Fault Localization for Data-Centric Programs

Diptikalyan Saha
IBM Research - India
diptsaha@in.ibm.com

Mangala Gowri Nanda
IBM Research - India
mgowri@in.ibm.com

Pankaj Dhoolia
IBM Research - India
pdhoolia@in.ibm.com

V. Krishna Nandivada
IBM Research - India
nvkrishna@in.ibm.com

Vibha Sinha
IBM Research - India
vibha.sinha@in.ibm.com

Satish Chandra
IBM T. J. Watson Research Ctr
satishchandra@us.ibm.com

ABSTRACT

In this paper we present an automated technique for localizing faults in data-centric programs. Data-centric programs primarily interact with databases to get collections of content, process each entry in the collection(s), and output another collection or write it back to the database. One or more entries in the output may be faulty. In our approach, we gather the execution trace of a faulty program. We use a novel, precise slicing algorithm to break the trace into multiple slices, such that each slice maps to an entry in the output collection. We then compute the semantic difference between the slices that correspond to correct entries and those that correspond to incorrect ones. The "diff" helps to identify potentially faulty statements.

We have implemented our approach for ABAP programs. ABAP is the language used to write custom code in SAP systems. It interacts heavily with databases using embedded SQL-like commands that work on collections of data. On a suite of 13 faulty ABAP programs, our technique was able to identify the precise fault location in 12 cases.

Categories and Subject Descriptors

F.3.2 [**Semantics of Programming Languages**]: Program Analysis; D.2.5 [**Testing and Debugging**]: Debugging aids

General Terms

Algorithms, Languages

Keywords

Automated debugging, Data-centric languages, Slicing, Semantic differencing

1. INTRODUCTION

Bug resolution is an important activity in any software maintenance project. Bug resolution for problems reported on applications already in use (in production) has two main implications. First, a client has discovered a bug in the field, and so it needs to be fixed urgently. Therefore, it is important to have good tool support that can help the programmer debug the program as quickly as possible, especially when the person who is debugging the code is not the programmer who originally wrote the code. Second, the bug has surfaced despite the fact that the code has been tested and has probably been running in the field for some time. That means it is likely to be an unanticipated corner case in otherwise correct code, and automated tools can focus their effort on identifying such corner cases.

The techniques presented in this paper, were developed to aid in faster resolution of programming errors reported for ABAP programs. ABAP is a widely used propriety language used in SAP-ERP systems, and is heavily data-centric in the tradition of PL/SQL and COBOL. Data-centric programs process large collections of data that typically originate from a database. ABAP contains both imperative and declarative syntax. The declarative syntax is similar to SQL and allows developer to do complex operations on collections of data. Henceforth, we refer to this declarative SQL-like commands in ABAP as database statements.

Figure 1 shows a sample program written in ABAP and Figure 2 explains the syntax of each of the commands. This program represents a business application that creates a report of orders placed by different customers. Figure 3(a) shows a sample input and output data combinations from the program. Each correct output row is followed by a $\surd$. The output row that is considered incorrect is marked with a $\times$ and is followed with the expected correct output. The `OrderTab` table contains the order details such as customer who placed the order `CstId`, item ordered `ItemId`, its price `Price` and year in which the order was placed

```
SELECT CstId ItemId Price Year from OrderTab INTO itab
SELECT CstId Discount Year from DiscountTab INTO stab

SORT itab CstId ItemId
DEL from itab where Year <= CurrentYear - 2.
LOOP AT itab INTO wa
  AT NEW CstId
    amount=0
  ENDAT
  amount = amount + wa.Price
  READ stab INTO fa WHERE CstId = wa.CstId
  IF subrc = 0
    amount = amount - fa.Discount
  ENDIF
  AT END CstId
    WRITE CstId amount
  ENDAT
ENDLOOP
```

Figure 1: Sample ABAP **program**

Command	Description
`SELECT`	project selected columns from a persistent database table to an internal table in the program
`SORT`	sorts the specified internal table on specified key(s)
`DEL`	deletes rows from a table that satisfies the condition
`LOOP`	iterates over an internal table, reading one row at a time into a local record
`AT NEW` `(AT END)`	a predicate that is true for a given row and field name(s) when the row is the first (last) one in the table or when the field's value in the current row is different from the previous (next) row
`READ`	selects a row from table based on the `WHERE` clause. If more than one row matches, the last row is returned
`WRITE`	prints the specified data
`wa.f`	field `f` of structure `wa` (actual ABAP syntax `wa-f`)

Figure 2: Basic ABAP **syntax**

(a) `OrderTab/itab`

CstId	ItemId	Price	Year
1	I1	10.0	2010
1	I2	10.0	2011
2	I3	10.0	2011

`DiscountTab/stab`

CstId	Discount	Year
1	2.0	2010
2	3.0	2011

Output

CstId	Amount	
1	16.0	× [= 10.0 + 10.0 - 2.0 - 2.0] [18.0✓= 10.0 + 10.0 - 2.0]
2	7.0	✓[= 10.0-3.0]

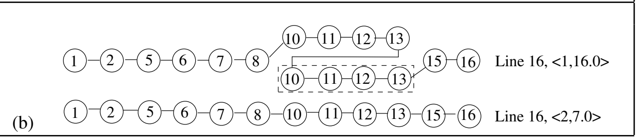

Figure 3: (a) Input and output for the ABAP **program illustrated in Figure 1. (b) Dynamic slices for each output row.**

`Year`. The `DiscountTab` table contains the discount applicable per customer per year. The output shows for each customer the total order amount. At the code level, the program first reads the input data from `OrderTab` and `DiscountTab` into internal tables `itab`, `stab` (lines 1,2), sorts table `itab` (line 4) and deletes records older than year `2010` (line 5). The `CurrentYear` variable is a parameter to the program and has value `2011`. The program then loops over the contents of `itab` (line 6), sums up the `Price` (line 10), subtracts any

(a) `OrderTab/itab-pre`

CstId	ItemId	Price	Year
1	I1	10.0	2009
1	I2	10.0	2011
2	I3	10.0	2011

`DiscountTab/stab`

CstId	Discount	Year
1	2.0	2011
2	3.0	2011

`itab-post`

CstId	ItemId	Price	Year
1	I2	10.0	2011
2	I3	10.0	2011

Output

CstId	Amt	
1	8.0	× [= 10.0-2.0] [16.0✓ = 10+ 10.0-2.0-2.0]
2	7.0	✓

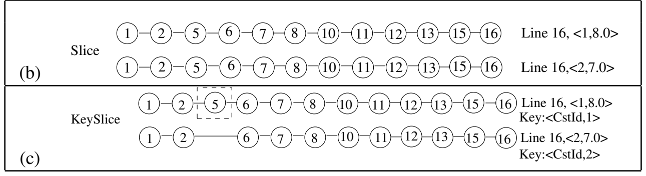

Figure 4: (a) Input and output for the ABAP **program illustrated in Figure 1. (b) Dynamic slices for each output entry. (c) Key based slices for each output row.**

(a) `OrderTab/itab`

CstId	ItemId	Price	Year
1	I1	10.0	2010
2	I3	10.0	2011

`DiscountTab/stab`

CstId	Discount	Year
1	2.0	2009
1	3.0	2010
2	3.0	2011

Output

CstId	Amount	
1	8.0	× [= 10.0 - 2.0] [7.0 ✓= 10.0 - 3.0]
2	7.0	✓

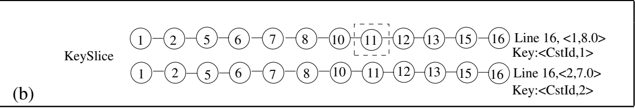

Figure 5: (a) Input and output for ABAP **program illustrated in Figure 1. (b) Key-based slice for each output row.**

relevant `Discount` (line 13) and prints out the total (line 16). One output entry is generated for each unique `CstId` present in the input `OrderTab`.

There are several challenges involved in developing fault localization techniques for these types of programs. The first challenge is that the analysis needs to handle both the imperative and declarative parts of the language. What should be the correct representative semantics for the different SQL-like commands in an ABAP program? Another challenge is that the analysis needs to be data driven, as the behavior of a command is very often dependent on the underlying data. Thus the same command at the same program point may run without any problems for most of the data and yet may throw an exception or generate incorrect output for some other data. As we shall see later in the paper, for the example shown in Figure 1, depending on the combinations of data available in the `OrderTab` and `DiscountTab` tables, three different bugs can manifest themselves in the output.

Problem resolution methodology. A large body of past work on fault localization relies on the usage of program slicing [1]. The basic idea in these techniques is that, when a program computes a correct value for a variable `x` and an incorrect value for variable `y`, the fault is likely to be in statements that are in the slice with respect to `y` but not in the slice with respect to `x` [14]. Similarly, if a program computes correct value for a variable `x` in a particular test case and computes incorrect value in another test case, then the potentially faulty statements would be in the difference between the slices of these two program executions [2, 22]. Our problem resolution methodology is motivated by these prior works and attempts to apply the same in the domain of data-centric programs.

One of the key challenges in applying any of the slicing based fault localization techniques in real world debugging scenarios is the lack of significant number of test cases that show the correct behavior of the program. These test cases are needed to collect the execution traces of correct examples that can be differentiated with the trace for the incorrect execution as reported in the bug description. However, in the context of data-centric programs, we can leverage the fact that a single run of the program yields an execution trace that can in turn be split into multiple independent slices, each of which is responsible for a single record in the output. This is because these programs typically loop over the input data records, aggregate the input depending on certain key fields and generate an output record per key value. In the case of the example discussed above, the key field was `CstId`. A defective program writes incorrect values for one or more key values. Further, if a user specifies that certain output rows are incorrect and we assume that the rest of the output is correct, we can associate a "incorrect" or a "correct" tag with each slice corresponding to each row of output. We can then compute the *difference* between the correct and incorrect slices to discover the statements that are potential sources of bugs.

To identify the bug reported in Figure 3(a), denoted by the output row postfixed with $\times$ mark, we first collect the dynamic trace by running it on the input that reveals the problem. We split the trace into multiple slices by applying dynamic slicing starting at each *instance* of line 16 (`WRITE`) in the execution trace. Figure 3(b) shows the slices. We then compute the difference between the slices to identify that line 10 through 13 are executed twice in the first slice versus once in the second slice. The lines are highlighted as fault inducing statements. This finding relates to the problem that the discount should have been given only once per customer. In the corrected code (done manually by the programmer), lines 11 through 14 were moved inside the `AT END` block (after line 15).

Key-based Slicing. It may appear that we can always generate incorrect and correct slices, and apply a differencing technique to this setting. However, this is not the case, as the dynamic slice based on simple data and control dependence may not differentiate between a correct and an incorrect execution slice. Consider the bug reported for the same program using input data in Figure 4(a). In this case the user was expecting to see the `amount` value as 16.0 for `CstId = 1` (considering that we have the code as it is in Figure 1). However, the code is deleting all order records that are older than 2 years (line 5). This means the conditional in `DELETE` statement in Line 5 is incorrect or incomplete. The slices for both the output rows are the same as shown in Figure 4(b) and hence differencing will not be able to identify any faulty lines. We resolve this by enhancing the existing dynamic slicing with the introduction of a key in the slicing predicate. The intuition for this is as follows. Usually in a data-centric program, as part of the output record, one or more input fields are written that together act as the identifier (or key) for the data and is unchanged from the input to output. For our example this field is the `CstId`. So, besides using an execution of a particular statement as the predicate for our slice, we also use the value of this key field as our slicing criterion. Figure 4(c) shows the key-based slices. Line 5 is only showing a side-effect for records in `OrderTab` with `CstId = 1`. The values in the table `itab` before (`Pre`) and after statement #5 (`Post`) are shown in Figure 4(a).

Once we do a differencing on these key-based slices, our technique highlights line 5 as the potential fault inducing code, as it effects the incorrect slice, and not the correct slice.

Semantic Differencing. In some cases even key-based slicing is not enough to differentiate between slices. Consider the input data in Figure 5(a) for the same program. Here the key slices for both the output records are the same. The actual difference is in the behavior of line 11 in its two different executions. The `READ` statement in ABAP returns only a single matching record. If there exist multiple records that match the selection criterion (`WHERE` clause), then it returns the first one. For `CstId = 1`, line 11 would need to select from two records, while for `CstId = 2`, there is only one matching record. Hence, the behaviors of the `READ` statement for these two keys are different. Our semantic differencing algorithm identifies such statements in the execution trace where different behavior of the command/statement have been exercised in the correct and incorrect key slices and highlights them as potential faults. For the example application, the fix is to change the `READ` (stmt 11) as `READ stab INTO fa WHERE CstId = wa.CstId and Year = wa.Year` such that appropriate year-wise discount is availed.

Novelty and Contributions. Key-based slicing and semantic differencing are novel generalizations of previous work on fault localization based on comparing program executions. Prior work in this area mostly takes into account just *whether* a statement appears in one slice but not in another one; but it does not take into account the *manner* in which it appears in a slice. Key-based slicing also takes into account the relevance of execution of statements to specific keys that were used in creating those slices; this is important in the context of data-centric programs. Semantic differencing contributes yet another attribute of statement execution, where the behavior of statement execution on specific input data is taken into account. We believe that our work is among the first to successfully adapt and extend fault-localization techniques to data-centric programs that occur in an industrial setting.

We have built a tool that takes an execution trace of an ABAP program and an indication of the buggy part of output. The tool then offers diagnosis of faults based on the techniques presented in this paper. In most cases, the diagnosis if found, was a single statement (for semantic differencing, we were looking for only a single statement difference).

Our experiments with the tool show that the above differencing techniques were able to accurately localize faults in 12 out of 13 ABAP programs provided to us by our colleagues in IBM Global Business Services. These ABAP programs were either old versions of some programs where there was a known defect that has since been fixed, or were programs in which a realistic defect was seeded by them. A baseline version of the tool that did not incorporate the generalizations of differencing mentioned above was effective only in 3 out of the 13 programs.

Organization. In the next section we give the details of our slicing algorithm. In Section 3, we elaborate on the differencing algorithm. In Section 4, we give the results of running our analysis on field bugs. Section 5 describes the related work and we conclude in Section 6 with some additional discussion on future work.

2. DYNAMIC SLICING

The efficacy of our differencing algorithm (cf. Section 3) depends on the accuracy of slicing. In this section we present an algorithm that performs precise backward dynamic slicing on execution traces containing database commands. Each slice is computed starting from a statement occurrence ($Line^{seq}$) that produces a row in the output, and on a set of variables (V), occurring in the statement. This constitutes the slicing criteria $\langle Line^{seq}, V\rangle$. We refer to the slices producing incorrect output as incorrect slices, and to the slices producing correct output as correct slices. The slices obtained in this way are analyzed with a differencing algorithm to reason about the possible faults present in incorrect slices.

Traditional slicing algorithms are oblivious to the rows and fields of tabular data (i.e., table accesses are treated as table[*].*). This results in overly conservative slices that are not effective in differencing data-centric programs. We obtain a row and field-sensitive algorithm based on the existing techniques of handling non-scalar data [29, 28, 19]. Our algorithm is built on top of precise (to the field-row level) dependency information, which can be obtained from the semantics of the statement and the data present in the execution trace. For example, the effect of a `DELETE` statement on a table is modeled such that it is possible to know the shift of indices of all the rows. The data effect of the `DELETE` statement will contain all those rows, whose table index is changed by the `DELETE` statement. In general, for each compound statement, its final effect is represented by a set of assignment statements, and a set of def-use pairs are identified from the assignment statements.

$1^1, 2^2, 4^3, 5^4, 6^5, 7^6, 8^7, 9^8, 10^9, 11^{10}, 12^{11}, 13^{12}, 14^{13}, 15^{14}, 16^{15}, 17^{16}, 18^{17}, 6^{18}, 7^{19}, 8^{20}, 9^{21}, 10^{22}, 11^{23}, 12^{24}, 13^{25}, 14^{26}, 15^{27}, 16^{28}, 17^{29}, 18^{30}$

(a) Execution Trace: List of $Line^{SequenceId}$

I^q	Statement	ϕ
16^{28}	WRITE	amount
15^{27}	AT END CstId	itab.length
13^{25}	amount = amount - fa.Discount	amount, fa.Discount, itab.length
12^{24}	IF subrc = 0	subrc, amount, fa.Discount, itab.length
11^{23}	READ stab into fa where CstId = wa.CstId	amount, stab[1].Discount, wa.CstId, itab.length
10^{22}	amount = amount+wa.Price	amount, wa.Price, itab.length stab[1].Discount, wa.CstId
8^{20}	amount = 0	wa.Price, itab.length, stab[1].Discount, wa.CstId
7^{19}	AT NEW CstId	wa.Price, itab.length, stab[1].Discount, wa.CstId itab[0].CstId, itab[1].CstId
6^{18}	LOOP at itab into wa.	itab[1].Price, stab[1].Discount, itab[1].CstId, itab.length itab[0].CstId
5^4	DEL from itab where ..	itab[2].Price, stab[1].Discount, itab[2].CstId, itab.length itab[1].CstId
2^2	SELECT .. from DiscountTab into stab	itab[2].Price, itab[2].CstId DiscountTab[1].Discount, itab.length, itab[1].CstId
1^1	SELECT .. from OrderTab into itab	OrderTab[2].Price DiscountTab[1].Discount, OrderTab[2].CstId, OrderTab[1].CstId

(b) Update of Data Dependency Set

Figure 6: Example: Slice Computation

The execution trace of the example in Figure 1 for input data specified in Figure 4, is presented in Figure 6(a). Figure 6(b) shows the update of the data dependency information after including each statement occurrence in the slice that is computed for the amount variable in the second row of the output, generated at statement occurrence 16^{28}.

Note that, due to shift of indices and its effect on the length of the table, the statement 5^4 is included in the slice. The DELETE statement actually does not affect the computation that is done for the second row of the output, as the addition is not performed on elements which have Year value $\leq$ 2009. Thus inclusion of DELETE statement in this slice makes it imprecise. Whereas, the slice computed with criteria $\langle 16^{15}, \{amount\}\rangle$ should include the DELETE statement, as the deleted rows, affect the computation, performed to compute the sum at 16^{15}.

2.1 Key-based Slicing

As discussed above, the row and field-sensitive slicing algorithm can result in imprecise slices. An important question to answer is, when does a statement occurrence become a part of the slice? Our dynamic slices represent the computations that affect specific rows in the output. A statement occurrence is not considered to be a part of the slice, if its absence has no effect on the *computation* of the variable values in the output row associated with the slice.

If a statement occurrence only affects the position of a row in the output, and not the values, we do not consider the statement to be part of the slice. In our experience it was the content of the row and not the order in which they occurred in the output that was more commonly observed to be at fault.

To determine whether a statement occurrence is effecting the variable values in the slicing criteria we need to check two conditions: ($C1$) if the statement occurrence is performing any operation that defines a variable in the dependency set, ($C2$) if the absence of the statement can change the dependency set in terms of addition or deletion of elements. If either of the conditions $C1$ or $C2$ is true, the statement is added to the slice. The statements that could change the dependency set by adding/deleting elements are called *db-change* statements. The DELETE statement in our running example does not satisfy the condition $C1$ for the slices corresponding to either row of the output. The condition $C2$ is satisfied for the first row of output: in the presence of the DELETE statement, the dependency set in the slice corresponding to CstId=1 contains OrderTab[1].Price (as shown in Figure 6(b)), whereas in the absence of DELETE statement, that slice would have contained OrderTab[0].Price and OrderTab[1].Price. However, the condition $C2$ is not satisfied for the DELETE statement in the slice corresponding to the second output (corresponding to CstId=2), as in the absence of the DELETE statement, the dependency set would have still contained only OrderTab[2].Price.

The condition $C1$ is straightforward to check, and it is assumed that this check is precisely done as a part of field-row-sensitive algorithm for the non-db-change statements. We now present a criteria to check the condition $C2$ on the db-change statements. The purpose of this criteria is to remove a statement from the slice which otherwise would be included in the slice by a field-row-sensitive algorithm which takes conservative decision for inclusion of such database statements in the slice. The criteria has two parts 1) key-value condition and 2) sequence condition.

The first condition can be motivated by revisiting the running example. It is evident that the slices with respect to the criteria $\langle 16^{28}, \{\texttt{amount}\}\rangle$ and $\langle 16^{15}, \{\texttt{amount}\}\rangle$ have association with key-value pairs $\langle \texttt{CstId}, 2\rangle$ and $\langle \texttt{CstId}, 1\rangle$ respectively. With this association, whether to include the statement occurrence 5^4 can be easily checked by determining whether the deleted rows *match* the key-value pairs. In general, we say a row r matches a key-value pair (k, v) if the value of key k in row r is equal to v. A statement is not included in the slice if any change performed by the statement (such as added or deleted rows) does not match with the key-value pairs.

However, even if the change performed by a statement matches the key-value pair condition, the statement may not have any effect on the slice criteria, and thus, may not be included in the slice. The *sequence condition*, described next, is a condition that applies in the common GROUP-BY/ORDER-BY pattern in database processing. Consider an example pre-state and post-state of the DELETE statement 5^4.

```
      itab(Pre)                      itab(Post)
CstId ItemId Price Year        CstId   ItemId  Price Year
  1      I1    10.0 2009         1        I2      10.0  2011
  2      I1    5.0  2009         2        I3      10.0  2011
  1      I2    10.0 2011
  2      I3    10.0 2011
```

Here the DELETE statement does not affect the slicing criteria $\langle 16^{28}, \{\texttt{amount}\}\rangle$ as only the position of output row has changed, and not its values. The deleted rows could possibly effect the aggregated computations only if in the pre-state they were *adjacent* to the remaining undeleted rows. Sequence condition states that the table elements that are used to compute the values of the variables in the slicing criteria, are accessed in sequence. If the sequence condition is not met after key condition is met, the statement is not included in the slice. Note that the example does satisfy the key-value condition, so the sequence condition offers extra precision by eliding the DELETE statement from the slice.

There are multiple ways to identify key fields for association.

- The key fields may be specified by the user. This is not an unrealistic assumption. Observations on actual bug reports, indicate that the users find it more intuitive to describe the problem using the key fields associated with the row, i.e., they are more comfortable in specifying something like, "the unbilled amount for customer 2 is wrong", rather than, "the unbilled amount in the second row of the table is wrong"
- Fields in the internal table that are not modified before being written out into the output.
- Fields in the internal table that are used to select the rows to operate on (in `SELECT`, `DELETE`, `MODIFY`, etc.).

To summarize, we introduced, the key-value, and the sequence conditions, to more precisely evaluate condition $C2$ for verifying the inclusion of compound statements that process table elements, into a slice. It is important to describe here that as we move backward in the slice, the checking of one or both of these conditions may no longer remain applicable. For instance, say we encounter a statement such as `SORT`, which sorts the table elements based on the key-values, then as we move backward in the slice, we expect the table elements, to not be in sequence w.r.t. the key-values. Similarly, we may encounter a statement such as `APPEND`, backward to which, the key-value may not exist in the table elements.

The details of our key-based slicing algorithm are presented in Figure 7.

- Lines 1-6, 8-11, 15-16, 20-21 represent a basic field and row sensitive dynamic slicing algorithm.
- Function `get_def_use(s)` returns a set of def-use pairs representing def-use relationship between each pair.
- Function `field_row_sensitive_inclusion_check` performs checks like nonempty intersection of dependence set and defs in duSet to check data dependency and control dependency to include a statement occurrence in the slice.
- Once a statement is identified to be included in the slice, the dependency set is updated by removing the def and including use of each def-use pair in duSet (Line 16).
- Above the check done by field-row sensitive check, we add the key-value and sequence checking (Lines 12-13) to determine whether a statement occurrence chosen by the basic field-row sensitive algorithm will be part of the slice. These checks are only done for db-change statements.
- Function `check_kv_assumption` checks that all the non-scalar elements in the dependency set satisfy the key-value pairs
- Function `check_seq_assumption` checks that all the elements in the dependency set are in sequence, so that any statement occurrence which has a change outside this sequence will not have any effect for a particular key value.

Note that to highlight the interesting part of the algorithm we do not present the slicing algorithm in terms of dynamic dependence graph [1], used to express the data and control dependencies in the execution trace.

While, it is possible to give a necessary and sufficient condition to check the condition $C2$, the evaluation of such condition is not scalable, and thus not ideal for practical purpose. In this paper, we therefore restrict the presentation to the practical and scalable technique of key-based slicing.

```
1  Function KeySlice ($I^q, V, S$)
2  Input: int $I^q$ /*sequenceid*/, Set $V$ /*set of vars */
3  Output: List $S$ /* List of statements */
4
5  Set $\phi = V$
6  int $i = I^q$
7  key—value—pairs $kvp = kvpairs$// computed or user provided
8  while $i$>0
9    $s$ = stmt.get($i$)
10   $duSet$ = get_def_use($s$) // statement specific
11   if(field_row_sensitive_inclusion_check(s,$duSet$,$\phi$)
12    $\wedge$(!key_assumption_valid || (check_kv_constraint($kvp$,$s$)
13     $\wedge$(!sequence_valid||check_seq_constraint($kvp$,$s$,$\phi$))))
14     )
15      $slice$.add($s$)
16      $\phi$ = update_dep_set($\phi$,$duSet$)
17      key_assumption_valid = check_kv_assumption($\phi$,$kvp$)
18      if key_assumption_valid
19        sequence_valid = check_seq_assumption($\phi$,$kvp$)
20   $i--$
21 return slice;
22
23 check_kv_constraint $kvp$, $s$
24   if $s$ is not a db—change statement
25      return true // condition C1
26   if any change by $s$ satisfies $kvp$
27      return true
28   else
29      return false
30
31 check_seq_constraint $kvp$, $s$, $\phi$
32   $c$ = changes by $s$ that satisfies $kvp$
33   if check_sequence_assumption ($\{c, \phi\}$,$kvp$)
34     return true
35   else
36     return false
37
38 check_kv_assumption $\phi$, $kvp$
39   for each structure variable $v.f$
40     if $f \in kvp.keySet$
41        if value of $v.f$ != $kvp$.getValue($f$)
42             return false
43   return true
44
45 check_seq_assumption $\phi$, $kvp$
46    for all $itab$
47       for any key $f \in kvp.keys$
48         $I$ = set of all $i$ s.t. $itab[i].f \in \phi$
49         if elements of $I$ are not in sequence
50            return false
51    return false
```

Figure 7: Key-based Slicing

3. SLICE DIFFERENCING

In this section we present the fault localization algorithms. We extend an existing approach to identify faults–to find the difference in behavior between *slices* that produce correct and incorrect output [14], and localize the faults at difference points. The novelty of our technique lies in the way we compute the differences between slices.

We apply two main differencing techniques. The first technique is relatively simple, where any control difference between two slices is determined (cf. Section 3.1). Such difference analysis typically determines the difference in sequences of statement occurrences in correct and incorrect slices. However, it is possible that sequence-based differencing may not produce any difference as the same sequences of statements can be present in both the slices. The second technique is particularly novel as it tries to find out the behavioral differences between two slices–finding statements that are present in both correct and incorrect slices but shows difference in their semantic behaviors (cf. Section 3.2). Note that, these differencing techniques are useful to localize faults when certain rows (not all) in the generated output have incorrect results.

3.1 Sequence-based Differencing

The main aim of sequence-based differencing is to identify statements that contribute towards computation of the incorrect result and do not contribute to the computation of at least one correct result. To realize the presence of such statements we perform a two step process: 1) grouping correct and incorrect slices into equivalence classes, and 2) perform pair-wise differencing between the representative elements of the correct and incorrect equivalence classes.

In data-centric programs it is common to find the slices containing traces of same set of statements, but differing in the number of iterations of the some loops. Thus, while creating equivalence classes in correct and incorrect slices, we combine two slices into the same class if they are exactly same or if they have different number (≥ 1) of iterations of the same statements in a loop. Generating equivalence classes in correct and incorrect slices reduces the number of pair-wise comparisons required to find differences between slices.

While sequence-based differencing of two slices, one from a correct equivalence class and another from an incorrect equivalence class, any difference of statements executed in the sequences of statement occurrences is noted. However, due to common nature of loop-iteration differences (such as, different number of loop iterations), these differences are given lower priority among all-sets of pair wise differences. The actual algorithm of sequence-based differencing in presence of loops is not presented here for brevity.

3.2 Semantic Differencing

As discussed above, an incorrect slice may not show any important difference compared to correct slices in sequence-based differencing. This is possible if a statement exhibits different "behaviors" in two slices due to the nature of input data to the statement. We call such difference in behavior as *semantic difference*. In this section, we illustrate such differences, and present algorithms to detect semantic differences. To the best of our knowledge, this is the first attempt to perform fault localization based on differences in the semantics as seen during the execution of program statements.

Consider the example program presented in Figure 1 and the corresponding test case shown in Figure 5(a). In this example corresponding to the `CstId=1` there are two rows in `stab`. In the `READ` statement at Line 11 when the selection condition is satisfied with multiple rows, then first (lowest index) matching row is selected based on ABAP semantics. In the example , the first row is selected with `Discount=2.0`, which results in output `8.0` instead of the correct output `7.0` corresponding to `CstId=1`. In this example, same slices exist for the two output rows as shown in Figure 5(b). So sequence-based differencing does not find any difference between slices.

In semantic differencing, we assume that there must exist a faulty statement in the program that appears both in the correct and incorrect slices, such that it *fortuitously* exhibits the intended semantics in the correct slices, but deviates from the intended semantics (based on programmer intent) in the incorrect slice. Remember that the fortuitous correct behavior in the correct slices is specific to the particular input data.

The important question is, how do we tell if a statement has deviated from its intended semantics? After all, programmers do not provide assertions after each statement to verify if the effect of the statement just executed is as they expected. We only know that the final effect, i.e. the intended output is present in the correct slices, and the unintended output in the incorrect slice.

In this paper, we use two kinds of heuristics to find the first statement in the incorrect slice which shows such a deviation.

Statement Type	Target Corner-case Difference
`READ` from itab into wa where C	Multiple/Unique rows are satisfied with C
`APPEND/INSERT` lines of jtab from idx1 to idx2 to itab.	The number of rows appended/inserted is different from idx2 - idx1 + 1
`INSERT`	The inserted row makes certain set of rows with same keys non-contiguous.
`APPEND/INSERT`	The inserted/appended row makes sorted data unsorted

Assignment
`MOVE`
`MOVE-CORRESPONDING`
transporting clause
	overflow
	overwriting same value

`LOOP` at itab. ... `ENDLOOP.`	the statement within loop contains delete from itab.
`AT NEW/END`	Whether `AT NEW` and `AT END` both is true for a single row.
`DEL ADJ` from itab comparing f1..fn.	the table is not sorted with f1..fn
`DELETE` from itab where C	Multiple/Single row selected by C.
selection condition $a \leq b$	$a = b$
selection condition $a \geq b$	$a = b$

Figure 8: Corner-case Differences

Corner Case Differencing. The first method of semantic differencing is called *corner-case differencing*. The semantics of some of the statements are classified into two separate categories: a normal case, and one or more corner cases. For example, in a `READ` statement, the `WHERE` condition could match multiple rows, or just one row. Since the first matching row is returned by the `READ`, the matching of just one row among several candidates is a corner case. A table of corner and non-corner cases for several statements is given in Figure 8. Given a trace, we can tell if a statement executed in a corner-case manner, or in a normal case manner.

Intuitively, this technique exploits the fact that most errors (typically seen in already tested code) occur due to non-handling of corner cases that are revealed in the incorrect slices, and not revealed in the correct slice. Key-based slicing determines whether there is *any* effect of a statement on a slice or not. Corner-case differencing tries to find out semantic difference of a statement with respect to correct and incorrect slices where the statement has *some* effect in both the slices.

In the example given in Figure 1 and data in Figure 5(a), we determine a corner-case difference in the `READ` statement, that in the case of correct slice only single row satisfies the selection condition, but in case of incorrect slice, the selection condition is satisfied with two rows. This difference is produced by looking at the semantics of `READ` statement and particularly evaluating the corner-case aspect in two slices. Note that, in this case, the difference in behavior of the `READ` statement is indeed this particular behavior found using behavioral differencing. The presence of this behavior (multiple satisfied selection) in `READ` is always a problem, as programmer may intend to get the first matching row always, and may not agree to specify an extra field in selection condition which increases the overhead of the operation. The fact that this difference in behavior showed in correct and incorrect slices is the key observation. Several other checks are presented in Figure 8.

Mutability Differencing. Our second method of differencing is called *mutability differencing*. Mutability differencing tries to make an intelligent guess on the correct form of the statement such that it produces different behavior than the observed behavior in the

Statement	Mutation
key constraint C in `READ/SELECT/DELETE` `APPEND/INSERT/MODIFY`	addition of key constraint deletion of key constraint
non-key constraint C in `READ/SELECT/DELETE/LOOP` `APPEND/INSERT/MODIFY`	modify C with post-condition imposed by a correct and all incorrect slices.
List of fields in `SORT`	addition and deletion of fields based on key constraint and field names in `DELETE ADJACENT` statement on the same table
`AT NEW` f. `AT END` f. on change f1..fn.	addition and deletion of fields to f based on key constraint and based on fields used in sort statement on the same table
`MOVE-CORRESPONDING`	`DELETE` or `INSERT` `MOVE` of other fields by breaking `MOVE-CORRESPONDING` to a set of `MOVE` statements

Figure 9: Mutation Operators

```
1  DELETE itab where f2 <= 0
2  LOOP into itab into wa
3    AT NEW f1
4       min = MAXINT
5    ENDAT
6    IF wa.f2 < min
7      min  = wa.f2
8    AT END of f1
9       WRITE wa.f1, min
10   ENDAT
11 ENDLOOP
```

`itab` (init)

f1	f2
1	-1
1	1
1	2
2	0
2	3
2	2

Output

1	1	✓
2	2	× [0✓]

Fix: at Line 1
where `f2` $<$ 0.

Case (A)

f1	f2
1	-1
1	1
1	2
2	-2
2	3
2	2

1	1	× [-1✓]
2	2	✓

Fix: at Line 1
where `f2` < -1.

Case (B)

Figure 10: Example: Mutability vs. Corner-case Differencing

incorrect slice, expecting that the produced behavior after mutation is potentially same as the correct behavior. The important aspect of our technique is that we only consider mutation of the statements that do not change the behavior of the statement in the correct slice.

To diagnose the fault reported for Figure 5(a), we can also use mutability differencing. Consider the `READ` statement on line 11 in Figure 1. We apply a mutation to the `READ` statement at line 11 in Figure 1. `Year=wa.Year` is added to the selection condition in the `WHERE` clause. This is based on the observation that `READ` statement with key option is typically used as joining condition between two tables. There exist two common fields `CstId` and `Year` in the input tables `OrderTab, DiscountTab`. Any one of them or their combination could be used as the joining fields. However, in the buggy program only `CstId` is being used. After adding the common field `Year` in the joining condition, the analysis finds that the behavior in the correct slices remained same as same row is selected as before, but instead of selecting the record with values `<1, 2.0, 2009>` , the statement has now selected `<1, 3.0, 2010>` in the faulty slice. Indeed, the fix for this problem is the above change. A customer should only be given the discount applicable to the year in which the order was placed. Note that in general it is possible to get such a mutation after trying several number of mutations, and the applied mutation might not be the final fix, but could help to indicate the kind of fix to be made.

In general, the mutations we consider are based on identification of the key-fields. We identify the key-fields looking at similarity of field names in two joining tables (as above), matching fields names in sort, `DELETE ADJACENT`, and `AT NEW` statement, matching field names sort and binary search specification in `READ` statement. A complete list of patterns for ABAP language is not presented here for brevity. A list of mutation operators for different ABAP specific statements is presented in Figure 9.

Mutability differencing can be effective in cases where corner-case differencing is not. In Figure 10, minimum value for `f2` is computed for each distinct `f1` value. Before this computation, deletion occurred with a condition on `f2` values `f2` $\leq$ `0`. In case (A), 1st and 4th rows are deleted by the `DELETE` statement. In case (A), say we want the second output to be `<2, 0>` instead of `<2, 2>`, and fix we need is `f2` $<$ `0` in delete condition. Corner-case differencing can find this error as for `f1 = 2` the deletion of the row was done on the corner case of the condition `f2` $\leq$ `0`, but for `f1 = 1` the deletion was done on a non-corner case. Consider Case(B) where the first row reported is wrong as the expected output is `<1, -1>`. Here both the conditional evaluation for deletion went through a non-corner cases. Thus, behavioral difference will not be able to perform any difference here. Mutability difference, on the other hand, can intelligently mutate looking at the post-conditions that is required `f2` $\neq$ `-2` $\wedge$ `f2 = -1` $\wedge$ `f2` $\leq$ `0`.

However, mutability differencing is not strictly more powerful than corner-case differencing, as will be shown later in `ZROTC` experimental subject in Figure 12.

3.3 Extensions

The other kinds of bugs seen in data-centric programs are incorrect input data, unwanted rows, all incorrect rows, and missing rows. We briefly describe the approach we take for such description of bugs.

Missing rows. In this case the bug report contains the description of the missing rows in terms of their key-value pairs. Note that it is not possible to determine the slices corresponding to the missing rows as we cannot form slicing criteria for missing variables. However, the intuition that we follow here is that, if there is any row to be produced corresponding to the missing key-value pairs, their slices will be similar to the correct slices computed for some

```
1  SELECT * from zitr into table it_itr.
2  LOOP AT it_itr INTO wa_itr.
3      CLEAR : v_exrate.
4      v_exrate = 'READ_EXCHANGE_RATE'(wa_itr.waers)
5      IF subrc = 0.
6        IF v_exrate > wa_itr.exrate.
7          wa_itr.dispamt = wa_itr.amount * v_exrate.
8        ELSE.
9          wa_itr.dispamt = wa_itr.amount * wa_itr.exrate.
10       ENDIF.
11       WRITE:/ wa_itr—matnr, 30 wa_itr—dispamt.
12     ELSE.
13       wa_itr.dispamt = wa_itr.amount * wa_itr.exrate.
14       ' Missing write here
15     ENDIF.
16 ENDLOOP.
```

ZITR

matnr	amount	waers	exrate
1	50	USD	47
2	100	EUR	66
3	100	UK	75

Output

matnr	dispamt
1	2400
2	6600

Missing row: 3, 7500

Trace: 1^{1},2^{2},3^{3},4^{4},5^{5},6^{6},7^{7},11^{8},16^{9}, 2^{10},3^{11},4^{12},5^{13},6^{14},9^{15},11^{16},16^{17}, 2^{18},3^{19},4^{20},5^{21},13^{22},16^{23}

Figure 11: Example: Missing Row

existing rows. Thus, we analyze (by stepping backward) each correct slice, to find the first statement occurrence (`s`) from end which has the missing key-value pairs in its `use` set, but not in its `def` set. If `s` does not occur in a loop, it is reported as a suspect as the selection operation in the statement has filtered out the missing key-value pairs. If `s` is in a loop, we determine all peer occurrences of `s` in different loop iterations, such that each of them (say `s'`) has the missing key-value pairs in its `use` and `def` set. The execution path starting from `s'` has the potential to produce the missing rows. Sequence-based difference of the execution path starting from `s'` and the execution path starting from `s` is presented to the user.

To explain the case in loop, we present the following example in Figure 11 along with the input data and trace. The row corresponding to key-value pair `<matnr,3>` is missing. The two slices corresponding to the generated rows with respect to the key `wa_itr-matnr` at write instances 11^8, 11^{16} are $\{1^1, 2^2\}$ and $\{1^1, 2^{10}\}$, respectively. Both the statement occurrences 2^2 and 2^{10} at line 2 contain the table `it_itr` in the `use` set which has the missing key-value pair, and do not contain the missing key-value pair in their `def` sets. Their peer statement occurrence 2^{18} has the missing key-value pair in both the `use` and `def` sets. 13^{22} is the difference between execution traces starting from 2^{18} and either of 2^2 and 2^{10}. This statement occurrence is presented to the user as fault suspect, which helps user to localize the missing write statement.

Incorrect input data. Along with the slice, we also highlight the variables that are defined in each statement occurrence in the slice and belong to the dependency set. This means for a select statement occurring in an incorrect slice, we identify the parts of the data selected which flow to the incorrect output. This information corresponding to the incorrect slices helps user to identify the incorrect input data.

Unwanted rows. In this case some (but not all) unwanted rows are found in the output. The fault localization problem is posed as an application of differencing where incorrect slices are computed based on key fields in the unwanted rows and correct slices are computed based on key fields in the rest of the rows. The key-based slicing, followed by sequence-based differencing, and, if required, semantic differencing is carried out to localize the fault.

Incorrect fields in all rows. In this case incorrect values in one or more fields for all rows in the output is reported as a bug. Incorrect slices are computed for each of the rows, and instead of computing sequence-based difference of them, the common statements in all the incorrect slices are computed. Furthermore, the mutation technique is applied to find a mutation of a statement (in the common statements set), such that the mutated behavior is different from all the existing behaviors of the statement in all incorrect slices. For each statement in the common set, corner-case differencing is applied to find a corner-case behavior of a statement which is present in all its occurrences in incorrect slices.

4. EMPIRICAL EVALUATION

We implemented our analysis algorithms for the ABAP language as a part of an analysis platform towards a joint program with IBM Global Business Services. We evaluate the effectiveness of our core contributions - key-based slicing, semantic differencing, and the analysis for missing rows in output. In our evaluation, the key-value pairs are obtained from the bug description. To perform semantic differencing, we implemented the semantics of several database statements of ABAP in Java, and hence could execute each database statement by mutating its original form.

4.1 Experimental Setup

We used a suite of 13 ABAP samples along with the bugs which were given to us by ABAP practitioners. 3 of these were client programs (SUTAX, ZQFPR, ZFR052). The other 10 bugs were replicas of bugs reported in client situations. As we did not have permission to use the client code, the bug was replicated on copies of similar SAP standard programs. Additionally, we also got 20 toy programs to test our algorithms. The fault relevant source snippets for all the subjects have been provided in the report [24]. To indicate the complexity of the subjects, lines of code and the size of execution (LOC/EXE) are provided in Figure 12. In the same table, we also report on the time it took to run the analysis (does not include trace collection time) for complete fault localization. The time includes sequence-based differencing of key-based slices followed by semantic differencing if required, or analysis for missing output. All the experiments are performed on a 2.53GHz PC with 4GB RAM, running Debian Linux 2.6. To conduct the experiments, we followed the method as given below -

• Execution trace was collected via an automated script, written using the SAP GUI Client scripting facility. The script simulates a step-thru debugging execution of an ABAP program, collecting use and def variable values.
• Fault observation was specified as a pair of precise slicing criteria and associated category (e.g. incorrect, missing).
• For problem categories incorrect and unwanted, we first identify the correct and incorrect slice criteria from user input. Then, we perform the sequence-based differencing on both field-row sensitive slices and key-based slices. If the sequence-based differencing does not yield any difference, we perform both the semantic differencing algorithms on key-based slices. For problems in missing category, we perform the missing algorithm described in subsection 3.3.
• The results were presented, as a navigable dynamic slice, mapped to the source, for the IBM GBS team to verify. The suspected faulty statement as identified by our algorithms was highlighted.

4.2 Results

In 12 cases out of 13, our analysis was able to localize to a single statement and it was manually verified that the line was the source of the fault. For the subject named SUTAX, we were not able to point out a single statement. Verification revealed that the program input was faulty and had to be fixed. The smallest key-based slice we reported was of size 12 (for ORDER program), and the input statement in the slice was related to the fault.

4.2.1 Key-based Slicing

In 5 cases sequence-based differencing yielded some difference. In 3 cases (RO13, IMAT, and IINV) the difference between field-row sensitive slices identified the fault. In 2 cases (MMAT and ORDER), the key-based slice was smaller than the field-row sensitive slice. The slice sizes are mentioned in the field-row, and key columns of Figure 12. A look at the relevant code snippets revealed that field-row sensitive slices had over-approximately included the `DELETE` statements. In case of ORDER this difference was vital in identifying the fault, as `DELETE` statement was key to the bug. In case of MMAT (* marked in Figure 12), field-row sensitive slice found a difference in the correct and incorrect slices, but that was not the exact faulty line. The sequence-based differencing of key-based slices showed the faulty line.

4.2.2 Semantic Differencing

In 5 cases sequence-based differencing (either just on field-row sensitive slices or key-based slices) was sufficient to identify the

Subject	LOC	EXE	Data Slice Size		Sequence Diff.		Semantic Diff.		Missing	Time
			Field-row	Key	Field-row	Key	Corner	Mutability [Space]		in Secs
RO13	1202	2948	8	8	✓	✓	-	-		23
IMAT	2661	3864	4	4	✓	✓	-	-		20
IINV	3154	5299	8	8	✓	✓	-	-		135
MMAT	1019	7251	5	4	✓	✓*	-	-		27
ORDER	1975	7386	13	12	×	✓	-	-		28
RLS	2013	569	4	4	×	×	✓	✓ [1]		1
ZROTC	1066	2397	4	4	×	×	✓	×		46
ZBMR	827	257	5	5	×	×	✓	✓ [1]		4
ZQFPR	1136	275	8	8	-	-	-	-	✓	1
ZFR052	944	520	4	4	-	-	-	-	✓	1
BABL	2795	367	6	6	-	-	-	-	✓	1
RV54	3492	1088	5	5	-	-	-	-	✓	1
SUTAX	1662	4028	12	12	-	-	-	-		2

Figure 12: Fault Localization Result

RLS

```
1  a: f1 f2 f3 f4 f5
2  b: f3 f4 f5 f6 f7
3  c: f1 f2  f3 f4 f5 f6 f7
4  LOOP into a
5    LOOP into b
6      MOVE—CORRESPONDING a to c
7      MOVE—CORRESPONDING b to c
8      MOVE a.f3 to c.f3
9      WRITE c
10   ENDLOOP
11 ENDLOOP.
```

(a)

ZROTC

```
1  SELECT from tab into table itab
2  LOOP at ktab
3    READ itab INTO w_itab
4          WITH KEY a = ktab.a
5    w_jtab.a = w_itab.a %overflow
6    APPEND w_jtab to jtab
7  ENDLOOP
8  ...
9  write_alv jtab.
```

(b)

ZBMR

```
1   SORT it_ekpo BY ebeln ebelp
2                 matnr werks
3   ...
4   LOOP AT it_ekbe INTO wa_ekbe
5     READ TABLE it_ekpo INTO wa_ekpo
6       WITH KEY
7             ebeln = wa_ekbe.ebeln
8             matnr = wa_ekbe.matnr
9             werks = wa_ekbe.werks
10                    BINARY SEARCH
11      ...
```

(c)

Figure 13: Code Snippets. (a) RLS, (b) ZROTC, (c) ZBMR

fault. In 3 cases, where the sequence-based differencing failed, semantic differencing was able to identify the fault. We now discuss in detail these 3 interesting cases -

RLS. In the example code snippet shown in Figure 13(a), the `MOVE-CORRESPONDING` X to Y statement moves the values from structure X to Y for the common fields. The Lines 1-3 show the fields for the structure. The correct assignments that needed to be done here are `c.f1=a.f1`, `c.f2=a.f2`, `c.f3=a.f3`, `c.f4=a.f4`, `c.f5=b.f5`, `c.f6=b.f6`, `c.f7=b.f7`. The assignment `a.f4` to `c.f4` was missing in the `MOVE` statement in Line 8. The error is only noticed when `a.f4` is different from `b.f4`. In an iteration which produced the correct output, both `a.f4` and `b.f4` were same. In an incorrect iteration `b.f4` had a non-zero value, whereas `a.f4` had a zero value. Both correct and incorrect slices had same sequence of statements having second `MOVE-CORRESPONDING` and not the first. So, sequence-based differencing failed to discover any difference. The corner-case differencing tried two corner (overflow and overwriting, cf. Figure 8) cases for the `MOVE-CORRESPONDING` statement. And the overwriting corner-case evaluation showed the following difference - `MOVE-CORRESPONDING` at Line 7 overwrites `c.f4` with its existing value in correct slice and with different value in the incorrect slice. This is also located using a mutation where the `MOVE-CORRESPONDING` statement in the slice is mutated to a sequence of `MOVE` statements and deleting the `MOVE-CORRESPONDING` to `c.f4=b.f4`. The assignment related to field `f4` is chosen for deletion as it is the common field between structure variables `a` and `b` that assigns values to structure variable `c`.

ZROTC. In Figure 13(b) Line 5 the overflow occurs in the assignment statement as size of $w\_jtab.a$ is smaller than that of $w\_itab.a$. The overflow is visible in incorrect slices as non-zero digits were truncated due to overflow. However, in correct slices only zeros were truncated, which did not produce any ill effect to the computed result. This statement was there in both correct and incorrect slices, and therefore sequence-based differencing was not able to catch this semantic difference. The corner-case differencing was able to catch this behavior as this is one of the corner-case that is determined in assignment statement (Figure 8). Note that, the fix to this bug is not a change the assignment statement, but requires a change in type in the declaration of the variables. As mutation only considers mutating a statement, this bug cannot be found using mutation.

ZBMR. The example shown in Figure 13(c) has the same flavor as our running example in Figure 1 with data in Figure 5. In this case the bug was the under-specification of the key constraint in the read statement. This resulted in the wrong row selection by the read statement in the incorrect slice. As explained in Section 3, both corner-case differencing and mutability differencing were able to find the error. Note that, in this case there was only one more common field (`ebelp`) between table `it_ekbe` and `it_ekpo` which was not present in the selection condition in the read statement. Thus mutability differencing considered only one mutation of the current statement.

The mutation space observed in our experiments was small as we use heuristics to restrict the mutation space.

4.2.3 Missing Rows in Output.

We explain a case for missing output with the code snippet of program `RV54` shown in Figure 14. The bug reported was that for some keys, corresponding rows were missing from the output. In the program, there was a `DELETE` statement, which was deleting the rows where the `f_new` field is null. Some computations were performed on the rows to produce output. We first obtained a key-based data slice which was of length 5 for an existing row in the

```
PERFORM batch_heading_babl
...
DELETE gt_output WHERE f_new IS INITIAL

PERFORM aendbelege_lesen
..
output gt_output
```

Figure 14: Code Snippet for Missing Row in RV54

output. Then by performing a backward traversal in the slice we found the `DELETE` statement that was deleting rows with the same key value as that of the reported missing data. As this statement was not in the loop, this was highlighted in the slice as the reason for the missing rows in the output. It was verified by the ABAP practitioners that, it was indeed the faulty line. The deletion should not have been to the `gt_output` table.

5. RELATED WORK

Static Program Slicing of ABAP Programs. Dor et al. [11] used static slicing to analyze how a given ABAP code uses data stored in different database tables. Their system PanayaAI, identifies all select statements in code and computes a forward static slice from these statements, to infer how the data returned from the select is being consumed in the program. They presented three algorithms; improving from a flow-field-context insensitive algorithm to field-sensitive, and finally to a flow-field sensitive algorithm. In their more recent work Litvak et al. [19] recognized the importance of field sensitive analysis in the ERP systems domain, and present an algorithm for efficient and precise computation of program dependences in the presence of large structure variables. In this paper we presented a backward dynamic slicing algorithm, which is field and row sensitive. We also observed that row-sensitivity plays an important role in reducing statement occurrences in dynamic slice.

Dynamic Program Slicing. There is significant work in the area of dynamic program slicing. Korel and Laski [18] had introduced the notion of a dynamic slice, way back in 1988. Agrawal and Horgan [1] significantly optimized the notion by dropping the executability constraints. Venkatesh [26] worked on separating the semantics based definition of a program slice from the semantic justification of an algorithm. Kamkar et al. [17] worked on interprocedural dynamic slices. Zhang and Gupta [29, 28] improvised the algorithms for dynamic slice computation in the presence of arrays, structures and pointers for complex real world programs.

Hainaut et al. [8] and Cleve [7] looked at applying dynamic analysis on data intensive systems, which contain embedded and dynamic SQL (such as in JDBC). The aim was to resolve the input queries being passed to a database as precisely as possible. The authors showed how just a static scan of the code for SQL statements does not suffice. The collected trace was used for program comprehension and to infer implicit referential constraints between database tables. We are similar to these papers as we also use dynamic analysis. However, our presented dynamic slicing techniques go a few steps further; in presence of integrated data intensive operations, we present row and field sensitive slicing, and extend it with key-based slicing.

Differencing based Fault Localization. Fault localization by differencing two program runs has been widely applied. The notion of spectrum (abstract trace) was introduced by Reps et al. [23] for acyclic, and intraprocedural path spectra. Harrold et al. [13] generalized the notion of spectrum and proposed spectra based on several program features - branch, complete-path, data-dependence, output, and execution trace. Tarantula [16] provides a visualization of various passing and failing test runs of a software system. Here the authors explored how visualizing the hit or miss count of various code statements in passing and failing runs can help users localize faults faster. Zeller [27] applied systematic delta changes to program input to generate guided passing and failing execution, that could be differenced to detect cause-effect chains more precisely. Renieris and Reiss [22] introduced distance spectrum. In a distance spectrum, a distance measure between the passing and failing spectra gives a measure of dissimilarity.

In the context of software verification, a number of techniques [25, 15] have been proposed to provide users with minimal information required to explain counter-examples resulting from model checking. Some techniques [4, 5] localize the errors in programs by identifying the diverging point between a counter-example and a *positive* example; a positive example is a sequence of statements in programs that does not lead to a violation of the property of interest. A similar approach is presented in [12] where errors are localized to program statements absent in all positive examples and present in all counter-examples leading to the same error condition.

In our approach, we difference between dynamic slices i.e. identify statements that have some effect in producing incorrect output but has different effect in producing correct output. However, we additionally, also identify if each executed statement showed similar behavior in each run.

Another novelty of our approach is that we split a single program execution into multiple slices that are further classified as correct or incorrect. We are able to do so because of the nature of the programs we are analyzing. Each program in a single run, produces some correct and some incorrect values. Mani et al. [21] applied a similar technique to retrieve passing and failing traces from a single execution to compute repair recommendations for model transforms. They used tainting to resolve how input data moved within a program to generate the output. However, their approach would not be applicable in our case where we need to handle data flows between code and external database. We used key based slicing to split a single execution run into multiple traces.

Mutation Analysis. In the area of testing, mutation technique is used to generate faulty programs from a correct program [10, 3, 20] to study the path divergence of faulty programs from the correct programs. Debroy et al. [9], use mutation based approach to suggest repairs to localized faults. The two classes of mutant operators used there are replacement of expression, replacement of assignment operator by another operator from the same class, and decision negation. Our mutation operators are specialized for database statements. Chandra et al. [6] determine alternate values of an expression to satisfy the goal of correcting the failing tests without breaking the correct traces. In comparison, we consider the syntactic mutations that change the outcome of a statement in the incorrect slices without changing its outcome in the correct slices. The space of syntactic mutations is managed due to the domain-specific nature of ABAP programs.

6. CONCLUSION

Fault localization using slicing and differencing have been identified as important techniques for performing fault localization in procedural programming languages. In this paper, we extend these techniques to data-centric programming languages which use embedded database specific statements to perform operations on in-memory and persistent data.

We present a new key-based dynamic slicing algorithm and two differencing techniques that use the underlying program semantics to localize faults in the data-centric programs. We applied our techniques on 13 real industrial programs and identified the underlying faults accurately in 12 of them.

We notice that, in data-centric programming paradigm the processing of data is separated out across different systems and languages. For example, many applications use the Java - JDBC - Stored Procedure framework to create a data-centric application. In future, we aim to check the applicability of our techniques in such paradigms.

Acknowledgment. We thank Anup Ghosh, Sugata Ghosal, Sourin Ghosh, Dhiman Ghosh, Anjan Nandy, Asidhara Lahiri, and Amitava Parui for many fruitful discussions and help in implementations of the project. We also thank Julian Dolby for sharing a version of the ABAP grammar. We also like to thank anonymous referees for their helpful comments on the earlier version of the paper.

7. REFERENCES

[1] H. Agrawal and J. Horgan. Dynamic program slicing. *ACM SIGPLAN Notices*, 25(6):246–256, 1990.

[2] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong. Fault localization using execution slices and dataflow tests. In *International Symposium on Software Reliability Engineering*, 1995.

[3] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Trans. Softw. Eng.*, 32:608–624, August 2006.

[4] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *Proceedings of POPL*, pages 97–105, New York, NY, USA, 2003. ACM.

[5] S. Basu, D. Saha, and S. A. Smolka. Localizing programs errors for cimple debugging. In *International Conference on Formal Techniques for Networked and Distributed Systems (FORTE)*, volume 3235, pages 79–96, Madrid, Spain, September 2004. Springer-Verlag.

[6] S. Chandra, E. Torlak, S. Barman, and R. Bodik. Angelic debugging. In *Proceedings of ICSE*, pages 121–130, New York, NY, USA, 2011. ACM.

[7] A. Cleve. Program analysis and transformation for data-intensive system evolution. In *Proceedings of ICSM*, pages 1–6, Washington, DC, USA, 2010. IEEE Computer Society.

[8] A. Cleve and J.-L. Hainaut. Dynamic analysis of sql statements for data-intensive applications reverse engineering. In *Proceedings of the 2008 15th Working Conference on Reverse Engineering*, pages 192–196, Washington, DC, USA, 2008. IEEE Computer Society.

[9] V. Debroy and W. E. Wong. Using Mutation to Automatically Suggest Fixes for Faulty Programs. In *Proceedings of ICST*, pages 65–74. IEEE, Apr. 2010.

[10] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Trans. Softw. Eng.*, 32:733–752, September 2006.

[11] N. Dor, T. Lev-Ami, S. Litvak, M. Sagiv, and D. Weiss. Customization change impact analysis for ERP professionals via program slicing. In *Proceedings of ISSTA*, pages 97–108. ACM, 2008.

[12] A. Groce and W. Visser. What went wrong: explaining counterexamples. In *Proceedings of SPIN*, pages 121–136, Berlin, Heidelberg, 2003. Springer-Verlag.

[13] M. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability*, 10(3):171–194, 2000.

[14] M. W. James R. Lyle. Automatic program bug location by program slicing. In *2nd International Conference on Computers And Applications*, pages 877–882, 1987.

[15] H. Jin, K. Ravi, and F. Somenzi. Fate and free will in error traces. In *Proceedings of the TACAS*, pages 445–459, London, UK, UK, 2002. Springer-Verlag.

[16] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of ASE*, pages 273–282, New York, NY, USA, 2005. ACM.

[17] M. Kamkar, N. Shahmehri, and P. Fritzson. Interprocedural dynamic slicing. In *Programming Language Implementation and Logic Programming*, pages 370–384. Springer, 1992.

[18] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[19] S. Litvak, N. Dor, R. Bodik, N. Rinetzky, and M. Sagiv. Field-sensitive program dependence analysis. In *Proceedings of FSE*, pages 287–296, New York, NY, USA, 2010. ACM.

[20] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Trans. Softw. Eng.*, 32:831–848, October 2006.

[21] S. Mani, V. Sinha, P. Dhoolia, and S. Sinha. Automated support for repairing input-model faults. In *Proceedings of ASE*, pages 195–204. ACM, 2010.

[22] M. Renieres and S. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of ASE*, pages 30–39, 2003.

[23] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. *Software Engineering—ESEC/FSE'97*, pages 432–449, 1997.

[24] D. Saha, M. G. Nanda, P. Dhoolia, V. K. Nandivada, V. Sinha, and S. Chandra. Fault localization in ABAP Programs. Technical Report RI11004, IBM, http://domino.research.ibm.com/library/cyberdig.nsf/index.html, March 2011.

[25] N. Sharygina and D. Peled. A combined testing and verification approach for software reliability. In *Proceedings of FME*, 2001.

[26] G. A. Venkatesh. The semantic approach to program slicing. In *Proceedings of PLDI*, pages 107–119, New York, NY, USA, 1991. ACM.

[27] A. Zeller. Isolating cause-effect chains from computer programs. *ACM SIGSOFT Software Engineering Notes*, 27(6):1–10, 2002.

[28] X. Zhang and R. Gupta. Cost effective dynamic program slicing. In *Proceedings of PLDI*, pages 94–106, New York, NY, USA, 2004. ACM.

[29] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *Proceedings of ICSE*, pages 319–329, Washington, DC, USA, 2003. IEEE Computer Society.