EXPERIENCE WITH FORTRAN VERIFIER

A Tool for Documentation and Error Diagnosis of FORTRAN-77 Programs

Reidar Conradi
Division of Computer Science Norwegian Institute of Technology
N-7034 Trondheim-NTH Norway
Phone: +47 7 593444 Email: conradi@wax.runit.unit.uninet nor mta no
Telefax: +47 7 595103 Telex: 55637 NTHAD.N (att.: Databehandling)

Abstract: FORTRAN VERIFIER (FORTVER) is a tool to assist the development, testing and maintenance of large FORTRAN-77 programs. FORTVER will check module interfaces, and produce various documentation and cross-references. It will compute inter-procedural, flow-insensitive sideeffects with complete static and dynamic aliasing in order to diagnose unused, unevaluated, unassigned, or otherwise illegally used variables. An improvement of Banning's algorithm for SDF-parameter aliasing is also introduced.

FORTVER has analyzed 50 production programs with 1.5 M lines of code, including 3 programs of 300 K lines each. FORTVER routinely diagnoses one serious error per 500-1000 lines at a cost of $20 per error, and can analyze 10-100 times bigger programs than the capacity of similar test tools. Results from four analyzed programs are reported and discussed.

FORTVER has recently been expanded with the FV/GE query editor, using data dictionary functions to assist programming-in-the-large. These tools are being integrated into the EPOS programming environment through the EPoS-DB system database. (EPOS means "Expert System for Program and System Development").

Keywords: FORTRAN, Dataflow Analysis, Error Diagnosis, Programming Environment.

Resume: Le VERIFICATEUR de FORTRAN (FORTVER) est un outil d'environnement de programmation pour aider à développer, à tester, à maintenir des programmes en FORTRAN-77 de taille importante. FORTVER vérifiera les interfaces de modules et produira diverses documents et inter-références. Il produira des effets secondaires interprocédures, basé sur l'analyse complète de l'aliasing statique et dynamique, pour diagnostiquer les variables qui n'ont pas été utilisées, évaluées ou autrement employées de manière illégale. Une amélioration de l'algorithme de Banning pour calculer l'aliasing des paramètres de références a été introduite.

FORTVER a analysé 40 programmes en fonctionnement avec un total 1.5 M lignes de code, y compris trois programmes de 300K lignes chacun. Il diagnostique de manière neutre une erreur grave par 500-1000 lignes au prix de 20S par erreur, et il peut analyser de programmes 10-100 fois plus grands que la capacité de semblables outils de test. Les résultats de quatre programmes analysés sont exposés et discutés.

FORTVER a récemment été élargi par l'introduction de FV/GE, un éditeur à questions, qui utilise des fonctions de dictionnaires de données pour aider à la programmation-en-grand. Ces outils sont en train d'être intégrés à l'environnement de programmation EPOS (Système d'Expertise pour la Développement de Logiciel) par le biais d'une base de données de système, EPoS-DB.

Mots clés: FORTRAN, Analyse de Transport de Données, Diagnose des Erreurs, Environnement de Programmation.
1 Introduction

FORTRAN is still a heavily used language for technical applications, and typical production programs consist of 10-500 K lines. These programs often have important economic consequences (strength analysis, reservoir simulation, flight control), and need to be efficient, reliable, correct and robust. They may have lifetimes of 5-20 years, may run on several different computers, and may be developed and maintained by an ever-changing populace of scientists and programmers (perhaps being inadequately trained). Partly as a consequence of this, the documentation is often poor, and this may prevent recruitment of new people to work on an old (but profitable) program product. And a quick glance into real-world FORTRAN-programs tends to induce a desperate feeling among computer scientists.

FORTRAN-77 is an old-fashioned language. It is vulnerable both to trivial interface errors, such as inconsistent parameter- or COMMON lists, and to classical errors. The latter ones are amplified by FORTRAN's awkward layout rules (non-significant blanks, fixed columns) and automatic declaration of unknown, sometimes misspelt names. The canonical example is the '0010[1.10]'-error in Mariner-I /SIGSOFT 85/. Involuntary omission of "variable", DIMENSION- or COMMON-statements may create similar problems, and is often caused by missing inclusion of shared declarations. Further, the porting of FORTRAN-programs can be hampered by incompatible dialects/supersets and by different wordlengths, causing misaligned COMMON-variables or loss of numeric precision. See /Knuth 71/ for empirical data of FORTRAN programs.

Thus, there is a need for language-specific analyzers and preprocessors to compensate for FORTRAN weaknesses, in addition to tools for programming-in-the-large. The latter tools fit into the framework of Programming Environments (PES) or the more general Software Development Environments (SDEs) /Henderson 86/. Few of such PEs are aimed specifically at FORTRAN, with the exception of ToolPack /Osterweil 83/ and Ran /Cooper 86/.

In the following, we shall present a new PE tool, FORTRAN VERIFIER or FORTIVER /Conradi 86/. FORTIVER assists the development, testing and maintenance of large FORTRAN-77 programs. It uses static, interprocedural code analysis to perform error diagnosis and documentation. An improved algorithm for REP-parameter aliases is employed to achieve this. Extensive experience exists from analysis of large production programs. Both FORTIVER and a cooperating FOR/76 query editor are being integrated into the EPOS programming environment /Conradi 87a/, via the EPOS-DB system database /Conradi 87b/.

2 Static flow analysis of FORTRAN-programs

By static we mean "before program execution", as opposed to dynamic or "during execution". Many static PE tools employ dataflow analysis: compilers use it for code optimization and error warnings, verifiers for program verification, test generators to produce test data, and diagnosis tools like FORTIVER to catch anomalous usage of variables.

2.1 Dataflow analysis for anomaly control: using USE, DEF and UNDEF

The following definitions apply to a basic block (B) in a subprogram /Fodick 76/ /McChinck 81/:

Let \( V \) be a variable, and BDEF, BUSE, BUNDEF, etc. sets of variables.
Let BDEF(B, V) reflect modification of \( V \), i.e. \( V \) is assigned to.
Let BUSE(B, V) reflect evaluation of \( V \), i.e. \( V \) is used as a value operand.
Let BUNDEF(B, V) reflect declaration of \( V \) after subprogram entry, or un-declaration after subprogram or loop exit.

Let DEF, USE, and UNDEF be the intraprocedural, flow-defined closures,
and SDEF, SUSE, and SUNDEF be the interprocedural, flow-defined closures.

A technical problem exists for DEF: Can a (partly) UNDEFINED expression be assigned to another variable, and thus DEFINE this? (Our answer is Yes!)

The following anomalous execution sequences can then be defined:

- SUNDEF ...non-accessing code... Unused variable.
- SUNDEF ...non-accessing code... SUSE Uninitialized or LIVE variable.
- SDEF ...non-accessing code... SUNDEF/SDEF Waisted variable value.

This means that flow analysis may catch clerical errors such as:

- I = 0 The letter "0" is mistyped zero, and the 0-variable may be diagnosed as an uninitialized variable.

This exact error has occurred in four of the analyzed programs!

Similarly, the sideeffects of a subprogram call consist of the SDEFined and SUSED variables of this call. The SDEF-information can be used to diagnose modified DO-loop variables, modified function parameters inside expressions (like in A+F(A)), or modified actual parameters being constants or expressions. This sideeffect information can also be channelled back to the user as a sophisticated cross-reference, and may reveal dubious use of variables across more than one subprogram.

In simple cases, some of the above anomalies can be caught without any flow analysis at all, e.g. by using a cross-referencer or by using compiler-directives like IMPLICIT NONE on VAX/VMS. In general, their exposure require full interprocedural flow analysis or even symbolic execution. The issue of imprecision versus tractability is treated in the next three sections.

2.2 May and Must Information

- May or flow-insensitive information reflects execution, that sometimes is invoked, typically within IF- and DO-branches.

- Must or flow-sensitive information reflects execution, that always is carried out, regardless of program branches. It is assumed that all branches are actually, but independently executable.

Ex. Imprecise dataflow analysis for IF-statements in PASCAL.

```pascal
IF B1 THEN X := V ELSE Y := V;
<Does B1 imply B2? E.g. is B2 = B1 and no intervening B1-DEFinition?>
IF B2 THEN WRITE(X) ELSE WRITE(Y);
```

Here the variables (X,Y) are may-USED, and (X,Y) are may-DEFed. Similarly, (V,B1,B2) are must-USED, but none are must-DEFed. Without special knowledge of B1 and B2, being generally undecidable /White B1/, the analysis will characterize both (X,Y) as may-LIVE or possibly uninitialized variables. Only (V,B1) can immediately be characterized as must-LIVE within this program fragment.

2.3 Alias analysis

Aliasing means different names of the same variable (or storage space) in the same context. The previous USE/DEF analysis must therefore be augmented ("widened") with alias analysis.

An alias relation between two variables V1 and V2 is often written as an alias pair. (V1,V2). There are 3 kinds of aliasing in FORTRAN:
- **Static aliases**: Explicit EQUIVALENCE overlap, or implicit overlaying of COMMON blocks in different subprograms. These are easy but tedious to compute, and represent must-information.

- **Dynamic array-index aliases**: A(I) and A(I). Pointer addressing belongs to this category in other languages. Dynamic array aliases are usually treated at a coarse-grained level, so that A(I) and A(J) are regarded as instances of A itself. That is, must-information instead of may-information.

- **Dynamic REF-parameter or callgraph aliases**: Overlapping of the form (Fi,Fj) or (Fi,Gk), where F's are formal REF-parameters and G's are global variables. We assume that REF-parameters are the de-facto (although not standard) parameter-transmission method in FORTRAN. Although such aliases are "dynamic", we will treat them statistically as may-information, i.e., they will be over-estimated.

All alias relations are reflexive and symmetric. The static and callgraph aliases are non-transitive, while the array aliases are assumed to be transitive.

**Ex.** REF-parameter or callgraph aliasing in FORTRAN.

```fortran
CALL P(I,L)
CALL P(G,L)
---
SUBROUTINE P(A,B) (A,B) and (A,G) will become local alias pairs in P, but do not apply simultaneously, given these calls.
```

The complete set of callgraph aliases is given by the closure of the alias relation on the callgraph, as mentioned, such aliases are non-transitive, since local alias pairs may not apply simultaneously (see above example). We will use a modification of Banning's top-down, recursive algorithm /Banning 79/.

**Ex.** Banning's algorithm for non-transitive alias closures.

```fortran
PROCEDURE VISIT(X,Y): (* X,Y represent variables *)
BEGIN
  IF (X,Y) is a new alias pair THEN BEGIN
    Mark (X,Y) as a new alias pair;
    1) FOR all calls to a proc P, where both X,Y are used as Ai,Aj ref-actuals
       DO VISIT(Fi,Fj); (* where Fi,Fj are the corresponding forms. *)
    2) FOR all calls to a proc P, where X is passed as an Ak ref-actual
       and Y is directly or indirectly accessed by P
       DO VISIT(Fk,Y); (* where Fk is the corresponding formal. *)
    3) IF X <> Y THEN (Perform step 2 with X,Y interchanged);
  END: (* (X,Y) is new alias pair *)
END: (* of VISIT-procedure *)
```

**Ex.** For all V DO VISIT(V,V); (* The effect of static aliases left out.*)

The algorithm may imply lengthy searches for interesting calls. Effective cut-offs in steps 1-3 will assume knowledge of "relevant" variable accesses inside P (i.e., transitive SUSP/SEDF closures). For instance, we should only call VISIT(Fk,Y) in step 2, if SDEF(P,Fk) and SDEF(P,Y) or SUSP(Y) is valid in procedure P, or vice versa. All other accesses of Fk and Y will generate irrelevant alias pairs. Given no "clairvoyance" on the usage of global variables or parameters inside P, or its called procedures, we must resort to worst-case assumptions. That is, to propagate all potential alias pairs downwards in the callgraph. This can be very time-consuming, and may drastically increase the number of irrelevant alias pairs. This again "contaminates" otherwise un-aliased variables and reduces the level of precision, see next section.
2.4 Coping with interprocedural USE/DEF analysis

Intraprocedural may-analysis has cost O(N^2 NVars), where N is the number of basic blocks. The corresponding must-analysis has cost O(N^2(D+2) NVars), where D is the number of back-edges ("loops") between basic blocks. D is usually small.

Interprocedural may-analysis of REF-variables and global variables ("sideeffects") behaves as intraprocedural must-analysis with respect to cost, but operates on the global callgraph. The cost is polynomial, see below. However, the corresponding must-analysis has exponential cost in the presence of callgraph aliases, as all possible call paths must be individually considered: O(NCalls*MaxCallDepth).

We have therefore chosen to perform only interprocedural may-analysis. It is carried out in three phases:

A) Compute intraprocedural USE/DEF may-information, with cost O(N NVars).
B) Propagate callgraph aliases top-down, with only static alias widening. At worst cost O(NCalls*NCalls*MaxRefParms NVars).
C) Propagate SUSE/DEF sideeffects bottom-up, with full static and dynamic alias widening. At worst cost O(NCalls*NCallLoops*2 NVars^2).
   but the FORTRAN cost is O(NCalls* (NGlobals+MaxLocals)*2 * AliasFraction).

There are many obvious cut-offs in phases B-C, such as not to treat global-variable sideeffects more than once for calls between two subprograms, or not to treat "flow-identical" calls more than once.

To achieve the clairvoyance cut-offs ("heuristics") in phase B, we propose an additional phase between A and B, calculating bottom-up sideeffects with only static alias widening. The effect of this yo-yo technique can be dramatic, if global variables are frequently used as actual parameters or if massive COMMON-equivalents prevail. Gains of a factor 10 in total execution time for phases B-C have been observed! This yo-yo algorithm contrasts the model by /Cooper 84/, where the SUSE/DEF sideeffects are divided into three groups: those for REF-parameters (including aliases) with cost O(NCalls*NCalls*MaxRefParms), those for global variables with cost O(NCalls*NCallLoops+2 * NGlobals), and an unspecified combination category O(Epsilon?).

2.5 Previous work

For program diagnosis and testing, there has been an increasing interest in formalized techniques such as dataflow analysis /Fosdick 76/ /Miller 81/.

The University of Colorado in Boulder has initiated several flow-based diagnosis tools: DAVE for FORTRAN-66 /Osterweil 76/, the HAL/S environment /Taylor 83/, and OMEGA for C /Wilson 85/. The tools are prototypes, and cannot properly handle aliases or big programs. They have all been plagued by cascades of "false alarm" warnings because of imprecise may-information. Taylor experienced 1700 "may-occur" and 20 "must-occur" errors in a 760-lines HAL/S-program. A ToolPack successor system for FORTRAN-77 is announced, but no hard data on its implementation exist.

The PROBE system for PL/I is perhaps the most ambitious flow-based tool, but it can only analyze programs with a few thousand lines because of resource constraints (7 MB for 500 PL/I statements) /Sarraga 84/.

The AIDA system for PASCAL has tried to apply dynamic data flow analysis through an instrumented compiler, but no empirical data is reported /Chan 86/.

The MAT system for FORTRAN-77 checks module interfaces and callgraph reachability, but performs no interprocedural flow analysis /Barnes 84/. Much "clutter" and interface errors are reported.
LINT /Johnson 79/ and CFLOW are UNIX-tools to check interfaces and callgraphs of C-
programs.

The CDL2 compiler employs interprocedural must-analysis of procedure calls
/Feuerhahn 78/. Programmers are often surprised by the "deepness" of the errors
reported.

Other compilers have used, or plan to use, interprocedural may-analysis for code
optimization, see surveys in /Muchnick 81/ and /Cooper 86/.

2.6 Possible improvements

Most of the above tools contain useful documentation facilities (surveys and cross-
references) and perform checks on module interfaces (parameter lists and COMMON-
blocks). The flow-sensitive tools suffer from exponential time consumption and
cascades of false alarms. They usually don't treat aliasing or formal procedures
either. They also generate rather bulky output texts (on files or paper), and are
seldom integrated in PEs.

3 FORTVER

FORTVER tries to repair some of these deficiencies. In addition to documentation
production and interface checking, FORTVER will apply fast, flow-insensitive,
interprocedural analysis with full alias widening to perform error diagnosis of
very large programs. Although, many flow-related errors escape notice, this is
probably the only cost-effective solution for big programs. As mentioned, an
experimental FV/QE query editor is running against the EPOS-DB system database,
seeing as a data dictionary.

3.1 FORTVER overview

FORTVER consists of a routine verifier (ROUTVER), a system verifier (SYSTVER), and
a query editor (FV/QE).

FORTVER structure:

[Diagram showing the flow of data between various components, including ROUTVER, SYSTVER, FV/QE editor, EPOS-DB, and user interaction.]
3.2 ROUTVER functions

- ROUTVER reads a list of source files, and the source is validated against FORTRAN-77 (ANSI 3.9-1978) or FORTRAN-4 (COLIN-DAY). Extensions for common target machines exist, and the size of different datatypes are user-controllable.

- Cross-references within a program unit are produced. All unused, unassigned, or "not-yet-assigned" local variables are flagged. (No local flow analysis is yet performed, see below in Sec. 6.1.) An annotated source file, and 1-2 error files may be generated.

- An intermediate, summary (text-)file is prepared for SYSTVER. Optionally, the USE/QEF summaries and cross-references are put into the EPOS-DB database.

3.3 SYSTVER functions

- A collection of intermediate files is read, and a trimmed callgraph established by eliminating unreachable subprograms and potentially recursive calls. Dummy calls are added from the main program to BLOCKDATA modules. Calls on formal procedures are substituted by calls to the UNION of possible actual procedure parameters. The callgraph is topologically sorted, and the maximal call depth calculated.

- Global lists of subprograms and COMMON-blocks, and associated cross-references are produced. The callgraph can be displayed, using textual indentation. Special lists of unused, multiply-declared, missing, or possibly recursive subprograms are prepared. Various program statistics are also produced.

- Checks of subprogram calls (number/type/kind of parameters, function type) and COMMON-blocks (size, layout) are performed.

- Relevant aliases and may-sideeffects of subprogram calls are computed. Attempts to assign into actual parameters, being constants or expressions, are flagged. Ex. CALL $1($299), where 999 is changed by $1. It is also checked if a callgraph-aliased variable can be changed through another one, as this represents a FORTRAN-77 violation. Unused, unassigned, or unevaluated variables across subprogram boundaries are likewise flagged.

- The 25 different error messages can be grouped into 5 "severity" classes, which can be independently routed to a maximum of 5 log files.

- SYSTVER is currently not interfaced to EPOS-DB.

3.4 A language-oriented editor with query functions: FV/QE

FORTVER is a batch-oriented code analyzer, strong on programming-in-the-large. For programming-in-the-small, various interactive editors have been developed. We realized that FORTVER-generated summaries could easily and beneficially be used for interactive program development through a prototype editor, the FV/QE.

We have therefore augmented a multi-window text editor with query functions against a system database. The database is filled in by ROUTVER. Such an editor is inspired by Interlisp's MasterScope (Teitelman 81), the VAX ADA environment (Mitchell 85), and the Rn environment. It consists of 15 K PASCAL lines. No textual query language has been developed.

Examples of commands to FV/QE:
- Show the declaration of a given name!
- Give me a set-up to call subroutine S2!
- Show all calls of S1, each in turn, in a separate window!
- What is parm.no. 2 of CALL PUTS(---) in PP02?
- Where is variable MAXSUM modified, directly and indirectly?
- Which subprograms will be affected by changing S1 or BL?
- Show the call tree of S2!

3.5 The EPOS environment and the EPOS-DB system database

One of the goals in EPOS is to get maximum synergy out of existing tools, by integration through the EPOS-DB database. Typical EPOS tools will be compilers, code analyzers like FORTVER, special editors, debuggers, linkers, configuration tools, project management tools, and knowledge-based assistants /Rich 86/.

PE environments like SAGA /Campbell 84/ have tried the same, but uses no flow-information or a common database. The An environment for FORTRAN is oriented against editing and compilation, and neglect important programming development issues. The ToolPack environment is more ambitious, but not yet implemented.

Databases for PE face a delicate compromise between functionality and performance, or granularity versus efficiency. This explains the recent interest for object-oriented databases.

EPOS-DB contains medium-grained, derived, language-independent, summary information of programs /Conradi 87b/. Typical data are name declarations, xref information, an annotated callgraph, aliases and sideeffects. Some general project information is also included.

Source text is still regarded as the primary program representation. The (derived) database must be updated frequently enough - a few times per day - to ensure reasonable consistency. Most module interfaces stabilize early in a programming project, so this will probably be OK.

EPOS-DB is of type Entity-Relation-Attributed, ERA. The current version has 21 entity types, 41 relationship types, and 132 attributes. It supports a two-level class hierarchy, where all objects have a common head and a special variant tail. The object head for entities allows general tree structuring of data. All database objects possess a unique and immutable 8-byte key, used for universal addressing. No versioning or concurrent updating is allowed.

EPOS-DB is implemented on top of ND-500 SIBAS, a network-based DBMS supplied by Norsk Data /NorskData 84/. This is a pragmatic choice, since we cannot spend 20-30 manyears on building our own DBMS.

Database access is semi-automatic, and go via a generic, object-oriented interface, EPOS-DB-SHELL. The interface allows i/o of small or large chunks of heap-allocated program data. Care has been taken to minimize write-back during updating, by incremental change control for each object or group (set) of such. The interface is implemented by 20 K lines of PASCAL code.

3.6 FORTVER implementation

FORTVER is configured to analyze programs over 1 M lines. Much emphasis has been put on fast algorithms (cf. propagation cut-offs in Secs. 2.3-2.4) and on compact data representations:
- Only "InterProc" or IP variables are given to FORTVER, excluding purely local or NIZ (Non-IP) variables. This saves a factor 5.
- Only reachable subprograms are treated by FORTVER, excluding 20-60% garbage.
- Only relevant sideeffects and aliases are stored, saving a factor 100.
FORTIVER contains 40 K lines of straight-forward PASCAL code. It has been running on VAX/VMS since autumn 1983 and on ND-500/SINTRAN since spring 1985, both 32-bit machines. Implementations on SUN/UNIX and IBM/VM will be ready in autumn 1987.

On a VAX-780, FORTIVER analyzes about 30 lines/cpu-sec, being one third of normal compilation speed. This speed figure includes full debug support. The analysis time is roughly linear to the number of subprograms, but extensive aliasing can cause a slight increase. A program system of 3000 modules, or about 300 K lines, will thus require 30 M of virtual storage and 2-3 cpu-hours on a VAX-780 (often the double in elapsed time). This appears to be acceptable.

On ND-500 with EP05-08 extensions, the ROUTIVER speed is 8-10 lines/cpu-sec, and the database occupies 3 times more space than the source text. The FV/QE operates on a conventional text terminal, and a typical database query takes 2-5 seconds.

4 Four case examples

We shall present the findings from four programs, analyzed on a VAX-780:
- A graph plotting program: GRAPH with GPFS-F library, 56 K lines.
- A real-time program for satellite navigation: SARSAT, 30 K lines.
- A strength analysis program (preliminary): SESAM80, 280 K lines.
- A relational database program: TECHRA, 61 K lines.

The ROUTIVER errors refer to NIP-variables, and the SYSTIVER errors to IP-variables. Calls to intrinsic functions (like SIN and ABS) are not counted. A star (*) means unavailable because of old FORTIVER version.

<table>
<thead>
<tr>
<th>GRAPH</th>
<th>SARSAT</th>
<th>SESAM80</th>
<th>TECHRA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total source, exclusive INCLUDES (lines): 56209 29673 280134 60660</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- whereof comments and blanks (lines): 30828 13834 9 31250</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Net source size (lines): 25381 15839 200000 29402</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**ROUTIVER-counts for NIP-variables:**

| Time used (cpu-sec): 2072 1268 10000 2112 |
| Declared, but unused vars (garbage): 133 305 >500 132 |
| Unevaluated, but assigned vars (waisted): 85 305 >300 17 |
| Evaluated, but not-yet-asg'ed vars (uninit): 27 110 44 15 |
| - whereof never assigned vars: 6 94 >30 6 |

**SYSTIVER-counts for IP-entities:**

| Time used (cpu-sec): 1400 302 1460 933 |
| Total relevant (reachable) program units: 887 519 1174 681 |
| - whereof subroutines: 728 410 1032 430 |
| - whereof functions: 663 347 961 313 |
| - whereof entry- subprograms: 49 8 51 124 |
| - whereof undecl. subprograms (missing): 9 1 2 0 |
| Excluded, multiply-declared subprograms: 6 53 17 0 |
| Called-once subprograms: 400 215 293 174 |
| Bottom-level (no outg.calls) subprograms: 95 175 194 102 |
| Relevant calls: 4152 1814 13614 2470 |
| - whereof potentially recursive ones: 11 - - 6 |
| Maximal call depth: 53 9 19 31 |
relevant COMMON-blocks:
- being referenced how many times:

<table>
<thead>
<tr>
<th>Relevant IP-variables:</th>
<th>35</th>
<th>45</th>
<th>45</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>1337</td>
<td>1800</td>
<td>1870</td>
<td>1835</td>
<td></td>
</tr>
<tr>
<td>Relevant IP-variables:</td>
<td>4506</td>
<td>3666</td>
<td>20768</td>
<td>2675</td>
</tr>
<tr>
<td>whereof formal parameters:</td>
<td>2017</td>
<td>1568</td>
<td>6951</td>
<td>981</td>
</tr>
<tr>
<td>whereof local IP-variables:</td>
<td>1500</td>
<td>1620</td>
<td>13309</td>
<td>1046</td>
</tr>
<tr>
<td>whereof COMMON-variables:</td>
<td>909</td>
<td>478</td>
<td>478</td>
<td>638</td>
</tr>
<tr>
<td>Max. formals in a subprogram:</td>
<td>16</td>
<td>34</td>
<td>36</td>
<td>12</td>
</tr>
<tr>
<td>Max. formals + IP-locals in a subprogram:</td>
<td>42</td>
<td>63</td>
<td>86</td>
<td>21</td>
</tr>
<tr>
<td>Static alias pairs:</td>
<td>391</td>
<td>617</td>
<td>3744</td>
<td></td>
</tr>
<tr>
<td>Dynamic alias pairs, only relevant ones:</td>
<td>302</td>
<td>29</td>
<td>1263</td>
<td>38</td>
</tr>
<tr>
<td>Dynamic aliased formals + globals:</td>
<td>185+95</td>
<td>27+6</td>
<td>784+11</td>
<td>51+11</td>
</tr>
<tr>
<td>Dynamic alias-modify error of variables:</td>
<td>82</td>
<td>14</td>
<td>&gt;300</td>
<td>20</td>
</tr>
<tr>
<td>Type errors in parm/function-result:</td>
<td>47</td>
<td>9</td>
<td>15</td>
<td>280</td>
</tr>
<tr>
<td>Wrong-number-of-parm errors:</td>
<td>9</td>
<td>50</td>
<td>-</td>
<td>10</td>
</tr>
<tr>
<td>Other parm.obscur.(Var-to-Array,Dim-err):</td>
<td>245</td>
<td>69</td>
<td>&gt;100</td>
<td>170</td>
</tr>
<tr>
<td>Expression-modify error in proc calls:</td>
<td>409</td>
<td>14</td>
<td>&gt;200</td>
<td>7</td>
</tr>
<tr>
<td>Not-indirectly-used actual params:</td>
<td>45</td>
<td>306</td>
<td>&gt;50</td>
<td>7</td>
</tr>
<tr>
<td>Unused (garbage) IP-variables:</td>
<td>42</td>
<td>99</td>
<td>&gt;100</td>
<td>11</td>
</tr>
<tr>
<td>Unevaluated (waisted) IP-variables:</td>
<td>103</td>
<td>173</td>
<td>&gt;100</td>
<td>2</td>
</tr>
<tr>
<td>Unassigned (uninitialized) IP-variables:</td>
<td>20</td>
<td>88</td>
<td>&gt;10</td>
<td>37</td>
</tr>
</tbody>
</table>

Comments.

Due to undeclared subprograms because of unavailable user libraries, the SYSTVER figures for "under"-used variables are somewhat inflated. Similar comments apply to SESAM, because of missing static alias analysis.

Anyhow, there is a staggering amount of uninitialized HIP-variables, although the current LIVE analysis is incomplete. There are also hundreds of unassigned, unevaluated, or plain unused IP-variables, and many not-indirectly-used actual parameters. Most of the unused variables are dummy formal parameters.

Both GPOS-F and TECHRA contain potential, but never executed recursive calls. The big number of static aliases in TECHRA is caused by massive EQUIVALENCEing of variables in a few COMMON-blocks.

There are several erroneous attempts to modify constants/expressions in procedure calls, so-called expression-modify errors. There are also many alias-modify errors for formal parameters (mostly arrays), apparently innocent.

More serious is the substantial number of blatant TYPE ERRORS in procedure calls. Over half of these are intentional, but many INTEGER*2-to-INTEGER*4 and INTEGER-to-REAL errors have been spotted. The number of wrong-parameter errors is also disturbing.

5 FORTIVER discussion

FORTIVER is probably the only interprocedural analysis tool to have analyzed really big programs, and which has a user community outside universities. We have experienced both theoretical and practical limitations. Since many previous tools have had problems in "getting on the air", we think that our comments have general interest.
5.1 ROUTVER discussion

The FORTRAN-77 compatibility warnings from ROUTVER are considered a nuisance. They should (and can) be turned off either selectively, or by a single command to specify the target machine. On the other hand, ROUTVER must accept full machine-specific extensions, particularly for I/O and library routines.

No complete, local flow analysis is performed, though a textual USE before DEF is taken as a not-yet-assigned variable. Although this analysis "leaks in both ends", surprisingly many errors are spotted.

Some simple program transformations should be considered: pretty-printing, generation of comment boxes, label renumbering, etc...

/Beans 84/ suggests that all names in COMMON-blocks should be unique in a program system, and that (NIP-)variables are checked to catch missing COMMON-declarations.

ROUTVER output to SYSTVER might have been generated from an instrumented FORTRAN compiler.

5.2 SYSTVER discussion

Precise diagnosis of type errors is hampered by liberal FORTRAN conventions for procedure calls and COMMON-block layouts. For instance, an array may be passed as an actual parameter to a normal variable-parameter, which again may be given to an array-parameter. These two "kind"-errors annihilate each other, assuming by-REF transmission. Likewise, the amount of unsafe but innocent bit fiddling is considerable, such as passing a small array to an equal-sized simple variable. There may also be false alarms on alias-modify or expression-modify errors for procedure parameters, if the modifying code branch is not executed, or if imprecise aliases flourish.

The COMMON-block layout checks have proved useful between machines of different wordlengths.

It is sometimes difficult for a programmer to trace the real cause of a reported error. Some SYSTVER assistance in displaying relevant call chains would have been helpful.

Undeclared library routines, possibly in another language, must be simulated by "IP-equivalent" FORTRAN-skelettes.

Many of the SYSTVER findings could have been utilized by an optimizing compiler: 30-50% of the procedures are called once, and 15-30% are bottom-level routines. The number of procedures without sideeffects and calls without locally relevant sideeffects have not been counted. The number of aliased REF-parameters appears to be small: 2-3%. /Banning 79/ reports similar numbers for PASCAL.

It is hard to evaluate the cost-benefit of interprocedural analysis over a single level ("caller-callee") analysis. The full IP-analysis is 30% slower, but requires 5 times more space. Some IP-findings (like most alias-modify errors) must be classified as noise, but other IP-usage checks have discovered many genuine errors. This information may also be given to a query editor.

Function calls with malignant sideeffects or unauthorized modifications of DO-variables cannot be diagnosed, since we don't know the calling context precisely - a ROUTVER weakness.
5.3 General FORTVER discussion

- 50 FORTRAN-programs with over 1.5 M lines have been analyzed, including three programs of 300 K lines each. All analyzed production programs have contained serious errors, on the average one per 500-1000 lines. A FORTVER analysis of a 300 K lines program will cost $200 in machine resources, plus about $1000 to prepare the run. In our case, about 60 grave errors were caught, i.e. about $20 per error! Most errors are found in rarely executed code parts.

Even if no specific errors are found, a FORTVER analysis can be used to certify an existing program. Really, the errors revealed by FORTVER represent the "tip of the iceberg":

- Another advantage is documentation: The size or extent of a submitted program system is seldom precisely known by its implementors. A size misjudgement by a factor 2 is not uncommon. All analyzed systems have contained clutter: unused, doubly-declared, or missing modules, as well as "under"-used variables. Even COMMON-blocks and subprograms with identical names have occurred. This is legal but not advisable!

- The basic drawback of FORTVER, particularly of SYSTVER, is its batch-orientation. The sequential inhale and exhale of data takes 60-70% of the cpu-time, and the output reports are big, technical, and often unmanageable. We need more selective means to suppress, control and display the output. Reports about the same entity are scattered out, and many errors are repeatedly reported, e.g. for parameter mismatches or COMMON-layout differences. We may consider filter commands to suppress already-seen or OK'ed errors:
  
  `IGNORE (TYPE-ERROR) FOR X AS-PARM-N0 2 IN CALLS FROM P TO Q`.

Similar predicates could be used as editor queries to EPOS-D8.

The best solution is probably to store the old analysis data in the database, and perform incremental, interprocedural analysis upon changes /Torrzon 85/.

This also brings up the issue of different program versions and configurations.

- FORTVER techniques should be extended towards other programming languages, such as COBOL, C, PASCAL, CHILL, and ADA. Several algorithms in SYSTVER must be generalized to accept non-FORTRAN language constructs.

6 Conclusion

FORTVER is now a commercial, cost-effective tool for documentation and error diagnosis. It has been used by a dozen different customers since 1984, and has analyzed programs up to 300 K source lines, which is one or two orders of magnitude greater than the capacity of similar test tools. Further work on FV/QE and on tool integration in EPOS is clearly needed.

The biggest successes:
Large programs analyzed at reasonable cost, diagnosis of important type- and usage errors, valuable program documentation generated, promising editor interface, interesting PE perspectives.

The biggest failures:
Poor user documentation, big and cryptic output, insufficient user customization, some redundant may-errors, costly and cumbersome batch set-ups, no analysis of subsystems or reuse of previous results.

Acknowledgements

FORTVER has been supported by The Royal Norwegian Council for Scientific and Technical Research (NTNF) under grant ED2021.13615 in 1984-86. EPOS is being supported by a similar grant IT0224.18457 in 1986-87. I wish to thank all my coworkers on these projects for their whole-hearted support.
References


