RI11004, 11 February 2011 Computer Science

IBM Research Report

Fault localization in ABAP Programs

Diptikalyan Saha
IBM Research - India

Mangala Gowri Nanda
IBM Research - India

Pankaj Dhoolia
IBM Research - India

V. Krishna Nandivada
IBM Research - India

Vibha Sinha
IBM Research - India

Satish Chandra
IBM T. J. Watson Research Center

IBM Research Division
Almaden - Austin - Beijing - Delhi - Haifa - T.J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted is accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T.J. Watson Research Center, Publications, P.O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com).. Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

Abstract

In this paper we present an automated technique for localizing faults in data centric programs. Data-centric programs typically interact with databases to get collections of content, process each entry in the collection and output another collection or write entries back to database. Many of the production faults in data centric programs are manifested because of presence of certain data values or data patterns in the database that were not checked for or handled correctly during development of the initial code. Our technique collects the execution trace of the faulty program, uses a novel precise slicing algorithm to break the trace into slices mapping to each entry in the output collection and finally does a semantic difference between the slices corresponding to correct output entries versus incorrect ones to suggest potentially faulty statements. We implemented our approach for ABAP programs. ABAP is the language used to code in SAP systems and interacts heavily with databases. It also contains embedded SQL like commands to work on collections of data. We applied our technique to a suite of 13 ABAP programs with faults from the field and it was able to identify the precise cause in 12 cases.

1 Introduction

Bug resolution is an important activity in any maintenance oriented 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 as fast as possible. Second, the bug has arisen despite the fact that the code has been well-tested and probably been running in the field for some time. That means it is probably a corner case in otherwise correct code and hence is very often a one-line fix. Since speed is of the essence, it is important to have good tooling support that can help the programmer debug the program as fast as possible. This is especially true when the person who is debugging the code is not the programmer who has written the code—and hence is not familiar with the code.

The techniques presented in this paper, were developed to aid in faster resolution of code bugs reported for ABAP programs. ABAP is a propriety language used by SAP and is heavily data centric. Data-centric programs process large collections of data that typically are coming 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

```
1   SELECT CustId ItemId Price Year from OrderTab INTO itab
2   SELECT CustId Discount Year from DiscountTab INTO stab
3
4   SORT itab CustId ItemId
5   DEL from itab where Year <= CurrentYear-2.
6   LOOP AT itab INTO wa
7     AT NEW CustId
8       amount=0.
9     ENDAT
10    amount = amount + wa.Price
11    READ stab INTO fa WHERE CustId = wa.CustId
12    IF subrc = 0
13      amount = amount - fa.Discount
14    ENDIF
15    AT END CustId
16      WRITE CustId amount
17    ENDAT
18  ENDLOOP
```

Figure 1: Sample ABAP program

command	description
`SELECT`	project selected columns from a persistent database table to internal to 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 the 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 a row from table `stab` based on the key value. If more than one row matches, the last row is returned
`WRITE`	prints the specified data

Figure 2: Basic ABAP syntax

(a)

OrderTab

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

DiscountTab

CustId	Discount	Year
1	2.0	2010
2	3.0	2011

Output

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

(b)

1 – 2 – 4 – 5 – 7 – 8 – (10 – 11 – 12 – 13) – 15 – 16 Line 16, <1,16.0>

1 – 2 – 4 – 5 – 7 – 8 – 10 – 11 – 12 – 13 – 15 – 16 Line 16,<2,7.0>

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

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 $\surd$, and the output rows that are considered incorrect are followed by a $\times$ mark and the expected correct output. The `OrderTab` table contains the order details such as customer who placed the order `CustId`, item ordered `ItemId`, it's price and year when order was placed `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`. It then loops over the contents of `itab` (line 6), sums up the price (`Price` at line 10), subtracts any relevant discount (`Discount` at line 13) and prints out the total (line 16). One output entry is generated for each unique `CustId` present in the input `OrderTab`.

There are multiple challenges involved in developing fault localization techniques for these type 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 different SQL like commands in an ABAP program. The other one 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

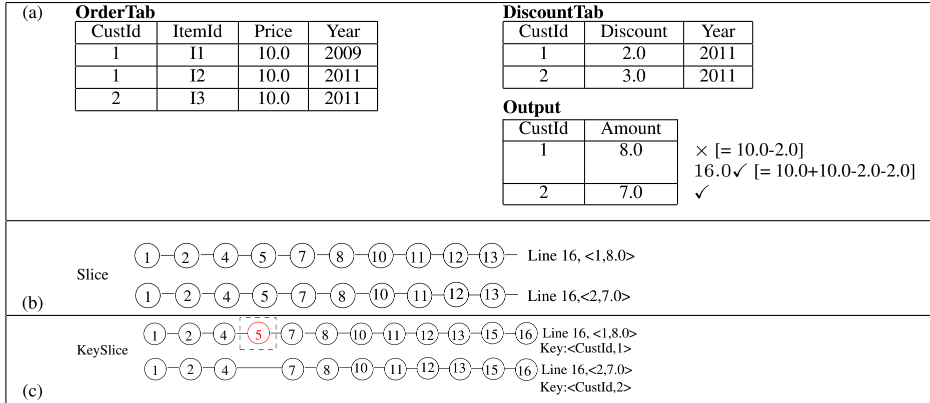

Figure 4: (a) Input and output for the ABAP program illustrated in Figure 1. (b) Dynamic slices for each output entry. (c) Key based slices for each output row.

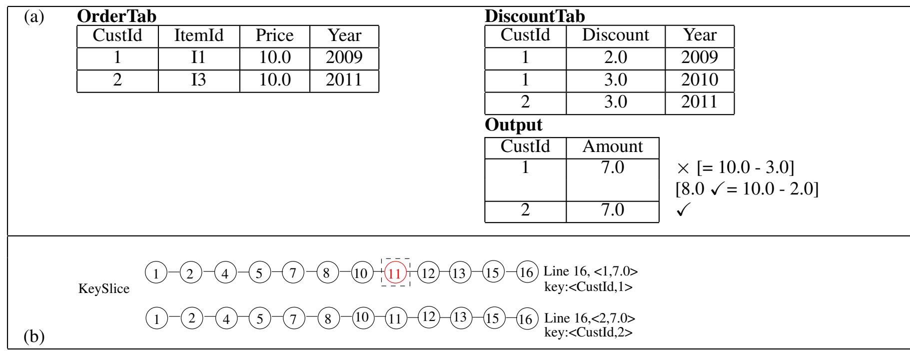

Figure 5: (a) Input and output for ABAP program illustrated in Figure 1. (b) Key based slice for each output row.

and yet may throw an exception or generate incorrect output for some other data. That said, we nevertheless, assume that the data in the input source on which the failing run executed is itself consistent. That is, it does not violate its own integrity constraints. This assumption is reasonable, because the same data-source typically feeds into several other applications that do work properly. Hence the assumption is that the bug is always in the code—not in the data. As we'll see further in the paper, for the example code in Figure 1, depending on the combinations of data available in the `OrderTab` and `DiscountTab` tables, three different code bugs are revealed based on the output. However, in each of the three cases, the amount reported for `CustId = 1` is incorrect.

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 w.r.t. y but not in the slice w.r.t x ([13]). 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 the difference between the slices of these two program executions ([21]). 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 bug 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. 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 shredded 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 example discussed above, the key value was `CustId`. 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) then, we are able to associate a "faulty" or a "correct" tag with each slice. 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 do differencing 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, lines 10 through 14 were moved inside the `AT END` block (after line 15). The c

Key Based Slicing It may appear that we can always generate faulty and good slices and potentially, apply a differencing technique to this setting. However, this is not always 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 same sample program in Figure 4(a). In this case the end-user was expecting to see the amount value as 16.0 for `CustId = 1` [considering that we have the example 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 (as we assume end-user expectation and data in the input source is correct). The slices for both the output rows are 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, you do write out an input field that acts as the identifier (or key) for the data and is unchanged from the input to output. For our example, this field is the `CustId`. 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 `CustId = 1`. The values in the table `itab` before (`Pre`) and after statement #5 (`Post`) are given below

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

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 faulty slice, and not the correct slice.

Semantic Differencing However, in some cases even key based slicing is not enough. Consider the data example 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 last one. For `CustId = 1`, line 11 would need to select from two records, while for `CustId = 2`, there is only one matching record. Hence, the behaviors of the `READ` statement for these two keys are different. Our 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 correct fix is to join with the `Year` attribute in the read statement so that for item bought in a specific year we get the correct discount for that year.

Novelty and Contributions Key-based slicing and semantic differencing are novel generalizations of previous work on fault localization by 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 of tremendous importance 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 given down to a single statement (for semantic differencing, we were looking for only a single statement difference.)

Our experiments with the tool show that generalized differencing was able to accurately localize fault in 12 out of 13 ABAP programs provided to us by our colleagues in IBM Global Business Services. For These ABAP programs were either old versions of some programs where there was a known defect that has since been fixes, 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 4 out of the 13 programs.

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

2 Dynamic Slicing

In this section we present an algorithm that performs precise dynamic slicing on execution trace 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 $\langle Line^{seq}, V\rangle$. All the slices that produce incorrect output are marked as called bad (faulty/incorrect) slice, whereas the slices that produce correct output data are called good (or correct) slice. The slices, obtained this way, are analyzed with a differencing algorithm to reason about the possible fault present in bad slice. Efficacy of our differencing algorithm (cf. Section 3) depends on the precision of the slicing algorithm.

Traditional slicing algorithms are oblivious to the rows and fields of the underlying table data, resulting in overly conservative slices, and is not suitable for an effective differencing in data-centric programs. Based on the existing techniques of handling non-scalar data ([31, 30, 18]), a row and field sensitive algorithm is obtained first. The algorithm is built on top of the 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 variables whose table index is changed by delete statement. The control effect of the delete statement will be all the statements whose number of executions in the trace is dependent on the number of rows in the table. In general, for each compound statement, its final effect is represented by a set of assignment statements, and a set 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
13^{25}	amount= amount- fa.discount	amount, fa.discount
11^{23}	read stab into fa where custId=wa.CustId	amount, stab[1].discount, wa.CustId
10^{22}	amount=amount+wa.price	amount, wa.price, stab[1].discount, wa.CustId
8^{20}	amount=0.	wa.price, stab[2].discount, wa.CustId
6^{18}	loop at itab into wa.	itab[1].price, stab[1].discount, itab[1].CustId
5^4	DEL from itab where ..	itab[2].price, stab[1].discount, itab[2].CustId
2^2	Select .. from DiscountTab into stab	itab[2].price, DiscountTab[1].discount, itab[2].CustId
1^1	Select .. from OrderTab into itab	OrderTab[2].price, DiscountTab[1].discount, OrderTab[2].CustId

(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) we show the update of the data dependency information after including

each of the statement 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, 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 has `Year` value ≤ 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 deleted rows affect the computation performed to compute the sum at 16^{15}. The affect in computation is captured both by control and data dependency. The data dependency is captured using the intersection of def set, and control dependency includes the statement as number of iteration of loop for `CustId`=1 is dependent on the delete statement.

2.1 Key-based slicing

As discussed in the previous section, the row and field sensitive slicing algorithm discussed before can result in imprecise slices. An important question to answer is, when does a statement occurrence is part of the slice? In our application, as dynamic slices are representative of the computation that affects the rows in the output, then a statement is not part of the slice if absence of it does not have any effect in the *computation* of the variable values in the output row associated to the slice. It is easy to see that, in the above example the delete statement does not have any effect in computing the `amount` value in the second row.

Here, we note one important assumption. If a statement occurrence only affects the position of a row in the output, and not the variable values specified in slicing criteria in the row, the statement is not part of the slice. For example, if a variation of the delete statement deleted all the rows related to `CustId`=1, which would have shifted the second output row to first, even then the delete statement is considered to be not part of the slice corresponding to the second row. Many a times the respective order of rows in the output is not important; as we have found in our experience it is rare to find bugs related to the order of rows in the output, instead the bug is found in the content of the row.

To determine whether a statement occurrence is affecting the variables values in the slicing criteria, we need to check two conditions: ($C1$) if the statement occurrence is performing any operation which defines a variable in the dependency set, ($C2$) if the statement can effect the dependency set itself in terms of addition or deletion of elements. If any one of the condition $C1$ and $C2$ is true the statement is added to the slice. For example, the delete statement in our running example do not satisfy the condition $C1$ for the slices corresponding to both the rows of the output, but satisfies $C2$ for the first row, and not for the second row. Checking condition $C1$ is relatively easier than checking $C2$ for statements that operates on tables.

In this paper, we present a *sufficient* criteria to check the condition $C2$. The main aim of the criteria is to try to remove a statement from the slice which otherwise be included by the basic row-sensitive algorithm. The main idea of our algorithm is to associate a set of key-value pairs with the slicing criteria, such that, the selection of the elements to compute the variable values specified in slicing criteria can be identified by the key-value pairs. In our example, it is evident that the slice with respect to the criteria $\langle 16^{28}, \{\texttt{amount}\}\rangle$ and $\langle 16^{15}, \{\texttt{amount}\}\rangle$ have association with key-value pair $\langle \texttt{CustId}, 2\rangle$, $\langle \texttt{CustId}, 1\rangle$ 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. 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. In general, a statement is included in the slice if any change performed by the statement (such as added, deleted, or updated rows) matches the key-value pairs.

Along with the key-value pair conditions, we associate *sequence condition* which states that elements that are used to compute the variables in slicing criteria are in adjacent rows in an internal table. This is particularly useful to express `group-by` constraint in database operation. Consider the example pre-state of the delete statement 5^4.

```
        itab(Pre)                  itab(Post)
CustId ItemId Price Year     CustId   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
$\langle 16^{28}, \{\texttt{amount}\}\rangle$ given our assumption of unimportance of position of output row. In this case, the delete statement performs a change that is satisfied by key-value constraint as one of the deleted row has `CustId`=2, but the change does not satisfy the key-value condition and the sequence condition together, as the deleted row is not adjacent to the rows in the dependency set and therefore will not be in the sequence for further selection.

There are multiple ways to identify key-fields for association.

- The key-fields can be specified by the user. This is not an unrealistic assumption in the context of fault localization, as we have observed many of the bug reports contained this information.
- 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 operate on the rows (select, delete, modify, and so on).

However, we can only use the key-value and sequence based conditions to filter out a statement from slice, only under the condition that both key condition and sequence condition hold in the existing element in the dependency set. We call them the key-value assumption and sequencing assumption. Thus checking of these assumptions is required and if the assumption is violated then appropriate approximation is chosen based on the chain provided in Figure 7. As we go up in the chain the slice gets more precise with increase overhead of computation.

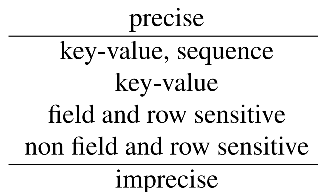

Figure 7: Slicing Algorithms Chain

Our key-based slicing algorithm is presented in Figure 8. Statements 1-6, 8-11, 16-17, 20-22 represent a basic field and row sensitive dynamic data slicing algorithm. The function `get_def_use(s)` returns a set of def-use pairs representing def-use relationship between each pair. We assume that function
`field_row_sensitive_inclusion_check` performs the necessary checks like nonempty

```
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 φ = 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,φ))
12     && (!key_assumption_valid ||
13       (key_assumption_valid && check_kv_constraint(kvp,s))
14        && (!sequence_valid || (sequence_valid &&
15          check_sequence_constraint(kvp,s,φ)))
16       slice.add(s);
17       φ=update_dep_set(φ,duSet);
18       key_assumption_valid=check_key_assumption(φ,kvp);
19       sequence_valid=check_sequence_assumption(φ);
20   endif
21 endwhile
22 return slice;
23
24 check_kv_constraint kvp, s
25   if s is not operating on table
26      return true
27
28   if any change by s satisfies kvp
29      return true;
30   else
31      return false;
32   endif
33
34 check_sequence_constraint kvp, s, φ
35   if s is not operating on table
36      return true
37
38
39   c = changes by s that satisfies kvp
40   if check_sequence_assumption (c ∪ φ)
41     return true;
42   else
43     return false;
44
45
46 check_key_assumption φ, kvp
47   for each structure variable v.f
48     if f ∈ kvp.keySet
49        if value of v.f != kvp.getValue(f)
50             return false
51   return true
52
53 check_sequence_assumption φ
54    for all itab
55       for any f
56         I = set of all i s.t. itab[i].f ∈ φ
57         if I is not in sequence
58            return false
59    return false
```

Figure 8: Key based Slicing

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 include in the slice, the dependency set is updated by removing the def and including use of each def-use pair in duSet [Line 17]. Above the check done by field-row sensitive check, we add the key-value and sequence checking 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 statements that work

on tables. The function `check_key_assumption` checks that the all non-scalar elements in the dependency set satisfied the key-value pairs, and function `check_sequence_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 execution trace.

Note that, it is possible to give a necessary and sufficient condition to check the condition $C2$. However, 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.

3 Slice Differencing

In this section we present the fault localization algorithms. Our fault localization method is particularly effective when we observe that in an execution some parts of the output are produced correct and some parts parts of the output are incorrect. We take an existing approach to find fault - to find the difference in behavior between *slices* that produce correct and incorrect output ([13]), and localize the fault at difference points. The novelty of our technique lies in the algorithm to compute the differences between slices.

In context of difference based fault localization, existing literature have used different type of program information to apply differencing - counter-example and positive example (that do not lead to property violation) ([10]), execution trace of passing and failing test runs ([21]). In this paper, we consider dynamic slices (for correct and incorrect output) generated out of a single execution trace as sequences to compare and find differences for localizing fault. In the context where single execution is generating a output with multiple entries, such slices are representative of the computations that are producing specific entry (row) of the output and as such difference between such computation will lead to the error. We omit the pure control statements (If/Case/Loop/While) from the slice, but include the compound statement (statements that can have both data and control effect) that can affect the control behavior of the execution. For example, Loop statement in ABAP can take both the form, one in which it loop till the loop condition gets false. In another form, it loops for all rows in a table. We consider the second form of statement in our slice, but do not consider the first type of loop statement. We use a control differencing algorithm to find out any difference in control behaviors of the slices. Based on whether the output produced is correct or incorrect, we classify slices into correct/good slices and incorrect/bad slices.

We apply two main techniques in differencing. The first technique is relatively simple, where any control difference between two slices is determined. Such difference analysis typically determines the difference in statement occurrences in correct and incorrect slices. This difference signifies that certain statements have some effect in computing the incorrect output, but have no contribution in computing the correct output. These statements are presented as potential fault points. However, it is possible that control based differencing may not produce any difference as same sequences of statements can present in slices. The second technique is particularly novel as it tries to find out the behavioral difference between two slices - finding statements that are present in both correct and incorrect slices but shows difference in their semantic behaviors (cf.

Sub-section 3.1). Note that, these differencing techniques are useful to localize faults when certain rows (but not all) in the generated output have incorrect results.

Control-based Differencing The main aim of control-based differencing is to identify statements that have contribution towards computation of the incorrect result and have no contribution in computation of at-least one correct result. To perform this we perform a two step process: 1) grouping correct and incorrect slices into equivalence classes and 2) perform pair-wise differencing between a representative elements of the correct and incorrect equivalence classes.

In data-centric programs it is very common that slices containing same set of statements, typically differ in number of iterations of the same loops. Thus, while creating equivalence classes in correct threads or in incorrect threads, we combine two slices into the same class if they are exactly same or if they have different number (> 0) of iterations of the same statements in loop. Generating equivalence classes in correct slices and in incorrect slices reduces the number of pair-wise comparisons require to find differences between slices.

While control differencing two elements, one from a correct equivalence class, and another from an incorrect equivalence class, any difference in sequence of statements is noted. However, due to common nature of loop-iteration differences, these differences are given lower priority among all-sets of pair wise differences. The actual algorithm of control-differencing in presented below.

We represent the slice by slice summary based on the following definition.

Definition 1 (slice Summary) *Given a slice $T = \langle s_0 = I_0^{q_0}, \ldots, s_n = I_n^{q_n} \rangle$, its slice summary $T_s(T) = \langle S_0, \ldots, S_m \rangle$ is defined as a sequence of summary nodes (S_i) where each summary node is either a node in the slice i.e. for $s_k = I_k^{q_k}$, $S_k = I_k$ where line I_k is not in a loop, otherwise a loop summary node $L_{k,l}$ which represents a sequence of slice nodes $s_k, \ldots, s_l$ all of them belong to a loop and S_0 is I_0 or $L_{0,j}$ and S_{i-1} is I_{k-1} or $L_{h,k-1}$.*

Each slice can be viewed as an alternative loop part and non-loop part. Each loop part is summarized by a loop-summary node. A loop summary node summarizes the slice nodes in a loop, based on the following definition.

Definition 2 (Loop Summary) *A loop summary node corresponding to sequence in a slice $s_i, \ldots, s_j$ is a 4-tuple $\langle I, \alpha, \beta, \gamma \rangle$ where I is the loop id, the line number of the loop statement. α is the sequence of slice summary nodes corresponding to its first iteration in the loop. Say the loop iteration number for the first iteration be l. γ is the sequence of slice summary node corresponding to the last iteration of the loop when number of loop iterations is more than one. β is $[(T_{s1}, I_1), \ldots, (T_{s_n}, I_n)]$ where each $(T_s i, I_i)$ is a pair of slice summary of a slice corresponding to a loop iteration and the set of loop iterations numbers greater than l of the slices that are represented by $T_s i$.*

Before the difference between various slices is computed, we classify the slices into equivalence class based on the equality relation which states that two slices are equal if their corresponding slices summaries are exactly equal. If the slices are equal then they have the same length and exact sequence of line numbers, but the converse is not true, as iteration number is included in slice summary.

Once the grouping is done, pair-wise differencing between correct and incorrect slices summaries are performed. In this comparison two slice summaries are compared for equality based on

the notion of order-obliviousness. If the two slices summaries are not equal then the difference is noted.

Definition 3 (Equality of slice Summary) *Two slice summaries* $T^1 = \langle S^1_0, \ldots, S^1_n \rangle$ *and* $T2 = \langle S^2_0, \ldots, S^2_m \rangle$ *are said to be order-oblivious equal (denoted as* $T^1 =_o T^2$*) if* $\forall i, S^1_i = S^2_i$ *and* $m = n$*. Two summary nodes* $S1$ *and* $S2$ *to be equal if* $S1 = \langle I \rangle$ *and* $S2 = \langle I \rangle$*, or* $S1 = \langle I_1, \alpha_1, \beta_1, \gamma_1 \rangle$ *and* $S2 = \langle I_2, \alpha_2, \beta_2, \gamma_2 \rangle$*, and* $I1 = I2$*, and prefix*(α_1, α_2) $\vee$ *prefix*(α_2, α_1)*, and if* $\gamma_1 \wedge \gamma_2$ *is not null then prefix*(γ_1, γ_2) $\vee$ *prefix*(γ_2, γ_1)*, and* $\forall (T_s, _) \in \beta_1, \exists (T_s, _) \in \beta_2$ *if* β_2 *is not empty and* $\forall (T_s, _) \in \beta_2, \exists (T_s, _) \in \beta_1$ *if* β_1 *is not empty.*

3.1 Semantic Differencing

A bad slice may not show any important difference with good slices in control-based differencing. This is possible if a statement exhibits different "behaviors" in two slices due to 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 semantic differences of a statement.

Consider the example program presented in Figure 1 and the corresponding test case shown in Figure 5(a). In this example corresponding to the `CustId=1` there are two rows in `stab`. In `read` statement at Line 11 when the selection condition is satisfied with multiple rows, then last matching row is selected for indexed table based on ABAP semantics. Considering `stab` to be an indexed table here, the last row is selected with `Discount=3.0`, which results in output `7.0` instead of the correct output `8.0` corresponding to `CustId=1`. In this example, same slices exist for the two output rows as shown in Figure 5(b). So control differencing would not be able to find any difference in slices.

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

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

Corner Case Differencing The first method of semantic differencing is called *corner-case differencing*. The semantics of each compound statement is classified into a different 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 last (for indexed table) matching row is returned by the read, the matching of just one row is a corner case. A table of corner and non-corner cases for several statements is given in Figure 9. Given a trace, we can tell if a statement executed in a corner-case manner, or in a normal case manner.

Statement	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.
insert/append	The inserted/appended row makes sorted data unsorted
Assignment move move-corresponding transporting clause	overflow overwriting same value
LOOP at itab. ... END-LOOP.	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 $a <= b$	$a = b$
selection condition $a >= b$	$a = b$
$a = b + c$	$a = b \vee a = c$

Figure 9: Corner Case Differences

Intuitively, this technique exploits the fact that most errors (typically shown in a already tested code) occurs due to non-handling of corner cases that are revealed in the bad slices, and not revealed in the good 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 with respect to good and bad 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 good slice only single row satisfies the selection condition, but in case of bad 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 read statement exists is indeed this particular behavior found using behavioral differencing. Note that, 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 good and bad slices is the key observation. Several other checks are presented in Figure 9.

Mutability Differencing Our second method 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 bad slice, expecting that the produced behavior

Statement	Mutation
key constraint C in read/select/delete/ insert/append/modify	addition of key constraint deletion of key constraint
non-key constraint C in read/select/delete/loop insert/append/modify	modify C with post-condition imposed by a good and all bad 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 10: Mutation Operators

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 good 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 `CustId` and `Year` in the input tables `OrderTab, DiscountTab`. Any of them or their combination could be used as the joining fields. However, in the buggy program only `CustId` 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, 3.0, 2010>` , the statement has now selected `<1, 2.0, 2009>` 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 indicate the kind of fix to be made. We implemented the semantics of each database statements of ABAP in Java, and hence could execute each database statement on the use/def data captured against each statement in the execution trace to evaluate the effect of each mutation.

In general, the mutations we consider is 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 and 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 10.

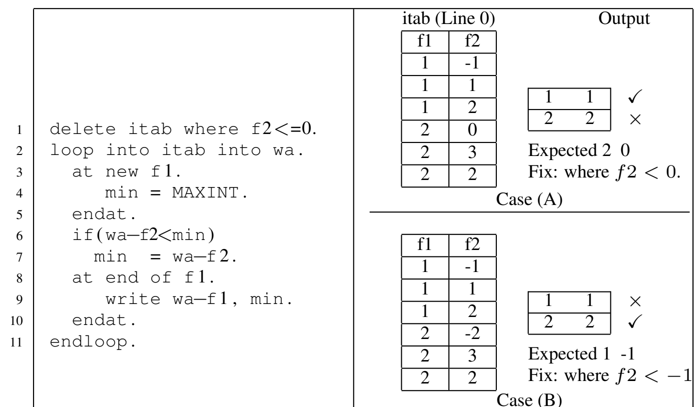

Figure 11: Example: Mutation vs. Corner Case Differencing

Mutability differencing can be effective in cases where corner case differencing is not. In Figure 11, minimum `f2` value is computed for each distinct `f1` value. Before this computation, deletion occurred with a condition on `f2` values $(f2 <= 0)$. In case (A), 1st and 4th rows are deleted by 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 difference can find this error as for `f1=2` the deletion of the row was done on the corner case of the condition $f2 <= 0$, 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>`. However, 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 $(f2! = -2 \wedge f2 == -1 \wedge f2 <= 0)$.

However, mutability differencing is not strictly more powerful that corner case differencing, as will be shown later in `ZROTC` experimental subject in Figure 15.

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

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 bad slices are computed based on key fields in the unwanted rows and good slices are computed based on key fields in the rest of the rows. The key-based slicing, followed by control 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 of more fields for all rows in the output is reported as a bug. Bad slices are computed for each of the rows, and instead of computing control difference of them, the common statements in all the bad 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 bad slices. For each statement in the common set, behavioral differencing method is applied to

```
1
2   Input: KV = A set of <field,value> pairs which
3   are missing from the output.
4
5
6   // Step 1
7   Slices = all output generating slices with
8               slice criteria = fields in KV
9
10  // Step 2
11  For each slice Slice ∈ Slices,
12     S += from end, find the first statement in Slice
13            where KV occurs in use but not in def.
14
15  // Step 3
16  For each stmt s ∈ S
17     if s in loop
18        start_point += find other instance of p
19                          that has KV in def & use
20        suspects = control execution trace
21           execution traces (in the loop)
22           starting from p (good) and start_points (bad)
23           differ
24     else
25        suspects += s
26
27  return suspects
```

Figure 12: Algorithm of Missing Row

determine a corner behavior of the statement which is present in all its occurrences in bad slices.

Missing Rows. The algorithm for missing row handling is presented at Figure 12. In this case the bug report contains the description of 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 values. 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 good slices computed for some existing rows. Thus, we analyze (by stepping backward) each good slice, to find the first statement occurrence (s) from end of slice 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 loop, all peer occurrences (say S) of s is determined in other loop iterations which has missing key-value pairs in its *use* and *def* set. The execution paths starting from the statement occurrences in S have the potential to produce the missing rows. Control difference of the execution paths starting from the statement occurrence in S and the execution path starting from s is presented as fault suspects.

To explain the case in loop we present the following example in Figure 13 with input data given below.

The two slices corresponding to the generated rows which go thru write statement instances 11^8, 11^{16} of line 11, with slice criteria `wa_itr-matnr` are $\{1^1, 2^2\}$ and $\{1^1, 2^{10}\}$. Step 2 finds $S = \{2^2, 2^{10}\}$. For each element in S, $start_point = \{2^{18}\}$, $Suspect = \{13^{22}\}$. The line 13 is pointed as the suspect here, and the information is presented to the user to check the block starting from 13^{22}. Note that, in the case of finding faults regarding missing row, differencing between the executation traces may need to be done. This is in contrast with the previous presented cases, where difference between slices are determined.

```
1  select * from zitr into table it_itr.
2  LOOP AT it_itr INTO wa_itr.
3      CLEAR : v_exrate.
4      vexrate = 'READ_EXCHANGE_RATE'(wa_itr-waers)
5      IF sy-subrc EQ 0.
6        IF v_exrate GT 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
17 endloop.
```

Figure 13: Example: Missing Row

ZITR

MATNR	Amount	WAERS	EXRATE
1	50	USD	47
2	100	EUR	66
3	100	UK	75

Output

MATNR	DispAmt
1	2400
2	6600

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

Missing KeyValue: $\langle MATNR, 3\rangle$

Figure 14: Input and output for the ABAP program illustrated in Figure 13

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 conducted experiments to evaluate the effectiveness of our core contributions - key-based slicing, semantic differencing and the analysis for missing rows in output. After describing the experimental setup, we present and discuss the results.

4.1 Experimental Setup

We used a set of real ABAP report programs from the field as our experimental subjects. Trouble-ticket history of a Maintenance Project was filtered to locate the tickets with ABAP report program faults, and the faulty versions along with the dependencies were chosen as subjects. The set size was kept small since the analysis results needed to be manually verified. The fault relevant source snippets for all the subjects have been provided in Figure 17. The codesnipptes for SUTAX, ZQFPR, ZFR052 cannot be provided due to confidentiality. To indicate the complexity of the subjects, lines of code and the size of execution (LOC/EXE) are provided in Figure 15.

To conduct the experiments following method was followed for each subject -

- 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.

Subject	LOC	EXE	Slice Size		Control Diff.		Semantic Diff.		Missing	Output Size	Timing in Secs
			Field-row	Key	Field-row	Key	Corner	Mutability [Space]			
RLS	2013	569	4	4	×	×	✓	✓ [1]		1	1
ZROTC	1066	2397	4	4	×	×	✓	×		1	46
RO13	1202	2948	8	8	✓	✓	-	-		3	23
IMAT	2661	3864	4	4	✓	✓	-	-		1	20
MMAT	1019	7251	5	4	✓	✓*	-	-		1	27
ORDER	1975	7386	13	12	×	✓	-	-		1	28
IINV	3154	5299	8	8	✓	✓	-	-		1	135
ZBMR	827	257	5	5	×	×	✓	✓ [1]		1	4
SUTAX	1662	4028	12	12	-	-	-	-		12	2
ZQFPR	1136	275	8	8	-	-	-	-	✓	1	1
ZFR052	944	520	4	4	-	-	-	-	✓	1	1
BABL	2795	367	6	6	-	-	-	-	✓	1	1
RV54	3492	1088	5	5	-	-	-	-	✓	1	1

- LOC: Lines of Code (Excluding the library)
- EXE: Execution Trace Size
- Slice Size: Average Incorrect Slice Size removing pure control statements in case of non-missing bugs, for missing bugs this is the slice size for output producing trace. The good slice size are not given here.
- Field-row: Slicing based on Field-Row Sensitive Algorithm
- key: Slicing based on key-based slicing on top of field-row sensitive algorithm
- Control Diff: ✓if there is any difference is slices, × otherwise. Note that this does not say whether the difference is the exact bug or not.
- If there is any control difference found between two field-row slices, it is possible that after doing key-based slice that difference is no more. This is attributed to the overapproximation of slice computation. But control diff of key-based slice can show another difference in slice. This is the case for MMAT.
- Space in Mutability differencing denotes that, based on the heuristics for the statement, this is total number of mutation that needs to be tried. In both the cases, this number is 1, signifying the effectiveness of the approach in practice.
- Missing: These programs showed missing row bug
- Output Size: This is potential fault points that are found based on the analysis.
- Time: Time for the analysis to complete

Figure 15: Fault Localization Result

- Fault observation was specified as a pair of precise slicing criteria and associated category (i.e. incorrect, missing)
- Analysis Algorithms were executed with the above two inputs. Both, Field-Row sensitive control differencing, and key-base differencing algorithms were executed for each of the cases. In the cases where neither of these located a differenced results, both the semantic differencing algorithms were executed, and finally if semantic differencing didn't succeed either and the slicing criteria was associated to a case of missing, the missing analysis was performed.
- 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 that was verified to be the fault. For the subject named SUTAX, we were not able to point out a single statement. Verification revealed that the input itself was wrong. Our key-based slice of size 12 was the best that we could present, and the input statement in the slice was related to the fault. Such a high percentage of bugs being detected by differencing or missing analysis may be attributed to the fact that the subjects are from production ERP systems.

4.2.1 Key-based slicing

In 2 cases (MMAT and ORDER), the key-based slice was smaller than the field and row sensitive slice. The slice sizes are mentioned in the field-row, and key columns of Figure 15. A look at the relevant code snippets reveals that field and row sensitive slices had over-approximately included the delete statements. In case of ORDER difference was vital in identifying the fault, as delete statement was key to the bug. In case of MMAT (* marked in Figure 15), field-row sensitive slice found a difference in the good and bad slices, but that was not the exact faulty line. The difference of key-based slices showed the faulty line.

4.2.2 Semantic Differencing

In 6 cases Control differencing (either just on field-row sensitive slices or key-based slices) was sufficient in identifying the fault. In 3 cases, where the control 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 17(a), the move-corresponding X to Y statement moves the values from structure X to Y for the common fields. The correct assignment that needs to be done here is 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. Missing assignment was a.f4 to c.f4 in the move statement. The error is only noticed when a.f4 is different from b.f4. In an iteration which produced correct values both a.f4 and b.f4 were same, in an incorrect iteration b.f4 has a non-zero value. Both good and bad threads had same sequence of statements having second move-corresponding and not the first. So, control differencing failed to discover any difference. The corner-case differencing tried two corner (overflow and overwriting, cf. Figure 9) 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 the same value in good thread and different value in the bad thread. This is also located using a mutation where move-corresponding statement is mutated to a sequence of move statements and deleting the move corresponding to c.f4 = b.f4 as f4 as the only related field in the slice.

ZROTC In Figure 17(b) in Line 5 the overflow occurs in the assignment statement. `w_jtab.a` has smaller size than `w_itab.a`. The overflow is visible in bad threads as non-zero digits were truncated due to overflow, whereas in good threads only zeros were truncated which did not produce any ill effect to the computed result. This statement will be there in both good and bad slices, and therefore control-flow differencing will not able to catch this behavioral difference. The corner-case differencing will be able to catch this behavior as this is one of the corner-case that is determined in assignment statement (Figure 9). Note that, the fix to this bug is not the 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 17(g) 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 Corner case differencing and mutability differencing both find

```
1   PERFORM batch_heading_babl.
2   ...
3   DELETE gt_output WHERE f_new IS INITIAL.
4
5   PERFORM aendbelege_lesen.
6   ..
7   output gt_output.
```

(a)

```
8   SELECT vfkk~fknum vfkp~fkpos INTO CORRESPONDING
9       FIELDS OF TABLE c_object_key FROM ..
10  l_header_key = c_object_key[].
11  DELETE ADJACENT DUPLICATES FROM l_header_key
12     COMPARING fknum.
13  c_object_key[] =  l_header_key.
14  SELECT * FROM vfkp INTO CORRESPONDING FIELDS
15         OF TABLE c_vfkp FOR ALL ENTRIES IN c_object_key
16        WHERE fknum = c_object_key-fknum AND ..
17  it_vfkp[] = l_it_vfkp.
18
19  table_output it_vfkp[].
```

(b)

Figure 16: Code Snippet for Missing Row in (a) RV54 (b) BABL

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 were small as we perform heuristic 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 16(a). The bug reported that, for some keys, rows were missing from the output. In the program, there was a delete statement (Line 3) 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 output. Then by performing a backward traversal in the slice we found the DELETE statement that was deleting with respect to the keys related to the missing rows. 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. Similar effect of delete statement is seen for the BABL program where we show the slice that is producing an output row in Figure 16(b). In this case, the delete adjacent statement is deleting the row corresponding to the missing key. This table is later joined together with another table which in turn did not get the row corresponding to the missing keys. The code for other two programs (ZQFPR,SFR052), showing missing behavior, are not provided here due to confidentiality.

5 Related Work

Static Program Slicing Dor et.al. [8] looked at ERP systems from a program analysis perspective to study the impact due to a customization change. They use static program analysis based techniques as a solution. In the context of their problem the static slice sizes do not matter much. However for the problem of fault localization it will lead to over-approximate slices. In their recent work Litvak et.al. [18] 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

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)

RO13

```
1  append fs_final to it_final_temp.
2  clear fs_final—kbetr.
3  fs_final—netpr = ...
4  fs_final—effpr = ...
5  MODIFY it_final_tmp FROM fs_final
6    TRANSPORTING netpr effpr kbetr
7    where infer = fs_final—infer.
8  APPEND LINES OF it_final_tmp
9                 TO it_final.
10 write_itab it_final.
```

(c)

IMAT

```
1  clear yiseg1.
2  loop at yiseg2.
3    append yiseg1.
4    move—corresponding yiseg2
5            to yiseg1.
6    move: zw_farbe to yiseg1—farbe.
7  endloop.
8  write_itab yiseg1.
```

(d)

MMAT

```
1  SORT belege BY matnr bwkey
2              cpudt cputm buzei.
3  LOOP AT belege.
4   ON CHANGE OF belege—matnr
5          or belege—bwkey.
6     CLEAR mengt.
7   ENDON.
8  mengt = mengt + belege—menge.
9  MOVE mengt TO belege—mengt.
10 MODIFY belege.
11 endloop
12 write_itab belege.
```

(e)

ORDER

```
1  SORT db_tab BY kunnr matnr vbeln.
2  delete ADJACENT DUPLICATES
3        FROM db_tab
4        COMPARING kunnr matnr vbeln.
5  LOOP AT db_tab.
6    satz_tab—netwr = db_tab—netwr.
7    ...
8    append satz_tab.ENDLOOP.
9  LOOP AT satz_tab.
10   rueck_tab—netwr = satz_tab—netwr.
11   ...
12   append rueck_tab. ENDLOOP.
13 LOOP AT satz_tab.
14   write rueck_tab—netwr.ENDLOOP.
```

(f)

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      ...
```

(g)

IINV

```
1    type T2: werks matnr...
2    SORT  T2 BY  MATNR WERKS.
3    LOOP AT T2.
4      AT NEW MATNR.
5
6      ENDAT.
7      AT NEW WERKS.
8        move t2—matnr to t_header.
9        append t_header.
10     ENDAT.
11   ENDLOOP.
```

(h)

Figure 17: Code Snippets

of large structure variables. However, their work is focused on static analysis and also the absence of row sensitive analysis can pose as severe limitation to the precision of the slicing information.

Dynamic Program Slicing Korel and Laski [17] introduced the notion of a dynamic slice. 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. [16] worked on inter-procedural dynamic slices. Zhang and Gupta [31, 30] improvised the algorithms for dynamic slice computation in the presence of arrays, structures and pointers for complex real world programs. Our presented dynamic slicing techniques go few steps further; in the context of database operations, we present row and field sensitive slicing, and extend it with key-based slicing.

Program Analysis in data-intensive systems Sivagurunathan et.al. [24] recognized the challenge posed to the slicing techniques by programs with I/O. They introduced pseudo variables into the program to make the hidden I/O state accessible to the slicer. Tan and Ling [25] recognized the same challenge when the programs access data stored externally. They followed a similar solution and introduced a set of implicit variables to capture the influence among I/O statements and validated their approach on flat file based storage. Willmor et.al. [28] extended the same approach to cover relational databases, and addressed field-sensitivity to make the influence more precise. In his Ph.D thesis Cleve [5] formulates the space of program analysis and transformation for data-intensive systems. Proposing that the complexity of database-aware program slicing task lies in the nature of the data manipulation language (DML), he categorizes DMLs into native, built-in, embedded, and call-based. For DMLs he generalizes the interaction with the host program via input-host variables, and output host variables. As in previous approaches he extracts the data-dependencies between the host language and DML into direct and indirect mappings, and inserts psuedo instructions for them in the SDG for enabling static dependency analysis. Hainaut et.al. [11] and Cleve [5] looked at dynamic analysis for embedded and dynamic SQL (such as in JDBC) in the context of data intensive systems. However their work focuses on resolving the dynamic queries and collecting its execution trace at the runtime. The continuity of dependence between the host, the data manipulation language (DML), and the datastore is a necessary aspect of slicing data-centric programs. However, for the objective of fault-localization in data-centric programs (in languages like ABAP), it is not sufficient. In this paper, we show that naive handling of data base operations (on physical or internal) would lead to imprecise slices, and present novel slicing techniques to compute precise slices.

Differencing based Fault Localization Most differencing based fault localization techniques can be placed in the general framework defined by Renieris and Reiss [21]. The notion of spectrum (abstract trace) was introduced by Reps et al. [22] for acyclic, and intraprocedural path spectra. Harrold et al. [12] generalized the notion of spectrum and proposed spectra based on several program features - branch, complete-path, data-dependence, output, and execution trace. Tarantula [15] explored the role of visualizing the hit or count statistical metrics for passing and failing spectra, with the idea of translating the passing and failing causes to a colored spectra. Zeller [29] 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 [21] 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 have been proposed to provide users with minimal information required to explain counter-examples resulting from model checking. In [23], the authors introduce the notion of *neighborhood of counter-examples* which can be used to understand the cause of counter-examples. A different approach based on game-theoretic techniques is put forth in [14] where counter-examples are augmented into *free segments* (choices) and *fated segments* (unavoidable). Errors are most likely to be removed by careful selection of free segments. In [3], errors in programs are localized 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 [10] where errors are localized to program statements absent in all positive examples and present

in all counter-examples leading to the same error condition. Based on the idea of detecting the divergence as the cause of the counter-example, [9] developed a technique that uses a distance matrix and constraint manipulations to pin-point the variable operations that led to the divergence. The technique is applied to one counter-example in the program. In [4, 27], the divergence between one counter-example and positive example is used to localize the error.

Without dealing explicitly with the database statements (such as sort, delete, delete adjacent, and read etc.) the precision of the above techniques will suffer significantly. The granular blowup of such statements via the corresponding internal behavior model, and the differences in it corresponding to the passing and failing cases, allows us to overcome this limitation.

Another major applicability constraint for spectra techniques is imposed by ERP system maintenance environments relating to the restriction on automated executions to collect passing and failing spectra. For the domain of data-centric programs, we overcome this constraint by retrieving passing and failing threads of execution from a single execution. Mani et.al. [20] applied this technique to retrieve passing and failing traces from a single execution to compute repair recommendations. However, the forward propagating dynamic taint analysis they use does not consider precision of propagation through array / table like structures. Further, unlike our work, they treat all loop-iterations as peers, where data-centric programs have an abundance of aggregating computations that group the iterations based on some fields, thus leading to imprecision.

Mutation Analysis In the area of testing, mutation technique is used to generate faulty programs from the correct program ([7, 2, 19]) to study the path divergence of faulty programs from the correct programs. The large number of mutants generated there is irrespective of the nature of the fault, also the technique there is not to use mutation to fix or identify faults. The closest to our work is the recent work by Debroy et. al. ([6]), where mutation based approach is taken for repairing the faults that are localized by the fault localization tool to suggest a fix. Unlike our technique the mutant are not used for localizing 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.

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.

References

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

[2] 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.

[3] T. Ball, M. Naik, and S. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 97–105. ACM, 2003.

[4] 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.

[5] A. Cleve. Program Analysis and Transformation for Data-Intensive System Evolution. 2009.

[6] V. Debroy and W. E. Wong. Using Mutation to Automatically Suggest Fixes for Faulty Programs. In *Proceedings of the Third International Conference on Software Testing, Verification and Validation (ICST)*, pages 65–74. IEEE, Apr. 2010.

[7] 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.

[8] 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 the 2008 international symposium on Software testing and analysis*, pages 97–108. ACM, 2008.

[9] A. Groce. Error explanation with distance metrics. In *Proceedings of TACAS*, 2004.

[10] A. Groce and W. Visser. What went wrong: Explaining counterexamples. *Model Checking Software*, pages 121–136, 2003.

[11] J. Hainaut and A. Cleve. Dynamic analysis of SQL statements in data-intensive programs. 2008.

[12] 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.

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

[14] H. Jin, K. Ravi, and F. Somenzi. Fate and free will in error traces. In *Proceedings of TACAS*, 2002.

[15] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 273–282, New York, NY, USA, 2005. ACM.

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

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

[18] S. Litvak, N. Dor, R. Bodik, N. Rinetzky, and M. Sagiv. Field-Sensitive Program Dependence Analysis. 2010.

[19] 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.

[20] S. Mani, V. Sinha, P. Dhoolia, and S. Sinha. Automated support for repairing input-model faults. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 195–204. ACM, 2010.

[21] M. Renieres and S. Reiss. Fault localization with nearest neighbor queries. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings*, pages 30–39, 2003.

[22] 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.

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

[24] Y. Sivagurunathan, M. Harman, and D. S. Slicing, i/o and the implicit state. *Proceedings of 3rd International Workshop on Automatic Debugging (AADEBUG'97)*, Volume 2 (009-06), 1997.

[25] H. Tan and T. Ling. Correct program slicing of database operations. *IEEE software*, 15(2):105–112, 1998.

[26] G. Venkatesh. The semantic approach to program slicing. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, page 119. ACM, 1991.

[27] C. Wang, Z. Yang, F. Ivančić, and A. Gupta. Whodunit? Causal analysis for counterexamples. *Automated Technology for Verification and Analysis*, pages 82–95, 2006.

[28] D. Willmor, S. Embury, and J. Shao. Program slicing in the presence of database state. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 448–452. IEEE, 2004.

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

[30] X. Zhang and R. Gupta. Cost effective dynamic program slicing. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, page 106. ACM, 2004.

[31] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 319–329. IEEE, 2003.