ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/221307432

Precise detection of un-initialized variables in large, real-life COBOL programs in presence of unrealizable paths

CONFERENCE PAPER · JANUARY 2011

DOI: 10.1109/ICSM.2011.6080812 · Source: DBLP

CITATIONS
2

READS
35

3 AUTHORS:

Rahul Jiresal
University of British Columbia - Vancouver
2 PUBLICATIONS **10** CITATIONS

SEE PROFILE

Adnan Contractor
Tata Research Development and Design Centre
2 PUBLICATIONS **3** CITATIONS

SEE PROFILE

Ravindra Naik
Tata Consultancy Services Limited
8 PUBLICATIONS **8** CITATIONS

SEE PROFILE

All in-text references underlined in blue are linked to publications on ResearchGate, letting you access and read them immediately.

Available from: Rahul Jiresal
Retrieved on: 18 January 2016

Precise Detection of Un-Initialized Variables in Large, Real-life COBOL Programs in Presence of Un-realizable Paths

Rahul Jiresal
Tata Research Development and Design Centre
Pune, India
rahul.jiresal@tcs.com

Adnan Contractor
Tata Research Development and Design Centre
Pune, India
adnan.contractor@tcs.com

Ravindra Naik
Tata Research Development and Design Centre
Pune, India
rd.naik@tcs.com

***Abstract*—Using variables before assigning any values to them are known to result in critical failures in an application. Few compilers warn about the use of some, but not all uses of un-initialized variables. The problem persists, especially in COBOL systems, due to lack of reliable program analysis tools. A critical reason is the presence of large number of control flow paths due to the use of un-structured constructs of the language.**

We present the problems faced by one of our big client in his large, COBOL based software system due to the use of un-initialized variables. Using static data and control-flow analysis to detect them, we observed large number of false positives (imprecision) introduced due to the *un-realizable paths* in the un-structured COBOL code. We propose a solution to address the realizability issue. The solution is based on the summary based function analysis, which is adapted for COBOL Paragraphs and Sections, to handle the *perform-through* and *fall-through* control-flow, and is significantly engineered to scale for large programs (single COBOL program extending to tens of thousands of lines). Using this technique, we noted very large reduction, 45% on an average, in the number of false positives for the un-initialized variables.

***Keywords*-un-initialzed variables; program analysis; defect detection;**

I. Introduction

In software systems the use of un-initialized variables may lead to system outages, as the variables may have garbage values. Such failures may imply that the system does not behave as per its specifications. Further, the garbage value of an un-initialized variable may flow through the program resulting in incorrect computations, and the severity of the failure may increase multi-fold. The outcome of running such a program may simply be undeterminable. Tracking down a reason for this kind of failures and localizing the fault caused due to an un-initialized variable is a non-trivial task. As reported by one of our client, some of the faults took several months to be localized and traced. Certainly, such long time-frames for localizing a fault due to un-initialized variables are not desirable for the systems that are in production.

For a major client in the retail banking domain, several branches of the bank use a single instance of the application that *services* the transaction requests made by the branches. The transaction requests are maintained in separate queues depending on the category of the request. This model improves the performance of the system, by saving the time to load and initialize the application for every service request. However, several simultaneous executions of the application can cause problems. Typically, such code is made re-entrant by using proper initializations of all data-items (variables). Absence of proper initializations, however, may lead to defects that are difficult to trace, especially when there are simultaneous, multiple executions of the application. The data-space of one execution may be used by another execution, if there are improper initializations. Unfortunately, due to the unstructured nature of COBOL constructs that result into introduction of several paths reaching the use of a variable, it is very difficult for the developer to ensure proper initializations.

A generic and simplest solution to this problem could be to initialize each variable just after its declaration. However, in large banking applications, which are primarily data-driven and the variables may be considerably large records, initializing all the variables may prove expensive in time. Therefore, these variables are typically initialized only when needed. Another reason, in some cases, is that the code was migrated from one platform to the other, and there is difference in the way the two platforms handle the initializations of the variables, leading to the use of un-initialized variables on the migrated platform.

A. Sample Failure Description

Honoring the confidentiality agreement that TCS has with its clients, their names, details of the source code and real defects are not mentioned in the paper. However, we explain sample failure by explaining the situation that arose in the Recurring Deposit (RD) account closure scenario.

The Bank reported a defect that only in some cases, the Recurring Deposit accounts would show discrepancies when they were closed. The customer account and the teller account would not match. The development team was not able to even reproduce the problem at the development

site for a few months. After a very careful and lengthy analysis, the team was able to simulate it. The error would occur when two separate branches simultaneously tried to close two different Recurring Deposit accounts. The causal analysis revealed that variables that held the closing amounts were not initialized along certain paths (amounts were not re-fetched from database). The re-fetch from database was avoided to enable better performance. Hence, the closing amount of one Recurring Deposit account would flow to the second Recurring Deposit account, leading to incorrect teller account update for the second account. These paths were encountered rarely, only when two bank branches attempted to close two Recurring Deposit accounts simultaneously and the requests were serviced by the same application instance; hence it was extremely difficult to reproduce and detect this defect.

To detect the un-initialized variables, even in the presence of un-realizable paths, we propose to adapt the technique outlined by M. Sharir and Amir Pnueli [1] to COBOL programs by treating PARAGRAPHs and SECTIONs as procedures, executing inter-procedural analysis by calculating summary analysis of each procedure and propagating the context information from each procedure call point to the procedure.

In the next section, we discuss the state-of-the-art to solve the un-initialized variables detection problem and describe the conventional solution in section III. The problems arising due to unrealizable paths in COBOL and the proposed solution are discussed in sections IV and V. We present the heuristics to further reduce the number of false positives and prioritize the *important* results in section VI. The results of our solutions are presented in Section VI.

II. Related Work

A large focus to detect un-initialized variables is by employing dynamic analysis. In the paper [2], Dewar et al. focus on an improvement in the language Ada that ensures automatic initialization of the variables that are not initialized. They also present a language extension for verification of these variables at run-time.

There are numerous run-time memory checking tools that can be used to detect references to un-initialized variables. Some of these tools are *Valgrind* [3, 4], *Purify* [5], *MemCheck* [3], *Dr. Memory* [6]. These tools are based on the concepts of intercepting and inspecting memory management APIs like malloc(), new(), free(), delete(). For every instance of the allocated memory, these tools check the access of the variables, and report an error if a variable is accessed without getting defined. However, such runtime memory checking tools are dependent heavily on the instruction set and the platforms. Another drawback in using such tools in the client production line is that they hamper the performance, and need much larger memory. For example, *Purify* reports around five-fold slower execution times [5]. For COBOL applications running on Mainframe systems, such tools can add to the cost by consuming the expensive Mainframe CPU cycles.

Making use of both dynamic and static analysis techniques, Nguyen et al. [8] present a combination of compile-time analysis along with source code instrumentation for run-time checking, but only for Fortran programs. In this approach, the authors propose inter-procedural array data-flow analysis. If the compile time information is insufficient, they initialize array elements with special values and instrument their uses with a check to assert the legality of the access. Even though the instrumentation needed is small, using a similar approach may not be feasible in our client environment due to the difficulties in executing the code on expensive and a myriad of platforms.

Compilers for some of the modern languages report the uses of un-initialized variables at compile-time either with errors (example is Java) or warnings (GCC). The known problem in compile-time static analysis is the lack of precision because of complex control and data flows in the code. Even in the absence of un-realizable paths, the precision of un-initialized variables reported by the GCC-COBOL compiler is questionable.

In his paper [7], Z. J. Czech puts forth an algorithm to find un-initialized variables during compile time. However, his solution does not guarantee the detection of *all the references to undefined variables*. It, therefore, becomes a moot point if it actually helps locating and removing *all* potential defects.

Our work for detecting uses of un-initialized variables in COBOL applications is based on the approaches mentioned in the technical report [1] by Micha Sharir and Amir Pneuli. We adapt their approach by treating the Paragraphs and Sections in COBOL as procedures, except that they share the data space of the program. When a Paragraph / Section is *performed*, the context of the *performing* Paragraph is remembered and propagated to the performed Paragraph to compute the precise analysis information. This approach reduces the number of false positives in the computed results by not propagating the analysis information of the performed Paragraph along *unrealizable paths*. The next two sections discuss the issues of the *unrealizable paths* as applicable in COBOL.

III. Un-Initialized Variables Detection And Issues

A variable is termed as un-initialized at a use-point if there is at least one program path from the entry point to the use-point through which the variable does not get a definition, meaning, along that program path, the variable is not assigned a value.

Static data-flow and control-flow program analysis techniques [12] are used to detect un-initialized variables in the code. It requires Dataflow functions to be defined for each node type of interest in the program, and Meet operation

on nodes having more than one incoming edges. It also requires the Control Flow Graph (CFG) of the program to be constructed. A CFG is a directed graph in which nodes represent statements, and an edge u → v represents possible flow of control from u to v. The technique involves traversing the CFG and solving corresponding data flow functions for each node to obtain the analysis information at the nodes.

At a given program point *P*, the strategy used to detect un-initialized variables is dependent on the *defined* set at *P*. The *un-initialized variables set UV* at *P* is stated as the variables being used at *P* but are not *defined* till *P*. We explain the strategy in further detail as follows.

i) For every program point, we compute the defined set using the following flow functions.

   Dataflow function:

   $$DF_o(n) : I_{in}(n) \cup \{v \mid v \in Gen(n)\}$$

   where,

   $I_{in}(n)$: set of defined variables carried from predecessor of node n

   $Gen(n)$: set of variables occurring in *lval* context at node n

   Meet operation at a node:

   $$M(n) : D_0 \cap D_1 \cap \ldots \cap D_i$$

   where,

   $M(n)$: Meet function at CFG node n for incoming paths 0 to i

   D_i : Set of defined variables flowing into node n along control path i

   Summarizing in words, at every node, all the variables that occur in *lval* context are identified, and carried forward to the next node. In an expression, any variable to which a value is being written to is said to be in *lval* context. At the node where multiple paths meet, only the variables that are defined along **all** the incoming paths are carried forward.

ii) For every statement in the program, fetch the expressions and identify all variables that are referenced in those expressions. The set of un-initialized variables derived from the defined set is computed as follows.

   $$UV(n) = \{v \mid v \in R_n \ and \ v \notin D_n\}$$

   where,

   $UV(n)$: set of un-initialized variables at node n

   R_n : set of variables referenced at node n

   D_n : set of defined variables till node n

Thus, the un-initialized variables are a set of variables that have not been defined at least along one path.

A. Automation and Results

The solution for the data-flow equations was encoded using *Prism* [11] workbench. The queries for computing the *defined* set and *un-initialized* set were programmed using the workbench primitives, and the CFG between paragraphs due to performs, gotos and fall-throughs, called as *Para-flow graph* was computed precisely.

By applying the above strategy on a large COBOL program AB0000 (single computation unit of over 60K lines of code), we observed about 6000 instances of un-initialized variables. Though the strategy is conservative in nature, this was too large a number to be of practical use for the development team to analyze the results and identify the subset of variables that are truly uninitialized.

We performed a careful analysis of a subset of the results. The study revealed the presence of considerable number of false positives in the results. Certain variables that were actually initialized were also reported. A more detailed analysis revealed that many of these variables were reported because there was no definition reaching the use point through one or more *unrealizable* paths.

In the next section, we discuss the problem of unrealizable paths and our potential solution.

IV. Unrealizable Paths

A COBOL program consists of one or more Paragraphs or Sections. A *Paragraph* or *Section* in COBOL is a set of statements represented by a *label*. A *Section* can consist of one or more *Paragraphs*. Using the PERFORM statement, the execution control is transferred to a Paragraph (Paragraph is *performed*). Similar to a sub-routine, a Paragraph, after executing statements contained in it, returns the control to the statement immediately after the PERFORM statement. The target of PERFORM statement is either the Paragraph name or a range of Paragraph names, the latter representing execution of a range of paragraphs. For simplicity, we treat a paragraph range as a basic cohesive unit and the COBOL program as a collection of paragraph ranges.

In the context-insensitive inter-procedural analysis (described in section III), the analysis information of a subroutine from one call point may get propagated to other callers of the same subroutine, since the calling context is not remembered. The information propagation thus takes the *unrealizable* path. In an application with sub-routines, the *unrealizable paths* are the paths occurring in the *Control Flow Graph* in the absence of calling context information.

Similarly, in case of COBOL, within a procedure, the analysis information from one PERFORM point may get propagated to other performers of the same Paragraph, resulting into an unrealizable path.

Consider the COBOL code in Figure 1 and a relevant part of its *Control Flow Graph* in Figure 2.

In figure 2, the nodes marked as *E* and *X* represent the *Entry* and *Exit* nodes of a paragraph, respectively. Also, the

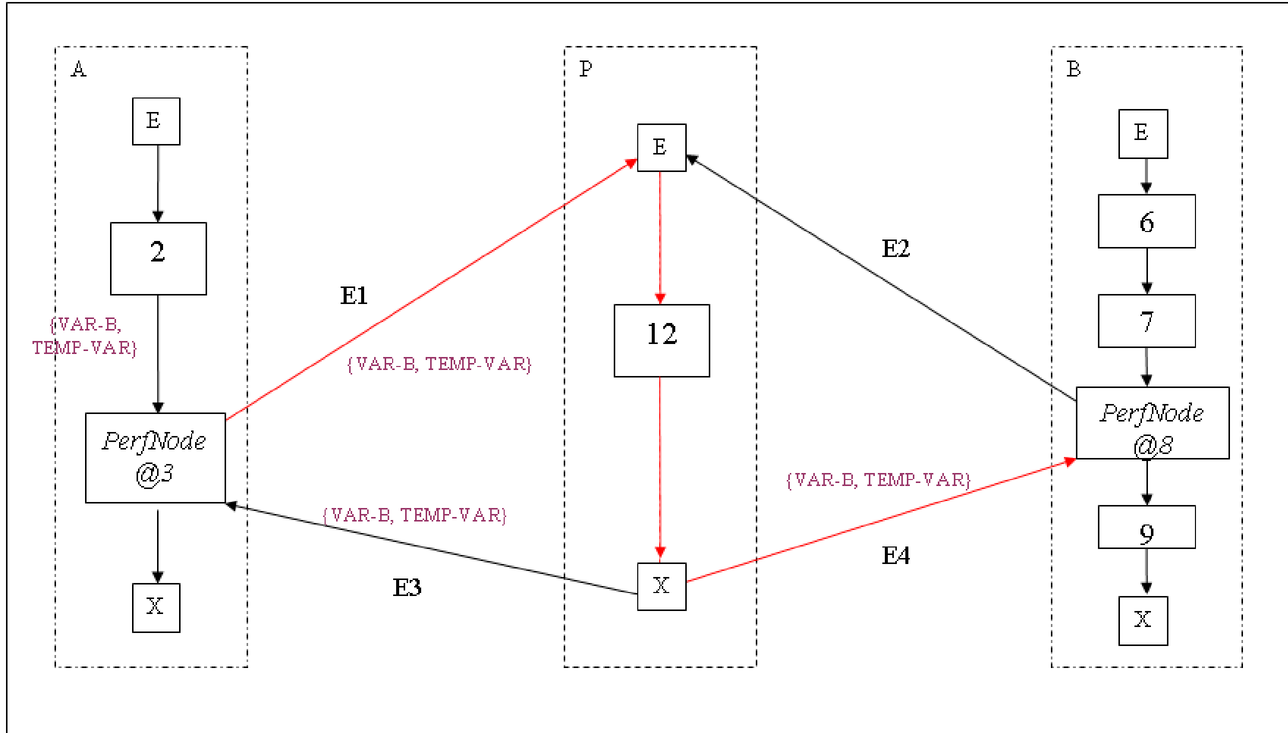

Figure 2. Control Flow Graph for code snippet in Figure 1

```
 1. A.
 2.    MOVE TEMP-VAR TO VAR-A
 3.    PERFORM P.
 4.
 5. B.
 6.    MOVE TEMP-VAR TO VAR-B
 7.    MOVE TEMP-VAR TO VAR-A
 8.    PERFORM P
 9.    MOVE VAR-B TO MOVE-VAR.
10.
11. P.
12.    MOVE TEMP-VAR TO VAR-P.
```

Figure 1. Sample code for Unrealizable paths in COBOL code

Perform statements at line 3 and line 8 are represented by *PerfNodes*. The sets of un-initialized variables flowing along the control paths are shown along the edges.

The PERFORM statements at line 3 and line 8 transfer control to the paragraph P. The incoming edges to P are E1 and E2 and outgoing edges are E3 and E4.
Here the actual paths are:

E1−E3 & E2−E4

But, there are Unrealizable paths too:

E1−E4 & E2−E3

Multiple contexts from perform nodes (from lines 3 and 8) are merged at the entry point of the paragraph P (line 11) and the analysis information at exit point (end of line 12) is propagated to all the perform points (end of lines 3 and 8). This introduces potential false positives in the results, as there is no provision to pair up corresponding incoming and outgoing edges.

By applying the original strategy mentioned in section III, the variable VAR-B, at line 9, is reported as un-initialized. This is clearly a false positive, as VAR-B gets a definition at line 6, before the paragraph P is performed. In the mentioned strategy, the contexts of both the *performers* are merged at the entry point of the *performee*. Because of this, the context of B at line 7 - the fact that VAR-B is defined at line 6 - is lost when it is merged with context of A at line 3. Therefore, after the Perform at line 8, VAR-B is treated as un-initialized. By definition, the variable should get a value on all paths leading to the point under consideration (line 9), but the unrealizable path E1-E4 has no definition for VAR-B.

To eliminate these false positives, one solution is to resort to context-sensitive analysis of the Paragraph ranges, or subroutines in general. The proposed strategy is described in the next section.

V. Context-sensitive Detection Of Un-initialized Variables

To make the analysis context sensitive, it is required to store the context at each invocation of the subroutine or

Paragraph. Conceptually, each *perform* statement needs to have its own copy of the *paragraph range* that it performs. If separate copies of the information are created for each *perform*, it is possible to avoid the problem of unrealizable paths.

There are mainly two approaches to achieve context-sensitivity. The first is the Cloning approach, involving *inlining* of the *performees* into *performers*. For obvious reasons, this approach proves to be inefficient as it replicates the *Paragraph ranges* at each *Perform* point. This approach does not scale-up, and does not work in the presence of recursion.

The second approach involves computing and storing summaries and contexts. In summary-based analysis, a summary is created for each paragraph range, which summarizes the effects of the paragraphs in that range. The algorithm has two phases - A bottom-up phase in which the summaries are incorporated into the *performers* and a top-down phase in which the context is propagated into the *performees (performed ranges)*.

A. Solution

While computing analysis information for the *defined* set as described in Section III, the global, context-insensitive approach considers the program as a whole unit and solves the data-flow equations without considering the context. This approach results in the unrealizable paths issue.

To resolve this issue, the program is considered to consist of paragraph ranges. Using the *Para-Flow* graph, which is a directed graph, in which nodes represent a paragraph range, and an edge u → v represents possible flow of control from the perform statement *u* to the paragraph range *v*, the paragraph ranges are identified.

Analysis information is first computed for the leaf level paragraph ranges in the *Para-Flow* graph, while treating each paragraph range in isolation. This information is stored as symbolic summary and is communicated up the *Para-Flow* graph to the callers of the range. Further, our approach is designed to remember the context (incoming information) at each perform node. This context, together with the summary of the paragraph range being performed, is used to calculate outgoing information at the bottom of the perform node.
The analysis information at each perform node is computed as:

$$N_{Bottom} : [N_{top} \text{ - } KillSum_p] \cup GenSum_p$$

where,

N_{Bottom}	:	Set of defined variables at bottom of perform node N
N_{top}	:	Set of defined variables at top of perform node N
$KillSum_p$	:	Kill summary information at exit of paragraph range p
$GenSum_p$	:	Gen summary information at exit of paragraph range p

For the above equations, we defined the *Gen* and *Kill* summary sets for a paragraph range *p* as follows.

$$GenSum_p = Defined(X)$$
$$KillSum_p = \emptyset$$

where,

X	:	Exit node of paragraph range p
$Defined(X)$	:	Set of defined variables at the exit node X

The Kill *Summary* set is an empty set since a variable, once *defined* is treated as always *defined*.

Applying the new program analysis strategy to compute the *un-initialized variables* set for the example in Figure 1, the results of the analysis information at the top and bottom of Perform Nodes (*PerfNodes*) are as follows.

Gen Summary of paragraph P:
{`VAR-P`}

Kill Summary of paragraph P:
{}

Defined Set (Context) at top of *PerfNode@3*:
{`VAR-A`}

Defined Set at bottom of *PerfNode@3*:
{`VAR-P, VAR-A`}

Defined Set (Context) at top of *PerfNode@8*:
{`VAR-A, VAR-B`}

Defined Set at bottom of *PerfNode@8*:
{`VAR-A, VAR-B, VAR-P`}

According to the Step ii of the strategy described in Section III, the set of un-initialized variables at line 9 is,
$UV(9)$ ={ `TEMP-VAR` }

Thus, it can be seen that at line 9, after resolving the *realizability* issue, VAR-B is not reported as un-initialized. At the same time, the strategy ensures that the no un-initialized variables are missed out, that is, there are no false negatives.

VI. Improving The Results Using Heuristics

The proposed and prototyped approach guarantees high precision. Even then, for large COBOL programs, the number of reported instances of un-initialized variables is fairly large (we illustrate the results in section VII). As discussed earlier in the paper, a variable needs to be defined on all the paths reaching its use to be flagged as *initialized*. As a large program can have hundreds of paths by which the assignment reaches the use, the variable will be flagged as un-initialized if it is not defined even along one path, however

practically infeasible that path is. This conservative approach ensures that the strategy does not miss any potential defect due to an un-initialized variables.

In practice, while fixing defects, the developer is reluctant to manually analyze a large number of un-initialized variables. Therefore, we prioritize the results using a pre-defined set of heuristics. Some of the reported un-initialized variables are filtered and marked as *low priority* using the heuristics. We are in the process of improving existing heuristics and evolving new ones. Few heuristics to prioritize the un-initialized variables are described in the subsequent sub-sections.

A. Removal Of Bad Code Smells

Unlike a PERFORM, a GOTO to a label acts as a non-returning jump to a paragraph. While a PERFORM statement is considered to be a good programming practice to reuse a portion of code, a GOTO is considered as a poor (unstructured) programming practice. This is because it can lead to difficult-to-track flow of control. In COBOL, additionally, the jump to a Paragraph because of a GOTO results in a *fall-through* of control to the subsequent Paragraph, leading to potentially incomprehensible behaviour. Agrawal et al. in their paper [9] have presented a statistic that about 50% of the *transaction outages* are caused by undesirable GOTOs in the program. Such undesired GOTOs are the reason behind the unintended *fall-throughs* in the code. They also state that removal of such bad GOTOs, and effectively *fall-throughs*, can reduce the number of un-initialized variables. The reason is that many of the unintended, accidental paths are avoided.

We run the tool [9] developed by Agrawal et al. on the large industrial programs, and fix the potential defects reported by the tool. These include removal of *out-of-range GOTOs*, unintended *fall-throughs* and dead paragraphs in the code. As per our observation, the un-initialized variables detected in the subsequent analysis are reduced by 3-5% for the large programs.

B. Instances In Utility Function Calls

In COBOL, parameters in the calls to other programs are passed by *reference*. In the called programs like database library or formatting library, the parameters are updated / modified in the called program. Such variables are usually not initialized in the calling program before the call. Even though, technically, such variables are *un-initialized* at the call-point, the developers typically want to ignore them during the analysis. A heuristic proposed by us allows developers to list all the *Utility programs* and for each such program, the parameters that are to be ignored while reporting. We mark such variables as *low priority* un-initialized variables.

After filtering the *low priority* un-initialized variables from the results, we observe an average 20% reduction in the results.

C. Prioritization Based On Fan-ins Of Paragraphs

A *Fan-in* of a paragraph is defined as the number of incoming edges for that paragraph in the *perform-graph* of the program. A *perform-graph* is defined as a graph $G < E, N >$ where N represents the paragraphs of the program, and there exists an edge E_{xy} from node N_x to N_y, if the paragraph x performs paragraph y.

If a paragraph has higher *fan-in* than others, the execution of that paragraph is more frequent than that of others. Hence instances of un-initialized variables in higher *fan-in* paragraphs may lead to higher chance of defects at runtime. Correcting initializations of such variables can lead to fixing more potential defects in less time. We sort all the un-initialized variables according to the *fan-ins* of the paragraphs they are contained in.

This heuristic proves very useful to the developers to prioritize their focus to the more important potential defects. Results of prioritizing the instances of un-initialized variables according to the *fan-in* of the paragraphs are shown in the results section.

VII. Results

We prototyped and applied the proposed strategy on a set of programs from our clients execution environment. These programs were of varying lengths from a few thousand lines of code to around 80 KLOC. The prototype was executed on JVM 5.0 running on Windows Vista (32 bit) with 4GB of installed memory. The results were shared with the development team for verification.

Table I
Comparison Of Un-initialized Variables Detected

Program	**Context insensitive analysis**		**Context sensitive analysis**		**False positives eliminated**		**Overall Improvements[3] (%)**
	1[1]	2[2]	1	2	1	2	
AB0000	1317	6090	737	2945	580	3145	51.6 (44)
CR0000	1173	4313	838	3001	335	1312	30.4 (29)
CR0025	336	871	187	437	149	434	49.8 (44.3)
CR7070	74	253	30	60	44	193	76.3 (59.5)
AB0024	59	120	48	103	11	17	14.2 (18.6)

1. Unique un-initialized variables
2. Total instances of un-initialized variables
3. Values in the brackets represent the % improvement on unique variables

We compared the results of un-initialized variables before

and after the realizibility change with respect to execution times and the number of variables detected.

Table I shows the comparison for the number of un-initialized variables detected using context insensitive analysis and context sensitive analysis. Sub-column 1 represents the unique variables that were detected, and sub-column 2 represents the total *use-points* of these variables. The third column shows the number of false positives eliminated by using the context sensitive analysis. The average improvement in the precision of the analysis is nearly 45%. The figures in the parentheses in the last column represent the percentage of the unique instances of un-initialized variables.

Although the results shown in Table I show a substantial improvement in the precision of un-initialized variables, the absolute number of the un-initialized variables is still high. Therefore, the heuristics discussed in the previous section are used to filter and prioritize the un-initialized variables that have lesser probability to surface as a defect.

Table II shows the results of the un-initialized variables post applying the heuristics.

In the second column, there is an evident reduction of around 3-5% in the reported variables after the removal of bad code smells. The third column represents the numbers after removal of reported un-init variables in the utility calls. An average reduction of 40% and maximum reduction up to 64% was observed after applying this heuristic. This also indicates that a large number of variables are passed as parameters to be updated / modified in the *called* programs.

Table II
RESULTS AFTER APPLYING THE HEURISTICS

Program	**Context sensitive analysis**		**After removal of bad code smells**		**After removal of Utility function call instances**	
	1	2	1	2	1	2
AB0000	737	2945	706	2779	585	1942
CR0000	838	3001	803	2872	689	965
CR0025	190	440	177	422	114	231
CR7070	30	60	29	59	18	37
AB0024	11	17	11	17	7	12

Table III shows the number of instances of un-initialized variables grouped by the fan-ins of the paragraphs they are contained in. As it is seen from the table, the larger programs (AB0000, CR0000) have more variables in the higher fan-in ranges. On the other hand, smaller programs do not have a large number of such instances because of the absence of high fan-in paragraphs. Also, the paragraphs with a zero fan-in are the entry points of the programs and are certainly executed.

The developer, typically, wants to fix the code where there are more likely potential defects. In the absence of run-time data, fan-in is a useful heuristic to find out more likely defects.

Table III
NUMBER OF UN-INITIALIZED VARIABLES FOR FAN-IN GROUPS

Fan-in	**AB0000**	**CR0000**	**CR0025**	**CR7070**
0	172	78	0	1
1-5	1615	636	215	36
6-10	45	124	0	0
10-20	69	78	8	0
21+	41	49	8	0

As depicted in Table IV, if we ignore the smaller programs, the execution time for the detection of un-initialized variables has increased by an average of around 65%. This increase is because of the time taken by the tool to compute the paragraph summaries and saving / loading the context information of the paragraphs. Also, the time taken depends on the number of paragraphs and the complexity of the Paraflow graph rather than the LOC of the programs.

Table IV
RUNTIMES OF UN-INITIALIZED VARIABLES DETECTION

Program	**LOC**	**# of Paras**	**Context insensitive analysis Runtime**	**Context sensitive analysis Runtime**	**% Increase**
AB0000	56,721	1004	1 m 55 s	3 m 12 s	66.9
CR0000	77,736	1275	2 m 27 s	4 m 6 s	67.36
CR0025	15,944	450	27 s	43 s	59.25
CR7070	2,961	84	2 s	5 s	150
AB0024	2,628	69	$\sim$ 1 s	$\sim$ 2 s	100

VIII. Future Work

As discussed in the results section, the approach proposed by us has increased the precision in the analysis of COBOL programs in the presence of unrealizable paths. Handling of unrealizable paths using the context-sensitive analysis will potentially improve precision in other data-flow queries for formal verification and slicing of COBOL programs.

Since uses of un-initialized variables represent critical faults leading to many different kinds of failures that are difficult to trace and debug, we aim to improve the solution further with help of more heuristics and dynamic analysis. We also plan to productize the prototype and measure the business gains achieved by the development team. One option is to measure the reduction in the number of failures that are attributed to the use of un-initialized variables, over a substantial period of time.

A. More Heuristics

Error handling is an important aspect of every enterprise application and constitutes a considerable chunk of the application code. The detection of error usually diverts the control from the main, functional path to error handling module. Analysis of the un-initialized variable reports for various programs revealed that there is substantial number of variables reported as un-initialized along the error paths because they dont have any definitions along the functionally valid paths. Elimination of such error paths is a potential heuristic that will reduce the number of un-initialized variables.

The effect of a *function* is communicated to the environment through its output variables. They include actual parameters to external function calls, variables being written to the database or rendered on the screen, and messages that are sent from the function. A possible heuristic is to detect automatically or enable the developer to select the output variables of the function. The heuristic will be used to report only the un-initialized variables that lie in the program slice based on the selected output variables.

B. Dynamic Analysis

Since un-initialized variables are very sensitive to the control-flow paths that are present in the program, we propose to use techniques akin to dynamic analysis. After instrumenting the code to log the paths, the idea is the execute the application for various functional test scenarios, collect the path logs and use them to prune the paths for control-flow analysis. Though the precision of un-initialized variables due to such pruning of control-flow paths depends on the *completeness* of the test scenarios, it is found to be a practically useful. Y. Smaragdakis and C. Csallner [10] report their experiences of building sequence of more powerful combinations of static and dynamic analyses for bug finding.

Another initiative in pipeline is the detection of un-initialized paths. Amongst all the paths leading to the use of an un-initialized variable, the variable may not have a definition only along a few paths. It would be a big help to the developer if he is provided with guidance about the paths along which he needs to add the definition, saving him lot of efforts to locate the path.

IX. Conclusion

In this paper, we have discussed the problems faced by one of our biggest clients in his applications, primarily due to the use of un-initialized variables. While using static program analysis techniques to detect un-initialized variables, a large number of false positives were observed because of the unrealizable paths existing in the COBOL code. We adapted the summary based inter-procedural analysis for COBOL, and engineered it specifically to address COBOL. On an average, 45% improvement in the precision was achieved after applying the approach. To help the developer focus on more important potential defects, we proposed the use of heuristics to filter out and prioritize the results. This filtration has yielded an average of 40% reduction in the instances that the developer has to manually analyze.

Acknowledgments

We acknowledge the co-operation of the TCS development team who own the business applications for helping us with analyzing and validating the results. We also acknowledge Shrawan Kumar for his reviews of the strategies and assistance to adapt the summary-based interprocedural analysis, and Pavan Chittimalli and Hemanth Makkapati for his help with the implementation.

References

[1] M. Sharir and A. Pneuli, *Two Approaches to Inter Procedural Data Flow Analysis*, Technical Report, New York University, September 1978.

[2] R. Dewar, O. Hainque, D. Craeynest and P. Waroquiers, *Exposing Uninitialized Variables: Strengthening and Extending Run-Time Checks in Ada*, Proceedings of Ada-Europe '02 Proceedings of the 7th Ada-Europe International Conference on Reliable Software Technologies, 2002.

[3] J. Seward and N. Nethercote, *Using Valgrind to detect undefined value errors with bit-precision*, in Proc. of the USENIX Annual Technical Conference, 2005, pp. 22.

[4] N. Nethercote and J. Seward, *Valgrind: A framework for heavyweight dynamic binary instrumentation*, in Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 07), Jun. 2007, pp. 89100.

[5] R. Hastings and B. Joyce, *Purify: Fast detection of memory leaks and access errors*, in Proc. of the Winter USENIX Conference, Jan. 1992, pp. 125136.

[6] D. Bruening, Q. Zhao, *Practical Memory Checking with Dr. Memory*, 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2011.

[7] Z. J. Czech, *Efficient Implementation of Detection of Uninitialized Variables*, The Computer Journal (1988) 31 (6): 545-549.

[8] T. Nguyen, F. Irigoin, C. Ancourt, and F. Coelho, *Automatic Detection of Uninitialized Variables*, Proceedings of 12th International Conference on Compiler Construction, pp 217-231, 2003.

[9] A. Agrawal and R. Naik, *Towards Assuring Non-recurrence of Faults Leading to Transaction Outages an Experiment with Stable Business Applications*, Proceedings of the 4th India Software Engineering Conference, 2011.

[10] Y. Smaragdakis and C. Csallner, *Combining Static and Dynamic Reasoning for Bug Detection*, Lecture Notes in Computer Science, Volume 4454/2007, 1-16, 2007.

[11] *PRISM : Static Data and Control Flow Analysis Workbench*, Technical Report, Tata Research Development and Design Centre, Pune, 2008.

[12] John. Kam and Jeffrey. Ulman, *Monotone Data Flow Analysis Frameworks*, Acta Informatica 7, 305-317, 1977.