G*, (*Term(G)*∪*Member(G)*).

An example of *feature syntax* for the ATM program is shown in Figure 5.

Definition of *feature syntax*: $FS = (G, F, \varphi)$ is a *feature syntax* where

- *G* denotes a BNF grammar,
- *F* denotes the set of features of a program (left hand side of the arrows in the *feature* section),

$\varphi: 2^{Element(G)} \rightarrow 2^F$ is a function that maps a set of elements of the grammar to the features it represents. This function represents the links between the elements of the grammar and the features of a program. For example, in the 'Feature' section of Figure 5, the first line [illegible] means that φ *({IP})* = *'Customer session'* or that φ *({IOT^P})* = *'Customer session'*. In other word, if the parse tree of an input contains either [illegible] or [illegible] then the feature *Customer Session* is invoked by the input.

Definition of *program feature:*

Given a feature $f \in F$, and a set of elements of grammar G, $E = \{e_1, e_2, \ldots\} \subseteq Element(G)$, such that $f \in \varphi(E)$, we say that the set of elements E represents feature f.

Note:

- The *grammar* section of feature syntax is a context free grammar expressed in BNF. We require context free grammar over regular expression for two reasons. First, certain programs such as compilers accept input that are context free thus, to identify the implementation of a feature of compilers we must define our notion of *feature* using context free grammar. Second, BNF notation allows levels of abstraction not easily expressed using regular expression. That is, we find it simpler for a programmer to assign the notion of features to terms and members of a BNF grammar rather than to parts of regular expressions.
- φ maps a set of elements of the grammar to a set of features. In the ATM example, φ could simply map one element to one feature but, in the general case, an element of the grammar could participate in the definition of several features, and conversely, several elements of the grammar could be grouped to represent one feature thus, φ is a many-to-many function.
- In the *feature* section of feature syntax, a set of elements E maps to a feature f; this means that f is invoked by an input set if the elements in E are found in the parse tree of that input set. On the other hand, no assumption can be made about the order in which the elements of E appear in the parse tree.
- Since the notation of the *feature* section only list one feature per line and that it is possible for a set of elements to map to more that one feature then that set of elements would appear on the right hand side of an arrow of several features.
- In the *feature* section of Figure 5, every feature starts with F_i. The symbols F_i are not an integral part of the *feature syntax*, they are introduced to simplify cross-referencing. In the remaining paper, instead of referencing features by their entire phrase, we often use the corresponding symbol F_i.

3.1 *Features used by input sets*

The grammar of *feature syntax* can be used to parse all valid input sets of a program, and from the parse tree of any input set, we can derive the set of features used by that input set.

Definitions:

- $\Lambda(G)$ denotes the set of all phrases that can be parsed using grammar G.
- $T(G)$ denotes the set of all parse trees that can be produced when parsing every phrase accepted by G.
- $P: \Lambda(G) \rightarrow T(G)$ is a function that transforms a phrase of the language into its parse tree structure.
- $E: T(G) \rightarrow Element(G)$ is a function that traverses a parse tree and returns a set of elements that belong to the parse tree.
- $\Phi: \Lambda(G) \rightarrow F$ is a function that takes a phrase in the language (or an input set) and returns the features used by the input set. Thus, $\Phi = \varphi \circ E \circ P$.

Note: $P: \Lambda(G) \rightarrow T(G)$ and $E: T(G) \rightarrow Element(G)$ are generic functions, they are not specific to a program whereas $\Phi: \Lambda(G) \rightarrow F$ is also specific to a program since φ is specific to a program.

Thus, the expressions $P(t_1)$, $E(t_1)$ and $\Phi(t_1)$ compute the parse tree of t_1, the elements of t_1, and the feature used by t_1 respectively.

$P(t_1)$	=	The parse tree of t_1.
$E \circ P(t_1)$	=	$\{P, I, Z, O, D, M, L, T\} \cup \{HOT^P$, ***ZZZZ Finished***^{I}, $1^Z, 2^Z, 3^Z, 4^Z$, OK^O, ***Deposit D***^{D}, ***Savings M***^{M}, LM^M, ***\$100***^{L}, ***Finished***^{M}, ***Finished***^{L}, $No^T\}$
$\Phi(t_1) = \varphi \circ E \circ P(t_1)$	=	$\{F_1, F_2, F_3, F_4, F_5, F_7, F_{12}, F_{14}, F_{21}, F_{22}, F_{23}, F_{24}, F_{25}\}$

4 Illustration of our technique

We now illustrate how a programmer can use our method to locate the code uniquely involved in the implementation of a feature using the operation $\cup_{invoking} - \cup_{excluding}$.

In Table 1, we first present a potential test suite for the bank ATM program, and show the result of applying our technique to identify the features invoked by each input sets contained in the test suite. Then, we discuss the partition of the input sets and the application of the set-operation $\cup_{invoking} - \cup_{excluding}$ on the execution traces created by the input sets. Additionally, to verify the precision of the answer, we also apply the set-operation $\cup_{invoking} - \cup_{excluding}$ on the sets of features invoked by the input set.

For this illustration, we use the ATM program of Section 2, the set of features defined in the feature syntax of Figure 5, and the input sets of the test suite for the ATM program shown in Table 1.

Here is the description of the generic process used to locate the implementation of any arbitrary feature. First, as shown in the third column of Table 1, we identify the features invoked by each input set of the test suite. Then, we classify the input sets into invoking and excluding sets. The classification is performed by checking whether the set of features invoked by an input set $(\Phi(t_i))$ contains the feature of interest. Finally, we collect the execution traces of all input sets, and we apply the operation

Name	Test content for test phrases	$\Phi(t)$
t_1	1234 Finished Deposit Savings $100 Finished Finished No	{F1, F2, F3, F4, F5, F7, F12, F14, F21, F22, F23, F24, F25}
t_2	1234 Finished Deposit Checking $100 Finished Finished No	{F1, F2, F3, F4, F5, F7, F12, F13, F21, F22, F23, F24, F25}
t_3	1234 Finished Withdraw Checking 20 Finished Finished No	{F1, F2, F3, F4, F5, F7, F8, F9, F20, F25}
t_4	1234 Finished Withdraw Checking 500 Finished Finished No	{F1, F2, F3, F4, F5, F7, F8, F9, F20, F25}
t_5	1234 Finished Withdraw Savings 200 Finished Finished Yes	{F1, F2, F3, F4, F5, F6, F8, F19, F20, F25}
t_6	1234 Finished Balance Checking Finished Yes	{F1, F2, F3, F4, F5, F6, F10, F17, F25}
t_7	1234 Finished Balance Savings Finished Yes	{F1, F2, F3, F4, F5, F6, F10, F18, F25}
t_8	1234 Finished Deposit Savings $50 Finished Finished No	{F1, F2, F3, F4, F5, F7, F12, F14, F21, F22, F23, F24, F25}

Table 1: Test-suite for the ATM program composed of seven input sets. The features used by each input set in the third column are automatically derived using the feature syntax in Figure 5. For the meaningful feature names of each F_i, refer to Figure 5.

$\Pi_{Invoking} - \Pi_{\overline{Invoking}}$ on *both*, the **sets of features**, and the **execution traces**.

In the rest of this section, we show the process to locate the implementation of three features. We use the notation Π^f for the set of inputs that invoke a feature *f*, and $\Pi^{\overline{f}}$ for the set of inputs that do not invoke feature *f*. The following presents three examples where we compute the sets Π^f and $\Pi^{\overline{f}}$ for the features *Savings-Deposit*, *Checking-Withdrawal*, and *Withdrawal*.

1. *Savings-Deposit* feature (F_{14}): t_1 and t_8 are the only input sets invoking F_{14} in Table 1. $\Pi^{Savings-Deposit} = \{t_1, t_8\}$
$\Pi^{\overline{Savings-Deposit}} = \{t_2, t_3, t_4, t_5, t_6, t_7\}$
2. *Checking-Withdrawal* feature (F_9): t_3 and t_4 are the only input sets invoking F_9 in Table 1. $\Pi^{Checking-Withdrawal} = \{t_3, t_4\}$
$\Pi^{\overline{Checking-Withdrawal}} = \{t_1, t_2, t_5, t_6, t_7, t_8\}$
3. *Withdrawal* feature (F_8): t_3, t_4 & t_5 are the only input sets invoking F_8 in Table 1. $\Pi^{Withdrawal} = \{t_3, t_4, t_5\}$,
$\Pi^{\overline{Withdrawal}} = \{t_1, t_2, t_6, t_7, t_8\}$

We now show and discuss the results of the application of the set operations on the execution traces. To compute imprecision due to feature coupling, we also apply the set operations on $\Phi(t)$ (the set of features invoked by the input set t) to verify that only the implementation of the desired feature has been identified. As we'll see in the third example, the set operation $\Pi_{Invoking} - \Pi_{\overline{Invoking}}$ is sometimes not sufficient to locate the implementation of only one particular feature. However, by applying the operations on $\Phi(t)$ of the input sets, we can at least identify the features whose implementation cannot be discarded.

1. *Savings-Deposit* feature (F_{14}):
$$\bigcup_{i\in\{1,8\}}\Phi(t_i) - \bigcup_{i\in\{2,3,4,5,6,7\}}\Phi(t_i) = \{F_{14}\}$$
This shows that, as we expect, since we are trying to identify the implementation solely related to the *Savings-Deposit* feature (F_{14}) will be identified. Figure 4 shows the resulting implementation.
2. *Checking-Withdrawal* feature (F_9):
$$\bigcup_{i\in\{3,4\}}\Phi(t_i) - \bigcup_{i\in\{1,2,5,6,7,8\}}\Phi(t_i) = \{F_9\}$$
This shows that, as we expect, solely the implementation of *Checking-Withdrawal* feature (F_9) will be identified. Figure 6 shows the resulting implementation.

Note on *precision*: the input sets t_3 and t_4 both invoke exactly the same set of features but their execution traces are slightly different. Thus, discarding one of these two input sets to locate the feature of interest would yield less accurate results. For example, if t_4 was removed from the suite then *ut28*, *ut29*, and *ut30* would not be identified as implementing the *Checking-Withdrawal* feature since none of the input sets would have executed these three statements. This shows that input sets even invoking the exact same features may be useful to compute complete answers.

On the other hand, t_1 and t_8 both invoke the same set of features, and also have the same execution traces. Thus discarding one of these input sets would give the same accuracy to the answer. This shows that if several input sets invoke the same features and have the same execution trace then we could ignore all but one of these input sets without affecting the result.

Figure 6: Code pertaining to the *Checking-Withdrawal* feature identified by our algorithm. *m26 & m27* have been executed by t_3 and *m29* has only been executed by t_4.

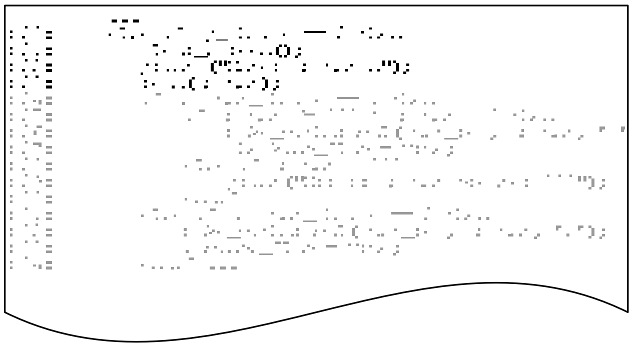


Figure 7: Code pertaining to the *Withdrawal & Amount to Withdraw* features identified by our algorithm. *m20, m21, m22 & m23* have been executed by t_3, t_4 & t_5. The shaded code has been executed by only some of the test cases t_3, t_4, t_5 *but not all of them*.

3. *Withdrawal* feature (F_8):
$$\bigcup_{i\in\{3,4,5\}}\Phi(t_i)-\bigcup_{j\in\{1,2,6,7,8\}}\Phi(t_j)=\{F_8,F_{51}\}$$
In this case, the result will contain *imprecision*. That is, we wanted to identify only the implementation of F_8 (the *Withdrawal* feature), and as shown by the operations on the set of features of the test cases ($\Phi(t_i)$), the implementation identified will also contain the implementation of, F_{51} (the *Amount to Withdraw* feature). A further analysis of the grammar of the ATM shows that, using the operation ($\cup$ $_{testing}$ $-$ $_{\cup testing}$) it is impossible to separate the implementation of F_8 and F_{51} because every input that invokes F_8 must also invoke F_{51}. Thus, using these set operations, it is impossible to locate only the implementation of the *Withdrawal* feature but our method can warn the programmer of such an *imprecision*. Figure 7 shows the resulting implementation.

Additionally, note that if the test case t_6 was not present in the suite, then the operations on the set of features would become
$$\bigcup_{i\in\{3,4,5\}}\Phi(t_i)-\bigcup_{j\in\{1,2,6,7,8\}}\Phi(t_j)=\{F_8,F_4,F_{51}\},$$
and the implementation identified by our technique would accordingly reflect this imprecision and also include statements *m24* through *m30* in the result. This shows that an incomplete test suite does not always yields the most precise result. On the other hand, by applying the operations on the set of features invoked by the input sets, we can express the imprecision contained in the results. In this case, by showing that the *Checking-Withdrawal* feature (F_4) and the *Amount to Withdraw* feature (F_{51}) would also be contained in the resulting implementation.

5 Expressiveness and limitations of our technique

The difficulty of applying our technique resides in the creation of the feature syntax of a program. BNF grammar is a well-known concept that many programmers know thus, creating the BNF grammar that expresses the input syntax of a program should remain a very small overhead. The second step is to list the features of a program, and link each feature to a set of elements in the BNF grammar. A list of features can be determined from the software requirements as well as from observation of program executions. Experiments are necessary to determine the overhead of building feature syntax of programs.

The expressiveness and limitation of this research comes from two factors:

- *The definition of program features.*

Our definition of program feature is based on the notation used for feature syntax. In turn, the expressiveness of feature syntax is determined, on one side, by the grammar section, and on the other, by the feature section.

- Since feature syntax encodes the syntax requirements of a program using a context-free grammar, our approach can express all syntactic concepts of context-free languages. Many software systems define languages at the context free level or lower. Even transaction-based or request-based programs such as applications with GUTs can benefit from our technique since there exist testing tools that capture user commands and inputs into scripts, and that it is possible to build BNF grammars that express the syntax of these scripts.
- Feature section of feature syntax expresses our definition of program feature. It represents the function φ that maps a set of elements of a grammar to the features it represent. This definition allows for any set of elements to

represent several features. On the other hand, our definition does not allow for a program feature to be expressed on the order in which elements of a grammar appear in the parse tree of an input set. For example, our definition does not differentiate between the following two scenarios of the ATM program. Scenario 1, a customer first withdraws money from a checking account and then deposits money in a savings account, and Scenario 2, a customer first deposits money in a savings account and then withdraws money from a checking account. There are three reasons we do not want to make our definition dependent on the order of elements in a parse tree:

1. Most implementations will execute the same program components independent of the order of the operations. For example, our implementation of the ATM program executes the same program components whether executed with Scenario 1 or Scenario 2 above therefore, not requiring a differentiation between the two scenarios.
2. We want the feature syntax notation to remain simple. Making features dependent on the order in which elements appear in the parse tree would require a more complex notation and definition of features such as representing features in terms of sub-tree.
3. When a grammar is recursive, there exists an infinite number of parse trees thus, there could be an infinite number of features. In practice, it is impossible for a programmer to specify an infinite amount of features in the *grammar* section of feature syntax. Thus, it would be possible to create valid inputs which could not be assigned and features.

- *Input sets contained in a test suite.*

In Section 4, we present a method that integrates our technique that identifies the features invoked by an input set with WS' technique in order to map features to code. To alleviate the need to build or to identify input sets, we propose to use the input sets of an existing test suite. Thus, the answers will depend on the quality of the input sets present in a test suite. If certain features are not invoked by any input sets or if the execution traces of the input sets do not cover certain parts of the code then the process that locates the implementation of features will lose precision. Imprecision is unavoidable when applying dynamic techniques. The best we can do to minimize imprecision is to execute a program with many input sets that, themselves, invoke many different combinations of features. Using such an approach, we may hope that the complete implementation of each feature has been covered by some of the input sets. This is the reason we recommend using test suites to collect our input sets. Test suites are often very large and they contain a large amount of input sets that invoke many combinations of features with various data sets in order to maximize code coverage. The down side of using a test suite is that software testers often build several input sets around the same testing criteria. In such cases, many input sets invoke the same set of features thus, could create the same execution trace making some input sets unnecessary to map a feature to code with precision, for example, one of the input sets i_2 and i_3 could be discarded when mapping features to code, and the answers would all keep the same precision. We are investigating techniques to reduce the number of input sets necessary to compute an answer while maintaining the same level of precision. These methods can either discard redundant input sets from a test suite, or generate the necessary input sets from the BNF grammar of the feature syntax. Many researchers have already investigated the problem of input generation from a grammar [8, 9, 10, 11].

6 Related works

Many efforts in test case generation [8, 9, 10, 11], and in generation and update of test oracles [12] have used context free grammar to express the syntactic requirements of program inputs. These efforts are not directly related to ours but they show the usefulness and the power of context free grammar to express program inputs.

Three researches are in direct relation to ours. Wilde and Scully [3, 4], who pioneered the use of execution trace to locate the implementation of features, Wong *et al* [6], and Reps *et al.* [5] all developed techniques that operate on execution traces to collect information about features.

These works derive their results from execution traces left by the execution of a program with input sets but, unlike ours, none proposed a method to identify the features invoked by an input set. Wilde *et al.* and Wong *et al.* both developed techniques to identify the implementation of features. Both researches represent an execution trace as a set of statements. On the other hand, Reps *et al.*'s technique does not assign pieces of code to a feature but rather, identifies the points of divergence between several execution traces and let the programmer determine the eventual relation between a program components and a features from the point of divergence. Reps *et al.* represented an execution trace as a sequence of statements. Our work complements these three researches since all can use our method to derive the features invoked by an input set then, apply the operation on the execution traces specific to their technique to map features to code.

7 Conclusion

We have presented a formal definition of program feature based in the feature syntax of a program. Using this formalism, we can automatically identify the features invoked by an input set. We then illustrated a new technique to map features to code that is more automated and less prone to error than the previous ones. Our method can be combined with other techniques that have used execution traces to map features to code. Doing so alleviates the task of the programmer who is not required to create (or to identify) the necessary input sets, instead, the input sets from an existing test suite are used to locate the implementation of features.

Additionally, as shown in the illustrations in Section 4, our technique not only applies set operations on execution traces but also on set of features invoked by input sets, this allows identifying imprecision in resulting implementation.

To improve efficiency, our application needs one improvement. Currently, we use all the input sets of a test suite to create a mapping between a feature and code. Test suite usually contains a large amount of input tests, this translates into long time to compute answers. To reduce the time to compute an answer, we need to reduce the number of input sets. We are investigating two methods to reduce the number of input sets. One method is to generate the sufficient input sets using the BNF grammar of the feature syntax, and the other method is to eliminate redundant input sets of a test suite, and to only consider a small, sufficient subset of input sets to map features to code.

8 References

[1] N. Zvegintzov, "Software: the analysis manual. Application analysis and enhancement," Unpublished (Tutorial), 1993.

[2] R. Wachtel, "Software change: a guide for process improvement and automation," Unpublished, 1992.

[3] N. Wilde and M. C. Scully, "Software Reconnaissance: Mapping Program Features to Code," *Software Maintenance: Research and Practice*, vol. 7, pp. 49-62, 1995.

[4] N. Wilde and T. Gust, "Locating User Functionality in Old Code," Proceedings of Conference on Software Maintenance, 1992, Los Alamitos, 1992.

[5] T. Reps, T. Ball, T. M. Das, and J. Larus, "The use of program profiling for software maintenance with applications to the Year 2000 Problem," Proceedings of ESEC/FSE 97: Sixth European Software Engineering Conference and Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering, Zurich, Switzerland, 1997.

[6] W. E. Wong, S. S. Gokhale, J. R. Horgan, and K. S. Trivedi, "Locating program features using execution slices," Proceedings of the Second IEEE Symposium on Application-Specific Systems and Software Engineering Technology, Richardson, TX, March 1999.

[7] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, techniques, and tools*. Menlo Park, CA: Addison-Wesley, 1986.

[8] P. Purdom, "A Sentence Generator for Testing Parsers," *BIT*, vol. 12, pp. 366-375, 1972.

[9] A. Celentano, S. C. Reghezzi, P. D. Vigna, C. Ghezzi, G. Granata, and F. Savoretti, "Compiler Testing using a Sentence Generator," *Software - Practice and Experience*, vol. 10, pp. 987-918, 1980.

[10] L. Spadafora and F. Bazzichi, "An automatic generator for testing compiler," *IEEE transaction on Software Engineering*, vol. 8, July 1982.

[11] M. Camuffo, M. Maiocchi, and M. Morselli, "Automatic software test generation," *Information and Software Technology*, vol. 32, pp. 337-346, June 1990.

[12] A. A. Reyes and D. J. Richardson, "Transformational Test Driver-Oracle Synthesis," presented at 1999 International Conference on Software Engineering - Workshop on Software Transformation Systems (ICSE-STS 1999), Los Angeles Airport Marriott Hotel, Los Angeles, CA, U.S.A., 17 May 1999.