Scientific Research and Essays Vol. 8(6), pp. 265 -273, 11 February, 2013
Available online at http://www.academicjournals.org/SRE
DOI: 10.5897/SRE11.1020
ISSN 1992-2248 ©2013 Academic Journals

Full Length Research Paper

A new conditional invariant detection framework (CIDF)

Hamid Parvin[1], Hamid Alinejad Rokny[2], Sajad Parvin[1] and Hossein Shirgahi[3]*

[1]Department of Computer Engineering, Islamic Azad University, Mahdishar Branch, Semnan, Iran.
[2]Department of Computer Engineering, Science and Research Branch, Islamic Azad University, Tehran, Iran.
[3]Young Researchers Club, Jouybar Branch, Islamic Azad University, Jouybar, Iran.

Accepted 24 August, 2011

Software engineering included some different process such as designing, implementing and modifying of software. All these processes are done to have fast developed software as well as reach a high quality, efficient and maintainable software. Invariants help programmer and tester to do most steps of software engineering more easily. Invariants are mostly always true but of course with a specific confidence. Since some invariants are produced in some conditions of program execution and not always, conditional invariants can show the behavior of program so much better. For producing this kind of invariants, it might use some technique of data mining such as association rule mining or using decision tree to obtain rules. So the paper will introduce a new perspective to dynamic invariant detection. Also the feasibility of conditional invariant detection is examined and a framework to extract them is proposed.

Key words: Daikon, invariant, association rules, variable relations, decision tree, program point, data mining, software engineering, predicate, verification.

INTRODUCTION

In recent years, invariant plays an important role in software engineering such as software testing and verification. Invariants are properties of program variable and relationships between these variables in a specific line of code which is called program point. Generation of invariants is a significant key in program verification. These properties and relationships among the program variables or constants are always true; thus programmer or tester can estimate the behavior of program in different program points. Invariant also is used in generating software behavioral model (Krkay et al., 2010), so invariant can also be useful in software engineering in this way. With the help of software behavioral model we can lightly perform design, validation, verification, and maintenance. As seems, one of the most significant contributions of invariants is in modifying of code where properties help programmer to verify the code. Software testing takes a considerable time in software development life cycle. Although software testing is done automatically in present day, but traditionally the onus of software testing was human's obligation (Vanmali et al., 2002). Testing is divided in two category; functional testing and structural testing. Functional testing, which is also called black box testing, performs testing without considering of the logic of program but by checking the program output against the input. This kind of testing does not take into account programming inner workings. On the other hand, structural testing or white box testing analyze program according to checking the actual code and knowing about its logic. Invariants are detected by static and dynamic approaches (Ernst, 1999).

In static approaches, runtime behavior and syntactic structures of program are analyzed actual running of code (WeiB, 2007). Static analysis completely is done automatically. One analysis which traditionally has been used in compilers for collecting necessary information in optimization is Data-flow. Indeed, Data-flow analysis detects some essential invariants in each program points and employs these invariants to find out the behavior of program. This kind of behavior can be used in compilers for optimization. Abstract interpretation is a theoretical framework for static analysis (Jones and Nielson, 1995).

On the other hand, Dynamic approaches extract

*Corresponding author. E-mail: hossein.shirgahi@gmail.com.

program properties and information by the help of actual executing of the program code (Ernst et al., 2006). In the other words, by executing the program with different inputs, called test suits, it is possible to detect invariants dynamically. Dynamic invariants extraction emerges to software engineering in recent years with the advent of Daikon (Ernst, 1999). Program properties of certain point of program are reported by use of invariant inference system via different test suits through different executions. Invariant mostly is checked in the entries and exits of each function.

This paper concentrates on the dynamic extraction of conditional invariants. Conditional invariants are the invariants which are revealed in specific form of conditional proposition, throughout all this paper. These invariants emerge dynamically and all of the steps are fully automatic. We are going to improve the quality of discovered invariants dramatically by using association rule mining. In association-rule-based invariant extraction system, invariants are represented through the variables' condition. For inferring invariants, they are two prominent issues (Vanmali et al., 2002): first we would be able to determine the beneficial invariants and then to exert inference on program context. In this paper we handle these two parts.

RELATED WORKS

In this part of the work, we discuss some implementations of dynamic invariant detection. Many valuable efforts have been done in this field but we mention here only these ones which are more relevant.

Dynamic invariant detection, as mentioned, is quoted by Daikon (Ernst et al., 2007). Daikon is the most prosperous software in dynamic invariant detection developed until now, comparing with other dynamic invariant detection methods (Ernst et al., 2007). However this software has some problems out of which the most serious one is being time-consuming.

DySy proposes a dynamic symbolic execution technique to improve the quality of inferred invariant (Csallner, 2008). It executes test cases like other dynamic invariant inference tools but, as well, coincidentally performs a symbolic execution. For each test unit, DySy results in program's path conditions. At the end, all path conditions are combined and build the result.

Software Agitator is a commercial testing tool which is represented by Agitar and is inspired by Daikon (Boshernitsan et al., 2006). Software agitation is a testing technique that joins the results of research in test-input generation and dynamic invariant detection. The results are called observations. Agitar won the Wall street Journal's 2005 Software Technology Innovation Award.

The DIDUCE tool (Hangal and Lam, 2002) helps programmer by detecting errors and determining the root causes. Besides detecting dynamic invariant, DIDUCE checks program behavior against extracted invariants up to each program points and reports all detected violations. DIDUCE checks simple invariants and does not need up-front instrument.

While there are many related work in the dynamic invariant detection, there is lack of any considerable related work about dynamic invariant detection. This makes this paper first attempt to deal with the dynamic detection of conditional invariants.

TERMINOLOGY

In this part of the work, we discuss about notions which we repeatedly use throughout this paper. The aim of this part of the work is to help readers obtain a better perception of the paper.

Definition 1. Invariants can be defined as prominent relation among program variables. Invariants in programs are formulas or rules that are emerged from the source code of a program and remain unique and unchanged with respect to the running phase of a program with different parameters.

Definition 2. Program points are specific points in a program, such as the Enter or Exit point of a function, which serve as report points for variable relations and invariants. Most frequent program points in use are the Enter and Exit points of sub-programs and functions.

Definition 3. Pre-conditions of a program point are the conditions, relations and invariants that hold immediately before approaching to that program point; in the case of sub-programs or a function Enter point of a sub-program or a function acts as its pre-condition.

Definition 4. Post-conditions of a program point are the conditions, relations and invariants that hold immediately after leaving from that program point. In the case of sub-programs, a function Exit point of a sub-program or a function is considered as its post-condition of it. Typically, post-condition also contains relations between the original value of a variable and its modified one (before and after that program point). In other words, invariants in post-conditions contain relations between variables in pre-condition and post-condition.

BACKGROUND

In this part of the work, we discuss about two techniques which help us to obtain the association rules from program code context. We go over association rule mining and a learner tool called decision tree.

Association rule mining

Here we briefly discuss what association rule mining is.

Table 1. Transactions.

Transactions	Items
T_1	A, B, C
T_2	B, C, D
T_3	B
T_4	A, B

Table 2. List of all itemsets.

Itemset	Supports (%)	Large/Small
A	50	Large
B	100	Large
C	50	Large
D	25	Small
A, B	50	Large
A, C	25	Small
A, D	0	Small
B, C	50	Large
B, D	25	Small
C, D	25	Small
A, B, C	25	Small
A, B, D	0	Small
A, C, D	0	Small
B, C, D	25	Small
A, B, C, D	0	Small

To expound consider following definitions:

Definition 5. Let $I = \{I_1, I_2, \dots, I_m\}$ be a set of m distinct attributes, also called *literals*. Let D be a database, where each record (tuple) T has a unique identifier, and contains a set of items such that $T \subseteq I$. An association rule is an implication of the form $X \Rightarrow Y$, where $X, Y \subset I$, are sets of items called *itemsets*, and $X \cap Y = \varnothing$. Here, X is called antecedent, and Y consequent.

Two important measures for association rules, support (s) and confidence (α), can be defined as follows.

Definition 6. The support (s) of an association rule is the ratio (in percent) of the records that contain $X \cup Y$ to the total number of records in the database.

Definition 7. For a given number of records, confidence (α) is the ratio (in percent) of the number of records that contain $X \cup Y$ to the number of records that contain X.

As definitions 6 and 7 express α ($X \Rightarrow Y$) = $s(X \cup Y) / s(X)$. Association rules are usually required to satisfy a user-specified minimum support and a user-specified minimum confidence at the same time. Association rule generation is usually split up into two separate steps:

Table 3. Extracted association rules

Rule	Confidence (%)	Rule Hold
$A \Rightarrow B$	100	Yes
$B \Rightarrow A$	50	No
$B \Rightarrow C$	50	No
$C \Rightarrow B$	100	Yes

Table 4. Boolean literal for transaction of Table 1.

Transaction	A	B	C	D	Transaction
T_1	√	√	√	×	T_1
T_2	×	√	√	√	T_2
T_3	×	√	×	×	T_3
T_4	√	√	×	×	T_4

1. First, minimum support is applied to find all frequent itemsets or large itemsets in a database.
2. Second, these frequent itemsets and the minimum confidence constraint are used to form rules.

The second step is straight forward but the first step needs more efforts. To clarify these definitions, consider the following example:

Suppose there is a small database with 4 items I = {A, B, C, D} and a database with transactions which shows in Table 1. The thresholds for minimum support and minimum confidence respectively are 40 and 60%.

Table 2 shows all related itemsets of Table 1 and their support as well as if it is large or not. As seen in Table 2, there are four large itemsets in the transaction. The first step of association rule generation has been done. Then it is supposed to extract rules with highest confidence for each large itemset. In Table 3 we show some association rules for transaction of Table 1.

As mentioned before extracting association rule from large itemsets is a straight forward work but generating large itemsets needs more effort and is more time consuming. There are some different algorithms for generating itemsets but since discussing them is out range of this paper, we do not mention them because.

One worthwhile issue to say is that literals can be Boolean. Boolean literal are more favorable and much faster for generating large itemsets. To clarify consider Table 4 As seen, we brought up the example Table 1 in Boolean state.

In Table 4, sign ✓ represents the existence of the item and sign × shows absent of that. For example transaction T1 is consist of A, B and C. The results of operating large itemsets generation an association rule mining on Table 4 respectively are Tables 2 and 3. Since generation large

Table 5. Our data.

Tid	Refund	Marital status	Taxable income (K)	Cheat
1	Yes	Single	125	No
2	No	Married	100	No
3	No	Single	70	No
4	Yes	Married	120	No
5	No	Divorced	95	Yes
6	No	Married	60	No
7	Yes	Divorced	220	No
8	No	Single	85	Yes
9	No	Married	75	No
10	No	Single	90	Yes

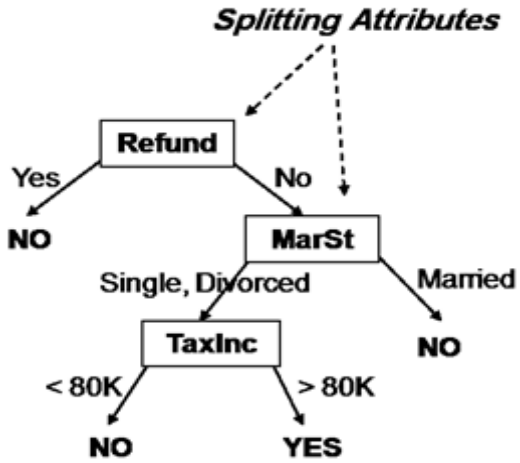

Figure 1. Process tendency for Table 5.

item sets on Boolean literals is faster and more efficient, in CIDF we use this kind of literals.

DECISION TREE LEARNING

Decision tree as a decision support tool uses a tree-like graph or model to operate deciding on a specific goal. Decision tree learning is a data mining technique which creates a model to predict the value of the goal or class based on input variables. Interior nodes are representative of input variables and the leaves are the representative of target value.

By splitting the source set into subsets based on their values, decision tree can be learned. Learning process is done for each subset by recursive partitioning. This process continues until all remaining features in subset have the same value for our goal or until there is no improvement in Entropy. *Entropy* is a measure of the uncertainty associated with a random variable.

Data comes in records of the form: $(x,Y) = (x_1, x_2, x_3,\ldots, x_n, Y)$. The dependent variable, Y, is the target variable that we are trying to understand, classify or generalize. The vector x is composed of the input variables, x_1, x_2, x_3 etc., that are used for that task.

To clarify what decision tree learning is, consider Table 5. Table 5 has 3 attributes Refund, Marital Status and Taxable Income and our goal is cheat status. We should recognize if someone cheats by the help of our 3 attributes. To learn the process, attributes split into subsets. Figure 1 shows the process tendency. First, we split our source by the Refund and then MarSt and TaxInc.

For making rules from a decision tree, we must go upward from leaves as our antecedent to root as our consequent. For example consider Figure 1. Rules such as following are apprehensible. We can use these rules such as what we have in Association Rule Mining.

(i) Refund=Yes⇒cheat=No
(ii) TaxInc<80, MarSt= (Single or Divorce), Refund=No⇒cheat=No
(iii) TaxInc>80, MarSt= (Single or Divorce), Refund=No⇒cheat=Yes
(iv) Refund=No, MarSt=Married⇒cheat=No

CONDITIONAL INVARIANT

Most of invariant extraction systems concentrate on perfect invariants and they are unable to express invariants which are appeared in special situation. This means, the invariants which are reported by invariant extraction system are true with the specific confidence but they do not figure out invariants which are true in a special condition. To clarify the matter, consider Figure 2 (This example is artificial and illustrates several points we are going to discuss).

In this example we assume variables x and y are global. An appropriate unit test for this function might be x<y and its complement. In an ordinary invariant extraction system

```
void compute()
{
      if (x < y)
      {
            int temp = x;
            x = y;
            y = temp;
      }

}
```

Figure 2. Example method whose invariant we want to infer.

```
1 orig(x)>orig(y) -> x=orig(x)
2 orig(x)>orig(y) -> y=orig(y)
3 orig(x)<orig(y) -> x=orig(y)
4 orig(x)<orig(y) -> y=orig(x)
```

Figure 3. Related invariants in our method.

the post-condition invariant which could be detected in of this function is:

(i) x>y

This invariant shows after leaving compute () the x values are always and are greater than y values. This invariant is adequate but it does not present a complete behavior of this function. This means this mere invariant cannot be useful neither in formal specification nor assert statement.

This deficiency puts us to think of having a set of invariants which can appropriately show the program behavior. In other words we need a set of invariants which tell us compute () swaps x and y values when y value is greater. The final outcome of post-condition of compute () invariants (or compute ()::: Exit in our method) are shown in Figure 3.

In upon invariants, orig(var) shows var value just before entrance of compute (). This approach removes the weakness of previous dynamic invariant inference. As could be seen, Figure 3 completely describes the function behavior.

Over all our work contains following parts:

(i) We introduce the idea of using association rule mining for invariant inference. We believe our method makes up the next generation of dynamic in variant inference tools. We believe our approach opens a new ways to perform dynamic invariant inference in not far future.
(ii) We implemented our approach in the invariant inference tool CIDF.

PROPOSED CONDITIONAL INVARIANT DETECTION FRAMEWORK

In this part of the work, we propose our idea in details. First, we provide predicates for each execution of program point and then invariant detector uses these predicate to extract the rules. Program points are usually function entries and exits. Function entries and exits are called Enter point and Exit point of function. For Enter point, all values of global variables and parameters participate while for exit all values of global variables and parameters as well as their prior values participate. With having more variety of invaluable predicates, more beneficial invariants are produced. Extracted rules show behavior of program point in conditional form. In the following part of this work, we discuss the classes of predicates and clarify all predicates.

For better understanding of the process, Figure 4 schematically shows the algorithm flowchart of employing association rule mining in extracting conditional invariants step by step. Each data trace file in Figure 4 contains possible predicate of a program point.

Classes of predicate

Here we present all classes of predicates which might be used by invariant detector. By the help of an association rule mining tool, we can extract conditional invariants. We try to provide a terse set of predicates to have an acceptable potential result but definitely there are some predicates which are missed. The following lists classes of predicates which CIDF computes, where x and y are variables:

(a) Predicates over any numeric variable:

(i) IsNonZero: when the variable is never set to 0
(ii) IsOne: when the variable is always equal to 0
(iii) IsMinesOne: when the variable is always equal to -1
(iv) IsEven: when the variable is always even
(v) IsPowerOfTwo: when the variable is always power of two

(b) Predicates over any string variable:

(i) IsNull: when the variable is always null
(ii) IsEmpty: when the variable contains no characters

(c) Predicates over two numeric variable:

(i) Ordering comparison: $x < y$, $x \leq y$, $x > y$, $x \geq y$, $x = y$, $x \neq y$

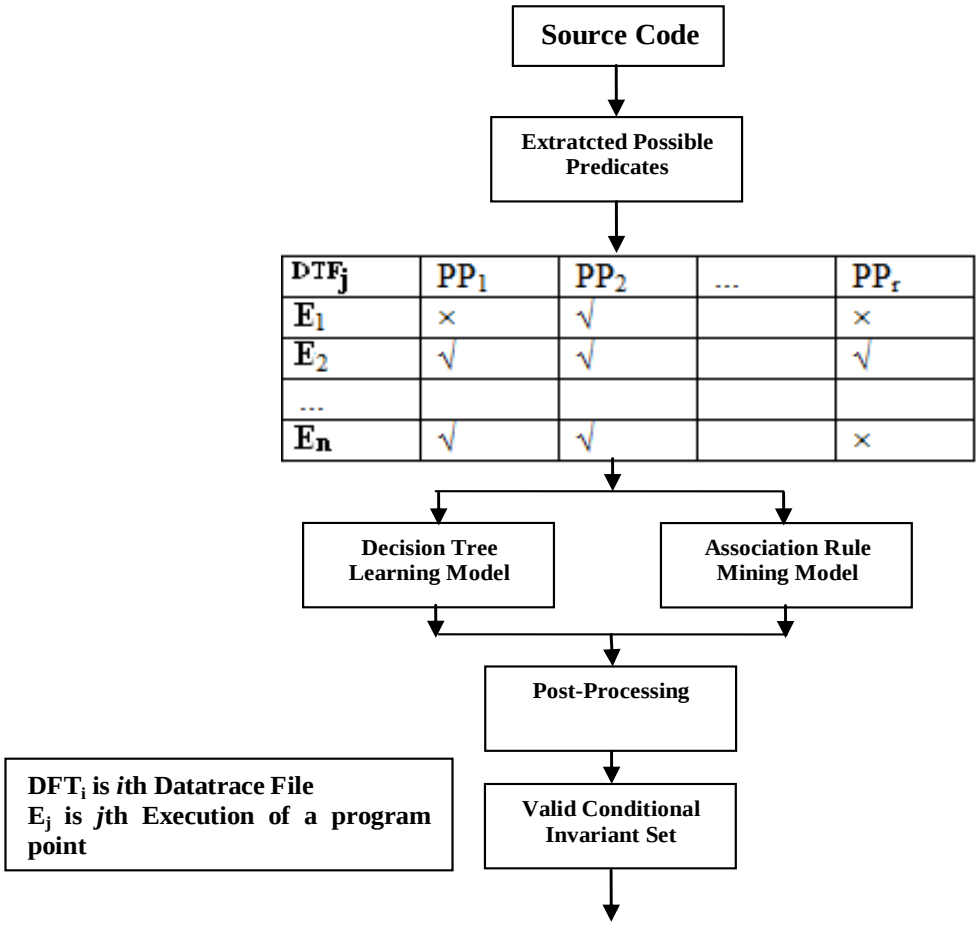

Figure 4. Algorithm flowchart of employing association rule mining in extracting conditional invariants.

(ii) Functions: y = fn(x) or x = fn(y), for fn a built-in unary function (absolute value, negation, bitwise complement)

(d) Predicate over two string variable:

(i) Equality: x = y when two strings are equal
(ii) Substring: y=sub(x) when y is substring of x
(iii) Reversal: y=rev(x) or y=rev(x) when x is the reverse of y

(e) Predicates over a array:

(i) Element relationship: when the array elements are equal or sorted by (=>,>,<,<=)
(ii) IsNonZero: when none of array elements are equal to 0

(f) Predicate over an array and a numerical variable:

(i) Membership: $x \in y$ (x and y are common type arrays)

(g) Predicate over two arrays:

(i) Comparison: $x < y$, $x \leq y$, $x > y$, $x \geq y$, $x = y$, $x \neq y$
(ii) Sub-array: y=sub(x) when y is sub-array of x
(iii) Reversal: y=rev(x) or y=rev(x) when x is the reverse of y

Presented predicates are produce for each program point. Each presented predicates have Boolean values. In other words this predicates might be true or false. By performing association rule mining on these predicate we would have some rules with specific support and confidence. In the following part of this work we discuss about association rule mining and the domination of this technique whether support our aim.

Using association rule mining on defined predicates

In the previously discussed heading “classes of predicate” we defined all predicates which are interfered with for each program point variable. In other words we bring forward any possible predicates in a specified program

Table 6. Related transaction for Figure 2.

Transaction	orig(x)>orig(y)	orig(x)<orig(y)	x=orig(x)	y=orig(x)	x=orig(y)	y=orig(y)
T_1	true	false	true	false	false	true
T_2	false	true	false	true	true	false
T_3	false	true	false	true	true	false
T_4	true	false	true	false	false	true
T_5	true	false	true	false	false	true
T_6	false	true	false	true	true	false

point. The obtained predicates, all, have Boolean values. These values can easily be used for mining association rules as described under the heading “Association rule mining”. Each time for each predicate as consequent, we check other predicates which make relations with the other predicates. Consider we have predicates P_1, P_2, P_3, … , P_q. We start with P_q we check all predicates if they have relation with P_q. It means we check if P_1 as the antecedent can result P_q otherwise we conjunct P_1 and P_2 and check if now they result P_q and so forth. Then we will perform these steps for P_{q-1}.

Closely looking at our paper tendency we consider the presented function in Figure 2. We discussed this function and its Exit program point conditional invariants. Now we demonstrate the steps to create these rules. First we must prepare our database and transactions. Each record shows one executing of function. We instrument the code so that in each execution, predicates between all variables are stored in a file. The result is presented in Table 6.

Table 6 shows neither all transaction nor all predicates but it presents just some of them to manifest the method. The minimum support and minimum confidence respectively are 50 and 100%. Two large itemsets which are inferred from Table 6 is:

(i) orig(x)>orig(y), x=orig(x),y=orig(y)
(ii) orig(x)<orig(y), y=orig(x), x=orig(y)

And the following rules are archived:

(i) orig(x)>orig(y)⇒ x=orig(x)
(ii) orig(x)>orig(y)⇒y=orig(y)
(iii) orig(x)<orig(y)⇒y=orig(x)
(iv) orig(x)<orig(y)⇒x=orig(y)
(v) x=orig(x)⇒orig(x)>orig(y)
(vi) x=orig(x)⇒y=orig(y)
(vii) y=orig(y)⇒orig(x)>orig(y)
(viii) y=orig(y)⇒x=orig(x)
(ix) y=orig(x)⇒orig(x)<orig(y)
(x) y=orig(x)⇒x=orig(y)
(xi) y=orig(y)⇒ orig(x)<orig(y)
(xii) y=orig(y)⇒ y=orig(x)

All presented rules are true and obey minimum support and minimum confidence but only four first one are tangible and others must be filtered. The four first rules are the same as rules we represent in Figure 2. These to conditional invariant describe the behavior of compute (). One thigh which is important to say is in this method rules' Consequent part contains only one predicate and we do not have compound consequences. In whole the process of Association Rule Mining Model box in the Figure 3 is illustrated in the Figure 5.

Time order

Here in, we check our approach time order. It is necessary we check the time order because we want to see if it is affordable. Assuming we have m variables in a program point. Each two variable make a predicate so overall we have q predicates. q is obtained via Equation (1):

$$\binom{m}{2} = q \quad (1)$$

So we have predicates P_1, P_2, P_3, … , P_q. To have a rule with P_q as the consequent, our association rule mining tool must check if each of P_1, P_2, P_3, … , P_{q-1} has relationship with P_q then it has to check if two of P_1, P_2, P_3, … , P_{q-1} have relationship and so forth. Consequently, for having a rule with P_q as the consequent, our tool has to handle (2) number of checks:

$$\binom{q-1}{1} * 2 + \binom{q-1}{2} * 2^2 + \cdots + \binom{q-1}{q-1} * 2^{q-1} \quad (2)$$

Totally, the association rule mining tool must handle (3) number of checks:

$$\sum_{i=1}^{q-1} \sum_{j=1}^{i} \binom{i}{j} * 2^j \quad (3)$$

For example assuming we have 7 variables in one of our program points. The total number of checks might be 2097152. If we have 10 variables the total number of

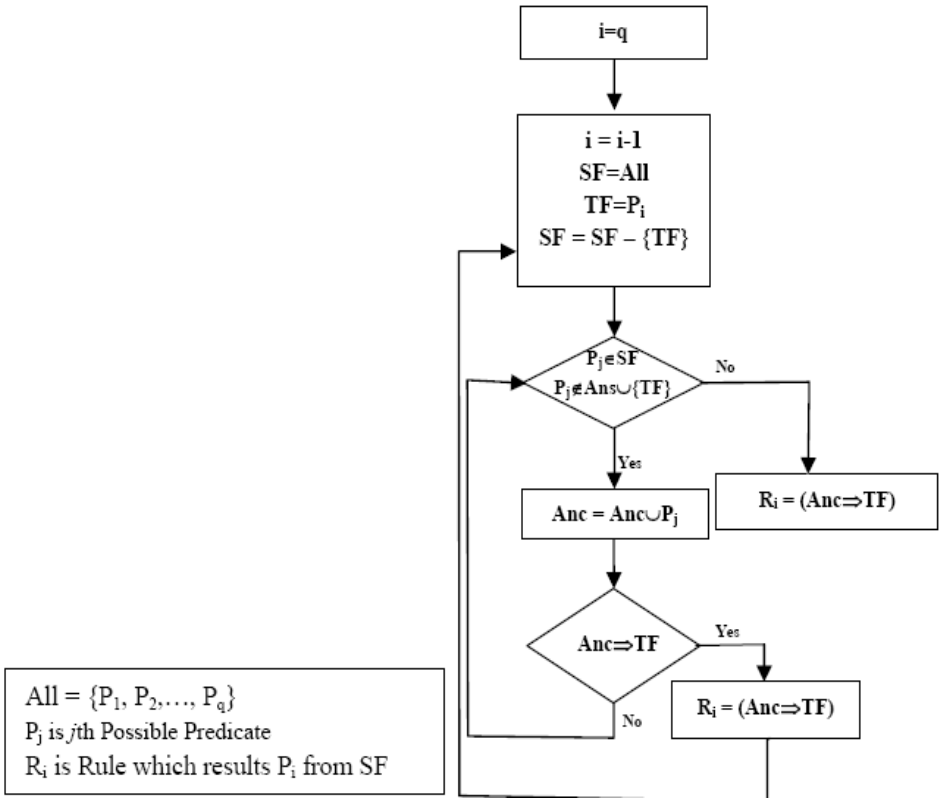

Figure 5. Association Rule Mining Model

checks might be 3518437208832. As can be seen, the time order is exponential. This time order is not acceptable at all. Of course we should pay attention that all these check is not handled because if for example P1 has relationship with Pq other sets of predicates which contains P1 will not be checked anymore and will not be interfered, but it does not affect the time order so much and overall time order is exponential.

Another issue which is worthwhile to emphasize again is that, in generating rules left-hand part or antecedent must be in the shortest state. For example if $P_1 \Rightarrow P_n$ rules such as P_1, $P_2 \Rightarrow P_n$ is not valuable.

Using decision tree

As discussed before, in decision tree, we can find a relationship between one attribute called goal or class and other attributes. In other words we can predict the goal by having other attributes. Two properties of decision tree are:

(I) Approximately lowest number of antecedents
(ii) Feasible highest confidence of the rule

These two properties might be so much helpful for generating association rules by the contribution of decision tree because our main purpose is to have some rules with lowest number of antecedents with high confidence. For employing this technique we should consider each predicate as the goal and try to capture the predicates which result our goal. Consider we have predicates P_1, P_2, P_3, … , P_q. We start with a predicate such as P_1 as our goal or class. We make the decision tree for P_1 and then we try to figure out other predicates which defined P_1's result. As mentioned before if we go upward from leaves to root in obtained tree, they can be rules which show when P_1 is true and when it is false. Then we obtain the P_2's tree and so forth.

Using decision tree for obtaining the rules is so much faster than normal association rule mining. Because by employing decision tree we do not have to check all the predicates to each other but we split our source into some smaller subsets and then classify each predicate lonely.

Figure 6 demonstrates the process of Decision Tree Learning Model box in the Figure 3.

CONCLUSION AND FUTURE WORK

Here in this subject, different properties of program are checked in different program points. These properties usually show the behavior of that program point. Invariants are always true with a specified confidence so they can not represent those behaviors which are true

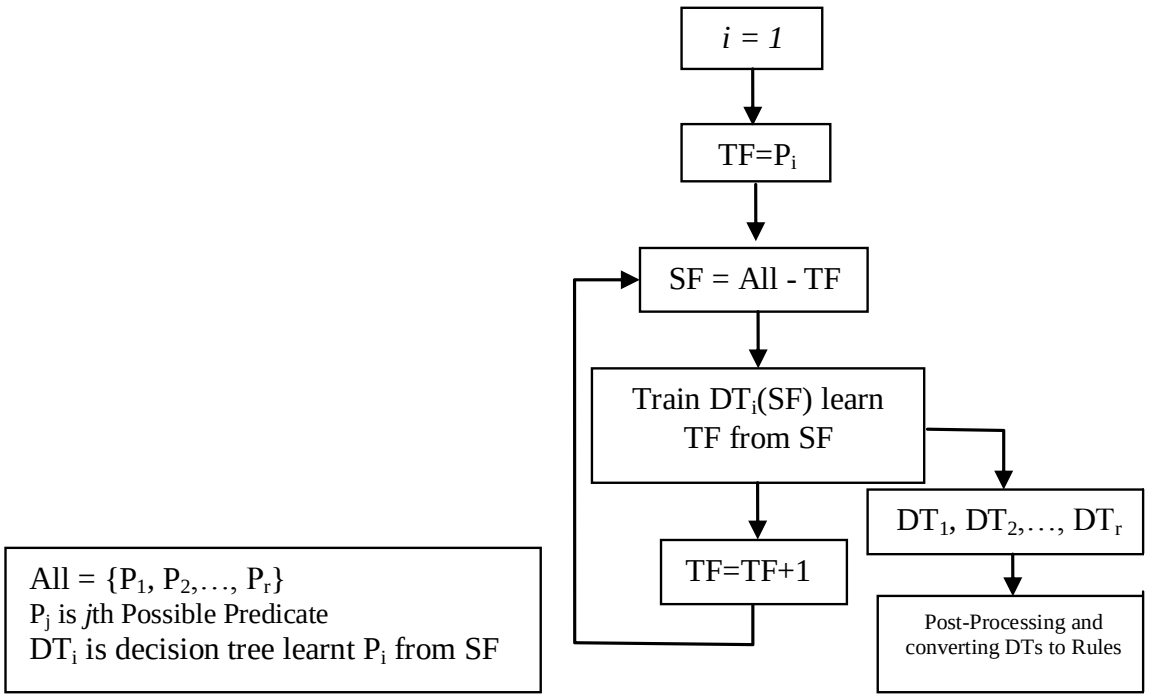

Figure 6. Decision Tree Learning Model.

with assuming a condition. Conditional invariant solve this problem because via these kinds of invariants we would have predicate which are true while another predicate is also true. We tried to generate the association rules by ordinary association rule mining and by repeated checks. The time order in via these methods is exponential and is not acceptable at all. So we brought up the decision tree and try to obtain rules with this technique. We check each predicate as a goal and try to find the related predicates which result to the goal.

For future work, we can try to obtain the rules by the help of Bayesian network. Bayesian network is another data mining technique which creates a model to predict the value of the goal based on other input variables. Bayesian networks are very efficient when the features (or predicates in our work) do not have correlation. By Bayesian network and its related methods we can detect the conditional invariant from presented predicates in each program point.

REFERENCES

Boshernitsan M, Doong R, Savoia A (2006). From Daikon to Agitator: Lessons and challenges in building a commercial tool for developer testing. ISSTA pp. 169–179.

Csallner C (2008). DySy: Dynamic symbolic execution for invariant inference. In: Proceedings of ICSE.

Ernst MD (1999). Dynamically discovering likely program invariants to support program evolution. In Proceedings of ICSE 1999, ACM pp. 213–224.

Ernst MD, Cockrell J, Griswold WG, Notkin D (2007). Dynamically discovering likely program invariants to support program evolution. IEEE TSE 27(2):99–123.

Ernst MD, Perkins JH, Guo PJ, McCamant S, Pacheco C, Tschantz M S, Xiao C (2006). The Daikon System for Dynamic Detection of Likely Invariants, Science of Computer Programming.

Hangal S, Lam MS (2002). Tracking down software bugs using automatic anomaly detection. In: ICSE, pp. 291–301.

Jones N D, Nielson F (1995). Abstract interpretation: A semantics-based tool for program analysis. In Abramsky S, Gabbay DM, Maibaum TSE (Year). Editors, Handbook of Logic in Computer Science, Oxford University Press, 4:527–636.

Krkay I, Brunx Y, Popescuy D, Garciay J, Medvidovic N(2010). Using dynamic execution traces and program invariants to enhance behavioral model inference. ICSE NIER.

Vanmali M, Last M, Kandel A(2002). Using a neural network in the software testing process. Int. J. Intell. Syst. 17(1):45–62.

WeiB B (2007). Inferring invariants by static analysis in KeY. Diplomarbeit, University of Karlsruhe.