# Semantic Diff: A Tool for Summarizing the Effects of Modifications

*Daniel Jackson**
*School of Computer Science*
*Carnegie Mellon University*
*daniel.jackson@cs.cmu.edu*

*David A. Ladd*
*Software Production Research Dept.*
*AT&T Bell Laboratories*
*ladd@research.att.com*

*This paper describes a tool that takes two versions of a procedure and generates a report summarizing the semantic differences between them. Unlike existing tools based on comparison of program dependence graphs, our tool expresses its results in terms of the observable input-output behaviour of the procedure, rather than its syntactic structure. And because the analysis is truly semantic, it requires no prior matching of syntactic components, and generates fewer spurious differences, so that meaning-preserving transformations (such as renaming local variables) are correctly determined to have no visible effect. A preliminary experiment on modifications applied to the code of a large real-time system suggests that the approach is practical.*

## 1 Introduction

A maintainer faces two principal obstacles when modifying a piece of code: first, understanding what the existing code does, and second, understanding the effects of a potential change. Most reverse engineering tools are designed for the first, and thus cannot provide a comparative analysis of two versions of a program. We have built a tool that helps overcome the second obstacle: given two versions of a procedure, it generates a report summarizing the semantic effects of the modification. This paper explains how the tool works, and reports on some preliminary experience applying it to the code of a large real-time system.

A maintainer can benefit in several ways from our tool. By running the tool after having made a change, he can correlate the tool's summary against his intent; discrepancies between the two are likely to indicate flaws. This might reduce the incidence of "fix on fix", when a modification's sole purpose is to fix a bug introduced in an earlier modification. A summary of the effects of a change is most important later, however, when the modified code is being modified again. Heavily modified code is far harder to understand than fresh code, often because maintainers fail adequately to document the effects of their changes. Our tool can relieve this burden substantially; it can provide the structure of the documentation for a change (which a maintainer might subsequently embellish).

## 2 Practical Semantic Diff

What are the key features of a practical semantic differencing tool? First, the vocabulary of its report should be semantic and not syntactic. A procedure's semantic effect is a relation between its inputs and outputs; the difference between two versions must thus be expressed in terms of input-output behaviour. This is not to deny the value of syntactic reports. A slice indicating affected lines of code is good for some applications, but does not tell a maintainer how clients of a procedure will be affected; it merely reduces the size of the program from which this information must be inferred.

Second, the tool should be fully automatic, fast and easy to use. Clearly this rules out any kind of theorem-proving (which would call for lemmas to be provided by the user and searches that might not terminate). But it also

*Address: School of Computer Science, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213. Phone: (412) 268–5143. Fax: (412) 681–5739. This research was sponsored in part by a Research Initiation Award from the National Science Foundation (NSF), under grant CCR-9308726, by a grant from the TRW Corporation, and by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA), under grant F33615-93-1-1330. The paper was set in typefaces generously provided by Bitstream, Inc.

militates against global analyses, such as interprocedural dataflow. The program we have been analyzing contains procedures that are, on average, a few hundred lines long; finding the differences between two versions of such a procedure takes our tool (a research prototype coded in ML) about ten seconds. The entire program, however, contains over a million lines of code. Programs of this size may take days just to compile.

Third, the summary produced should be accurate. Determining the exact semantic differences between two versions of a program is an undecidable problem, so no automatic tool can be perfectly accurate of course. A semantic diff tool, unlike textual diff, must therefore make errors: missing genuine differences, or finding spurious differences where none in fact exist.

Program analyses that are designed for compilers cannot tolerate errors that compromise the correctness of the code, so they are always conservative. Existing techniques that compare versions of a program have grown out of work on compiler optimization, and adopt this assumption, always erring on the side of spurious differences, but never missing a real difference. For example, techniques to reduce the cost of regression testing by eliminating needless tests or by not retesting unaffected code (such as Binkley's [2] and Bates and Horwitz's [1]) can safely find differences where none exist, since this only results in unnecessary test executions, but must never miss differences, which might lead to inadequate testing.

When the output of the tool is intended for human consumption, a different kind of accuracy may be called for. Always erring in one direction, by overestimating differences, can be catastrophic if the user is swamped with spurious messages. And the occasional omission of a difference is less critical; a human user is more forgiving than a compiler, and can make use of results that are not completely reliable.

For this reason, we have taken the radical step of trying to maximize the accuracy of our tool's output by compromising its soundness. A tool based on the standard approach to comparison regards two versions as identical only when their dependence graphs are isomorphic [1,2]. This demands that the nodes of the graphs match syntactically; the only semantic abstraction is in the relative ordering of statements. As a result, almost any change to the code will be flagged as a difference, including all the common meaning-preserving transformations applied frequently by programmers during maintenance: renaming local variables, adding or removing temporary variables for subexpressions, packaging variables into a record structure, and so on.

Our tool would find no difference in any of these cases. It would also find no difference when an off-by-one error is fixed. Luckily, though, most large systems contain little intricate code, and the most troublesome modifications arise not from such intricacies but rather from larger structural peturbations.

## 3 What The Tool Does

Our tool compares two versions of a program one procedure at a time. Each procedure is analyzed independently, with only local information. Global analysis, as explained above, is often infeasible in large systems, and procedure-wise comparisons seem to be effective.

The first step is to obtain, from each of the two versions of the procedure, an approximation to its input-output behaviour. This approximation is essentially a binary relation over the set of variables accessed by the procedure: its arguments, results and any global variables it reads or writes. One variable is related to another if its value after execution of the procedure depends on its value before.

Consider, for example, a C procedure that updates a global variable *TOT*:

```
void add (int x) { /* old version */
    if (x != HI) {TOT = TOT + x;
    else TOT = TOT + DEF;}
```

The dependence relation for this procedure contains the following pairs:

(*TOT*,*TOT*), (*TOT*,*x*), (*TOT*,*DEF*), (*TOT*,*HI*)
(*x*,*x*), (*DEF*,*DEF*), (*HI*,*HI*)

saying that the value of *TOT* after depends on the values of *TOT*, *x*, *DEF* and *HI* before, and the values of *x*, *DEF* and *HI* all depend on their values before (as they are unchanged).

Suppose we flipped the branches of the if-statement, and rewrote the conditional expression, intending to make no change to the behaviour of the program (but unwittingly writing = for ==):

```
void add (int x)  {/* new version */
    if (x = HI) {TOT = TOT + DEF;
    else TOT = TOT + x;}
```

The relation now contains the pairs

(*TOT*,*TOT*), (*TOT*,*DEF*), (*TOT*,*HI*)
(*x*,*HI*), (*DEF*,*DEF*), (*HI*,*HI*)

The tool compares the two relations and list any pairs that appear in one but not the other. For this example, our tool reports:

*new version removes dependences*: *x on x*, *TOT on x*
*new version adds dependences*: *x on HI*

This nicely expresses the flaw with the modification: *x* no longer depends on itself but on *HI* instead (since the if-test

now assigns *HI* to it), and since the initial value of $x$ is always lost, *TOT* no longer depends on $x$.

The actual analysis is a bit more sophisticated than this example suggests. When a variable has some structure, the tools tracks dependences of its components separately (but does not distinguish the elements of an array). This is critical in practice, because most procedures take few arguments, each with a complex structure. The most common kind of modification in the code we examined looks like this:

```
struct msg {int a, b;};

void foo (struct msg *p, struct msg *q) {/* old */
   q-> a = p->a;}

void foo (struct msg *p, struct msg *q)  { /* new */
   if (p->a == 0)
      q-> a = p->b;
   else
      q->a = p->a;}
```

The change is thus to handle an exceptional case by assigning a different field $p{->}b$ to $q{->}a$ when $p{->}a$ is zero. This is summarized as:

*new version adds dependence*: $q{->}a$ *on* $p{->}b$.

The tool also incorporates the names of called procedures in the dependence calculation, so that if a component depends on the same variables as before, but is computed by calling different procedures, a difference is reported. This is a rather crude trick, but it appears to work well in practice, giving a reasonable approximation while avoiding interprocedural analysis.

The current prototype ignores aliasing, treating pointers like any other name. So an assignment to a field $p$ of a structure referenced by a pointer argument $m$ such as

$$m{->}p = x$$

would yield a dependence $(m{->}p, x)$, even though m may not reference the same storage location it referenced initially. The prevalence of aliasing in code is hard to predict, but is certainly very dependent on the application and coding style. We have not come across many examples that call for alias analysis, but we believe it to be an essential part of a practical tool. We have implemented similar tools that combine this style of dependence calculation with alias analysis [9], and if carefully done, the loss of accuracy need not be too great.

## 4 How The Tool Works

The key approximation of the tool is to ignore the actual state transitions of a procedure, and instead view its behaviour as a dependence relation between the inputs and outputs. To capture, in addition, information about how the final value of a variable depends on the behaviour of called procedures and the values of constants, the relation can include pairs that relate a variable to the names of procedures of constants. The dependence relation $D$ is thus:

$$D\text{: } \mathit{VarName} \leftrightarrow (\mathit{VarName} \cup \mathit{ProcName} \cup \mathit{ConstName})$$

where a pair of variables $(x,y)$ in $D$, for instance,says that the value of $x$ after execution depends on the value of $y$ before, and a pair $(x,f)$ consisting of a variable and a function name says that the final value of $x$ depends on the behaviour of the function $f$.

The tool also calculates the set of variables that are potentially modified, for calculating control dependences:

$$M\text{: } \mathit{Set}\ (\mathit{VarName})$$

The dependence relation of the whole procedure can be built compositionally from the dependence relations of its constituent statement, by the following rules, in which $D(S)$ denotes the dependences of a statement $S$ and $M(S)$ denotes its modifications:

$$D(S;T) = D(T) \circ D(S)$$
$$D(\textit{if } b \textit{ then } S \textit{ else } T) = D(S) \cup D(T) \cup (M(S) \cup M(T)) \times \{b\}$$
$$D(\textit{while } b\ S) = D(S)^* \cup ((M(S) \times \{b\}) \circ D(S)^*)$$
$$M(S;T) = M(S) \cup M(T)$$
$$M(\textit{if } b \textit{ then } S \textit{ else } T) = M(S) \cup M(T)$$
$$M(\textit{while } b \textit{ do } S) = M(S)$$

The $\circ$ operator is relational composition, so the first rule expresses the transitivity of dependences: that if $x$ depends on $y$ for statement $T$, and $y$ depends on $z$ for statement $S$, then $x$ depends on $z$for the sequence $S;T$.

The dependence rule for if-statements uses the modification calculation to add control dependences to any variable potentially modified in either branch. The dependence rule for loops accounts for their unbounded nature by forming the transitive closure of the dependences of the body. In practice, a couple of iterations is sufficient, and the cost (in time and space) of the dependence calculation is linear in the length of the procedure.

There are two primitive statements to consider: a simple assignment and a procedure call. For an assignment $x{=}y$, $x$ depends on $y$ and all other variables are unchanged, and thus depend on themselves:

$$D(x = y) = \{(x,y)\} \cup \{(\alpha,\alpha) \mid \alpha \in \mathit{VarName} \backslash \{x\}\}$$

and $x$ is the only modified variable

$M(x = y) = \{x\}$

Fields are treated just like variables, so that a variable naming a structure with three fields, one of which is also a structure with two fields say, would be treated as four distinct variables. Since we avoid interprocedural analysis, we make the worst case assumptions for a procedure call. For the statement

$x = P(y,z)$

for instance, we make $x$ dependent on $y$ and $z$, and in addition add cross-dependences of all the fields pointed to by $y$ and $z$ on each other. The variables $y$ and $z$ themselves are not modified, since parameters are passed by value in C.

Having calculated the dependences of the two versions of the procedure, the tool reports any differences. If the dependence relations are $D_{old}$ and $D_{new}$, it would report:

*new version adds dependences*: $D_{new} \setminus D_{old}$
*new version removes dependences*: $D_{old} \setminus D_{new}$

Some details have been omitted in this brief explanation; the tool must first, for example, eliminate dependences of variables local to the procedure (and thus not visible to a caller).

In the absence of aliasing and pointers, the dependence construction is conservative. While there may be spurious dependences, no real dependences are omitted. But the results of the comparison are in no sense conservative: there may be differences reported when none exist, and no report when there are differences. In our application, however, the relative frequency of incorrect reports is what matters, and techniques that provide some kind of guarantee (such as comparison of program dependence graphs, which can promise never to miss a difference) do so only at the expense of generating far more spurious reports.

If aliasing occurs, the dependence construction itself is no longer conservative, since dependences may arise through sharing that is not visible to the tool. Alias analysis complicates the dependence construction enormously [8,9], and its benefits in this application have yet to be determined.

The implementation does not actually calculate dependences over the syntax tree; we gave the rules in this form to illustrate their simple compositional nature. Rather, it constructs a flow graph and propagates dependences iteratively until a fixed point is reached. Recently, we have devised a simple technique [11] for calculating dependences using a representation similar to the program dependence graph [4], but generalized to handle procedural abstraction.

## 5 Preliminary Experiments

Our tool was developed following a week-long study of a sample of modifications applied to a large real-time system, whose maintenance currently employs a few hundred developers. The tool itself is written in Standard ML of New Jersey (in about 8000 lines of code). It takes two versions of a procedure written in C and generates a report listing the differences between their dependence relations.

We examined the modification history for the past year of the files in a single directory, comprising about 5000 lines of code, divided into 35 files, each containing a single procedure. By looking at the abstracts written by the maintainers summarizing their changes, and by examining the code itself with the Seesoft visualization tool [3], we sifted out about 20 suitable candidates (the remaining modifications being insertions of large amounts of fresh code).

Our tool fails to report semantic differences in 2 cases, both of which were quite subtle; in one, for example, the order of application of two bit-shifting operations was reversed. In the remaining 18 cases, the differences are detected. In only one case was a spurious dependence pair reported that a human reader could eliminate by examining the code.

In most cases, there were also a few dependence pairs that a maintainer would not have expected, but whose elimination would have required extensive global analysis. Arguably, these are genuine differences worth flagging, because they can only be ignored given strong assumptions about the environment of the procedure. In a system this size, it is not possible to extract coherent versions corresponding to the states of the entire program before and after a local modification, so it seems appropriate that the maintainer should be informed of these differences, dismissing them at his own risk.

Two examples of running the tool on actual modifications are reproduced in the appendices; the code has been simplified and the format of the tool's output has been slightly altered to make it more readable. In the first case (Appendix 1), a procedure set 8 out of about 300 fields of a structure. The modification added code to handle some exceptional conditions in which fields were to be set instead to various default values. The procedure had about 100 lines of code and the changed introduced 30 more. The tool calculated about 400 dependence pairs for each version, and found 14 pairs in one but not the other.

The second example (Appendix 2) illustrates a case in which a spurious difference is reported. An inspection had suggested an improvement in coding style to a procedure that concatenated several strings to a field of a message. On appending each string, it incremented the total length

by the length of that string, eventually setting another field to the final value of the length. The modification replaced this series of increments by two statements: one to save the starting address of the field, and another to subtract it from the final address and assign the result to the length field. There was thus no semantic difference. The procedure was about 200 lines long before the change, and about 60 lines were removed. The tool found a difference of one pair between the two cases: a dependence of the length field on the starting address. Eliminating it would require a knowledge of arithmetic, which is beyond the scope of almost all program analysis tools. (Actually, there are two length fields, so there are two differences.)

The lack of alias analysis in our tool turned out, perhaps surprisingly, not to be a serious problem. We came across no case in which it led to spurious difference reports. There are several reasons for this. First, although the code we examined is full of pointers, aliases occur relatively infrequently: they are used primarily to achieve the effect of call-by-reference in a language that only provides call-by-value. Second, a poor approximation in calculating the dependences of one version tends to matched with an equally poor approximation in the other, and thus the flaws cancel out. Only when the modification to the code affects an aliased variable does aliasing matter, and this seems to be rare. Despite all this, alias analysis would not have added many new spurious reports, so we are likely to add it to a future version of the tool.

## 6 Related Work

The scope of a modification may be determined most directly by syntactic comparison of the two versions. The UNIX diff tool, for example, attempts to match the lines of two files, printing out any unmatched lines. Since most large programs are developed under some kind of source control mechanism, determining which lines have changed is not usually the problem, and finding ways to present the information matters more. The maintainers of the system we studied keep a modification database that records every line that is added or deleted; we used the Seesoft visualization tool [3] to display the code with each modification (whether an addition or a deletion) highlighted in a different colour. We could view a whole directory on a single screen, find interesting modifications by looking for various colours, and then zoom in to examine the code. Without Seesoft, our experiment would have taken much longer than a week.

These tools are line-based, so even a lexical change with no syntactic effect will appear as a modification. There are a number of tools that generate syntax tree representations of code; this suggests finding differences between versions by comparing their trees. Cdiff [7], for example, compares checksums of trees produced by CIA [5]. Since the smallest unit of executable code tracked by CIA is a function, the most Cdiff can say is whether or not a function's parse tree has changed.

All syntactic tools have a severe limitation: they only show the syntactic scope of a change, even though modifying one line of the code might affect the behaviour of many lines, some quite remote from the site of the change. Representing programs with program dependence graphs eliminates this problem, since the repercussions of a change can be determined conservatively by following dependence edges from the modified statements through the graph, labelling all those statements on the way as potentially affected. By comparing the dependence graphs of two versions of a program, the statements that might be affected by a modification can be identified. This can be exploited to reduce the cost of regression testing, by running tests on a slice of the new version and by eliminating unnecessary test cases [1,2].

This approach is not good for our application. First, it gives the scope of a change but not its effect. Instead of identifying syntactic components that are affected by the change, our tool reports on the actual observable effects, in terms of input-output dependences.

Second, the comparison is still largely syntactic, since the graphs are compared node-by-node, each node being a syntactic statement. The dependence graph representation of a program is barely an abstraction semantically; the only information it discards is the syntactic ordering of independent statements. So almost any syntactic change, even if it has no semantic effect, will be seen as a difference. A tool based on dependence graphs would have found no commonality between the two versions of the first example in Section 3, for instance, since the swapping of the branches of the if-statement renders them incompatible.

Our dependence representation, in contrast, is a very weak approximation to the meaning of the program. As a result, the difference report may both omit actual differences and include spurious ones. The detection of differences is far more accurate than could be achieved by comparing dependence graphs, however, since the number of missed differences is small in practice, and our tool generates far fewer spurious reports.

Our approach has another advantage over graph comparison: it does not require any prior syntactic matching. We can take any two versions of a procedure, even if they have almost nothing in common syntactically, and still find significant commonalities if the computations share dependences.

Griswold and Notkin have pioneered a completely

different approach to modification, focusing on tools for transforming code rather than analysing it after the fact [6]. The meaning-preserving transformations that confound graph comparison approaches can be performed automatically, ensuring no semantic effect by checking certain syntactic conditions. A renaming of a local variable, for example, would be allowed only when the new name does not clash with existing names. The real benefit of this approach, however, is not in manipulating individual procedures, but in major restructurings. These are extremely rare in systems like the one we studied, partly because they are so hard to do safely. We therefore see Griswold and Notkin's approach as complementary to ours, being suited for a different kind of modification.

The dependence representation described in this paper is derived from our earlier work on Aspect [9,10], a tool that detects bugs in code by comparing the dependences of a procedure's code to dependences declared in its specification.

## 7 Future Work

The most significant source of spurious difference reports is procedure call, for which we make worst case assumptions. Global analysis may be infeasible for very large programs, but some information about the dependences of called procedures is needed.

The most promising solution involves specifications. Instead of turning to the code of a procedure, the tool would look up its specification in a centralized database. The database would be constructed incrementally, the user being prompted with questions about procedures (which argument is modified? what does it depend on?) as the need arises. This decouples the analysis from the code of the called procedure itself, which may be hard to obtain in a compatible version, or may not even exist yet. It also allows the user to improve the accuracy of the tool, by encouraging "leaps of faith", telling the tool to ignore certain dependences that are deduced from the code, but which the user knows are absent. Finally, these specifications could be compared to the code they purport to represent (by the method described in [10]), so that inconsistencies could be detected automatically.

## Acknowledgments

Eric Sumner helped initiate this project. Brendan Cain provided his expertise with the modification databases and change management systems, and helped us find our way round the source code. David Weiss gave us helpful comments on a draft of the paper.

## References

[1] Samuel Bates and Susan Horwitz, "Incremental Program Testing Using Program Dependence Graphs", *Proc. ACM Conf. on Principles of Programming Languages*, 1993.

[2] David Binkley, "Using Semantic Differencing to Reduce the Cost of Regression Testing", *Proc. International Conf. on Software Maintenance*, 1992.

[3] S.G. Eick, J.L. Steffen, E. Sumner Jr., "Seesoft: A Tool for Visualizing Line-Oriented Software Statistics", *IEEE Trans. on Software Engineering*, 18(11), pp. 957–968, November 1992.

[4] J. Ferrante, K. Ottenstein and J. Warren, "The Program Dependence Graph and Its Use in Optimization", *ACM Trans. on Programming Languages and Systems*, 9:3, July 1987.

[5] J.E. Grass and Y.F. Chen, "The C++ Information Abstractor", *Proc. USENIX C++ Conf.*, San Francisco, CA, pp. 265–278, 1990.

[6] W.G. Griswold and D. Notkin, "Automated Assistance for Program Restructuring", *ACM Trans. on Software Engineering and Methodology*, 2(3), July 1993.

[7] J.E. Grass, "Cdiff: A Syntax Directed Diff for C++ Programs", *Proc. USENIX C++ Conference*, Portland, OR, pp. 181–193, 1992.

[8] Susan Horwitz, Phil Pfeiffer and Thomas Reps, "Dependency Analysis for Pointer Variables", *ACM Conf. on Programming Language Design and Implementation*, 1989.

[9] Daniel Jackson, *Aspect: A Formal Specification Language For Detecting Bugs*, Technical Report MIT/LCS/TR-543, Massachusetts Institute of Technology, Laboratory for Computer Science, June 1992.

[10] Daniel Jackson, "Abstract Analysis with Aspect", *Proc. International Symposium on Software Testing and Analysis*, Cambridge, MA, June 1993.

[11] Daniel Jackson and Eugene J. Rollins, *Chopping: A Generalization of Slicing*, Technical Report CMU-CS-94-169, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, July 1994.

## Appendix 1: First Example

In the examples, the two versions are shown overlaid, with deleted code (belonging to the old but not the new version) in *italic*, with a minus sign in the margin, and the inserted code (belong to the new but not the old) in **letter-spaced bold**, with a plus sign in the margin.

**Code**

```
void ISinc_prms( mcr_ptr )
register CP_MCR_PTR   mcr_ptr;
{
UCHAR          rd_inf_arry[2];
ISOLINE        org_l_inf; ...
USHORT         nbytes; ...
ISRET_VAL      ret_val;

        ret_val = ISgetprm( ISRDIPRM, rd_inf_arry, &nbytes );
        if ( ret_val == ISSUCCESS ) {
                mcr_ptr->cp_rdi_rcvd = 1;
                mcr_ptr->cp_rdir1_info = rd_inf_arry[0];
                mcr_ptr->cp_rdir2_info = rd_inf_arry[1]; }
        ret_val = ISgetprm( ISOLIPRM, (UCHAR *)&org_l_inf, &nbytes );
        if ( ( ret_val != ISSUCCESS ) || ( nbytes != 1) ) {
                mcr_ptr->cp_o_line_info = 255; }
        else {
                mcr_ptr->cp_o_line_info = org_l_inf.orig_line;
                ret_val = ISgetprm( ISCHNPRM, (UCHAR *)&mcr_ptr->cp_chpn, &nbytes );
                if ( ret_val == ISSUCCESS ) {
                        mcr_ptr->cp_len_chpn = nbytes;}
+               else {
+                       ret_val = ISgetprm( ISCGPPRM, (UCHAR *)&mcr_ptr->cp_chpn, &nbytes );
+                       if ( ret_val == ISSUCCESS ) {
+                               mcr_ptr->cp_len_chpn = nbytes; }}
        ret_val = ISgetprm( ISOCNPRM, (UCHAR *)&mcr_ptr->cp_ocdpn, &nbytes );
        if ( ret_val == ISSUCCESS ) {
                mcr_ptr->cp_len_ocdpn = nbytes; }
+       else {
+               ret_val = ISgetprm( ISCPNPRM, (UCHAR *)&mcr_ptr->cp_ocdpn, &nbytes );
+               if ( ret_val == ISSUCCESS ) {
+                       mcr_ptr->cp_len_ocdpn = nbytes; }}
        ret_val = ISgetprm( ISRDNPRM ISRGNPRM, (UCHAR *)&mcr_ptr->cp_rdpn, &nbytes );
        if ( ret_val == ISSUCCESS ) {
                mcr_ptr->cp_len_rdpn = nbytes; }
        return;
}
```

**Report**

```
function [ISinc_prms]
new version adds dependences:
        return value on ISCGPPRM, ISCPNPRM
        mcr_ptr->cp_rdpn on ISCGPPRM, ISCPNPRM, ISRGNPRM
        mcr_ptr->cp_len_rdpn on ISCGPPRM, ISCPNPRM, ISRGNPRM
        mcr_ptr->cp_ocdpn on ISCGPPRM, ISCPNPRM
        mcr_ptr->cp_len_ocdpn on ISCGPPRM, ISCPNPRM
        mcr_ptr->cp_chpn on ISCGPPRM
```

## Appendix 2: Second Example

**Code**

```
void ISinc2_s7x( msg_ptr, mcr )
register ACDCSMSGBUF          *msg_ptr;
register CP_MCR_PTR   mcr;
{
register int       i;
USHORT             length, tot_length;
UCHAR              *from_ptr, *to_ptr;
ISINCALL           tmp_incall;
ISCGPN             cpn;
–UCHAR                    tmp_char;
UCHAR              tmp_char2, tmp_char, *save_ptr;

        from_ptr = (UCHAR *)&msg_ptr->text.dcs_hdr;
        to_ptr = (UCHAR *)&tmp_incall;
        length =  msg_ptr->text.dcs_hdr.cp_length;
        for ( i = 0; i < length; i++) {
                *to_ptr++ = *from_ptr++;}
        from_ptr = (UCHAR *)&tmp_incall;
+       save_ptr = (UCHAR *)msg_ptr;
–       tot_length = sizeof(OSMSGHEAD) + sizeof(MGSHDR)
–                       + sizeof(VCXGRP_NUM) + sizeof(VCXMEM_NUM);
–       msg_ptr->text.dcs_hdr.opt_start = tot_length;
–       to_ptr = (UCHAR *)msg_ptr + tot_length;
+       msg_ptr->text.dcs_hdr.opt_start = sizeof(MGSHDR) + sizeof(VCXGRP_NUM)
+               + sizeof(VCXMEM_NUM);
        to_ptr = (UCHAR *)msg_ptr + sizeof(OSMSGHEAD) +
                sizeof(MGSHDR) + sizeof(VCXGRP_NUM) + sizeof(VCXMEM_NUM);
        *to_ptr++ = length;
        *to_ptr++ = (UCHAR) ISINCPRM;
        for ( i = 0; i < length; i++ ) {
                *to_ptr++ = *from_ptr++; }
–       tot_length = tot_length + length + 2;
        *to_ptr++ = 1;
        *to_ptr++ = (UCHAR) ISCPCPRM;
        *to_ptr++ = mcr->cp_callparty_cat;
–       tot_length = tot_length + 3;
        *to_ptr++ = 2;
        *to_ptr++ = (UCHAR) ISFCIPRM;
        *to_ptr++ = mcr->cp_fci1;
        *to_ptr++ = mcr->cp_fci2;
–       tot_length = tot_length + 4;
–       if ( mcr->cpcr.cpn_digcnt > 0 )
        if ( mcr->cpcr.cpn_digcnt != 0 ) {
                cpn.oe_ind = mcr->cpcr.cpn_digcnt % 2;
                cpn.nat_addr_i = mcr->cpn_i_nature;
                cpn.numbplan = mcr->cpn_i_nplanind;
                cpn.pres = mcr->cpn_i_dispind;
                cpn.screen = mcr->cpn_i_screen;
                cpn.spare1 = 0;
```

```
                length = ( mcr->cpcr.cpn_digcnt + 1 ) / 2;
                for ( i = 0; i < length; i++ ) {
                        tmp_char = mcr->cpcr.cpn_dig[i];
                        if ( ( tmp_char & 0xf0 ) == 0xa0 ) {
                                tmp_char = tmp_char & 0x0f;}
                        if ( ( tmp_char & 0x0f ) == 0x0a ) {
                                tmp_char = tmp_char & 0xf0; }
                        cpn.addsig[i] = tmp_char;
                        tmp_char2 = tmp_char >> 4;
                        tmp_char2 = tmp_char2 | (( tmp_char & 0x0f)<<4);
                        cpn.addsig[i] = tmp_char2;}
                length = length + 2;
                from_ptr = (UCHAR *)&cpn;
                *to_ptr++ = length;
                *to_ptr++ = (UCHAR) ISCGPPRM;
                for ( i = 0; i < length; i++ ) {
                        *to_ptr++ = *from_ptr++;}
-               tot_length = tot_length + length + 2;}
        if ( mcr->cp_o_line_info < 100 ) {
                *to_ptr++ = 1;
                *to_ptr++ = (UCHAR) ISOLIPRM;
                *to_ptr++ = mcr->cp_o_line_info;
-               tot_length = tot_length + 3;
-               if ( mcr->cp_len_chpn > 0 )
                if ( mcr->cp_len_chpn != 0 ) {
                        *to_ptr++ = mcr->cp_len_chpn;
                        *to_ptr++ = (UCHAR) ISCHNPRM;
                        from_ptr = (UCHAR *)&mcr->cp_chpn;
                        for ( i = 0; i < mcr->cp_len_chpn; i++ ) {
                                *to_ptr++ = *from_ptr++;}
-                       tot_length = tot_length + mcr->cp_len_chpn + 2;}}
-       if (mcr->cp_len_ocdpn > 0 )
        if (mcr->cp_len_ocdpn != 0 ) {
                *to_ptr++ = mcr->cp_len_ocdpn;
                *to_ptr++ = (UCHAR) ISOCNPRM;
                from_ptr = (UCHAR *)&mcr->cp_ocdpn;
                for ( i = 0; i < mcr->cp_len_ocdpn; i++ ) {
                *to_ptr++ = *from_ptr++;}
-               tot_length = tot_length + mcr->cp_len_ocdpn + 2;}
-       if ( mcr->cp_len_rdpn > 0 )
        if ( mcr->cp_len_rdpn != 0 ) {
                *to_ptr++ = mcr->cp_len_rdpn;
                *to_ptr++ = (UCHAR) ISRGNPRM;
                from_ptr = (UCHAR *)&mcr->cp_rdpn;
                for ( i = 0; i < mcr->cp_len_rdpn; i++ ) {
                        *to_ptr++ = *from_ptr++;}
-               tot_length = tot_length + mcr->cp_len_rdpn + 2;}
        msg_ptr->text.dcs_hdr.cp_type = 174;
-       msg_ptr->text.dcs_hdr.cp_length =  tot_length - sizeof(OSMSGHEAD);
-       msg_ptr->msghead.length = tot_length;
+       msg_ptr->msghead.length = ( to_ptr - save_ptr );
+       msg_ptr->text.dcs_hdr.cp_length =  (to_ptr - save_ptr) - sizeof(OSMSGHEAD);
}
```

**Report**

function [ISinc2_s7x]
new version adds dependences:
msg_ptr->msghead.length on msg_ptr
msg_ptr->text.dcs_hdr.cp_length on msg_ptr