First Steps Towards a Tool for Legacy Systems

Mariano Méndez, Fernando G. Tinetti*

III-LIDI, Fac. de Informática, Universidad Nacional de La Plata,
50 y 120, La Plata, Argentina

Abstract. After its first release, software must face change, because change is a part of its true essence. Often, programmers have to deal with software built by others. When an improvement, change or update must be introduced, programmers need first to understand the existing software. In order to achieve that understanding, development tools are crucial. This paper describes some key features required by a tool to help programmers to understand and handle existing software. We propose to put all these features together so as to add them to an IDE (Integrated Development Environment). This paper purports to characterize a set of steps in order to help in and/or manage that transformation process.

Key words: Software Transformation, Legacy Systems, Fortran Legacy Systems

1 Introduction

Software is constantly exposed to the pressure of change because change is part of its essence [6]. Thus, how these changes are introduced in an already operational program is still a challenge nowadays. If we try to imagine how a change is introduced in a system built following modern techniques of Software Engineering, most probably the conclusion will be that it is not an easy task. Every change will impact (in)directly on the software specifications and in a much more profound way on the software behavior. The impact entails a long list of elements such as: requirements specification, requirements validation, system documentation, source code, tests, maintenance tasks, and others major artifacts resulting from the development process. The nature of the changes to be applied on programs is very complex, even on those systems built thoughtfully. There are authors that believe that software development is program transformation, they assert that "programmers view software development as a process to convert one program version to next" [14]. The same tasks conducted in a Legacy System can become an arduous challenge. Most of the legacy software lacks system documentation, requirement specification, testing, and so forth. The process to update or improve this kind of software is still a challenge. Moreover, the preliminary process of understanding the system is a feat in itself . We focus our research on scientific legacy software, written in Fortran, but we expect this research can

* Comisión de Investigaciones Científicas de la Prov. de Bs. As.

be extended to any kind of software. These days, applications written in Fortran (the most long lived programming language with a particular evolutionary process [4,17,16]), are available on the web as scientific software that can be downloaded and used by anyone.

We have gathered software coming from a set of institutions such as NASA (USA National Aeronautics and Space Administration), CERN (Conseil Européen pour la Recherche Nucléaire, or European Council for Nuclear Research), and USA US Geological Survey Water Resources, just to analyze some Fortran source code examples. Modern programming techniques does not seem to be used in the development of the collected software (downloadable from the corresponding web sites).

Why is it so important to improve scientific software? Most of the scientific software developed in Fortran and used in production environments is closely related to natural events not completely understood by human beings to this day. There are multiple models of earthquakes, tsunamis, quantum physic phenomena, biological population dynamics, weather research and forecasting, and so forth. Thus, in order to help scientists in this process, quality software tools become necessary. In this work, we will focus on scientific legacy software written in Fortran and the key features for a tool which is meant to upgrade, improve and/or apply changes.

2 Legacy Source Code

In this section we want to analyze some features found in scientific Fortran programs available on internet. Initially, a list of problems or characteristics that should be corrected or improved in Fortran legacy software should be defined. One of the first interesting facts in the collected code, is that code identified as Fortran 90 (in .f90 source code files) still has some old language features. Even when Fortran 90 is fully compatible with FORTRAN 77 (Fortran 90 can be seen as a superset of FORTRAN 77) some FORTRAN 77 superseded features are extensively used, such as fixed format source code, old style Do Loop (no End Do statement used) and old logic operators (i.e.: .eq., .neq.) [16,17,1,2].

From the view point of coding style, the use of GO TO statement has been avoided since several decades [8] but nowadays we still find code written using the GO TO statement. The use of this statement was proved to be a source of code complexity increase, error prone source code, unintelligibility, unreadability and so on [9]. There are other non desirable features that come to light such as a) Simulate a WHILE statement with GO TO statement, e.g. Fig. 1; b) Spaghetti code [9], e.g. Fig. 2; c) More than one subprogram entry point, e.g. Fig. 3; d) Usage of magic numbers [10]; e) Usage of non self-descriptive variables [10]. All of these characteristics sometimes turn the source code unreadable and incomprehensible, transforming the updating or improving process into a highly time consuming and sometimes unsuccessfully task.

```
    . . . .
20     jlo = jhi - inc
       IF(jlo < 1) THEN
           jlo=0
       ELSE IF(x < xx(jlo) .EQV. ascnd) THEN
               jhi = jlo
               inc = inc + inc
               GO TO 20
            END IF
       END IF
  . . . .
```

Fig. 1. Extract from Hst3d (U.S. Geological Survey) [15]

```
 . . . .
55   continue
     if (i1.lt.1) goto 999
     if (snam.ne.stname(i1)) goto 999
     if ((scom(1:2).eq.scompt(i1)(1:2)).and. snet.eq.snetwk(i1)) goto 900
     if ((scom2(1:2).eq.scompt(i1)(1:2)).and. snet.eq.snetwk(i1)) goto 900
     if ((scom(1:2).eq.'XX').and. (snet.eq.'XX')) goto 900
     i1=i1-1
     goto 55
   . . . .
```

Fig. 2. Extract from Hash v1.2 (U.S. Geological Survey) [13]

```
C      INITIALIZE ET2 FOR 'STATE' AND SET UP COMPONENT COUNT
C
     ET2(1)=ET
     ET2(2)=0.D0
     GO TO 11
C     ENTRY POINT 'DPLEPH' FOR DOUBLY-DIMENSIONED TIME ARGUMENT
C          (SEE THE DISCUSSION IN THE SUBROUTINE STATE)
     ENTRY DPLEPH(ET2Z,NTARG,NCENT,RRD)
     ET2(1)=ET2Z(1)
     ET2(2)=ET2Z(2)
 11  IF(FIRST) CALL STATE(zips,list,pv,pnut)
     FIRST=.FALSE.
```

Fig. 3. Extract from JPL Planetary and Lunar Ephemerides (NASA)

3 Handling Legacy Programs

In the software improvement or updating process we have to differentiate two stages. The first one is the understanding stage in which a clear comprehension is needed in order to know how and where a change should be applied. By the end of this stage, the programmer or the team should have acquired or

elicited knowledge from the program. The second stage is closely connected to the process of how the changes are introduced in the program. We propose to follow five steps in this process:

1. Identify and save (e.g. in a software version manager) the current legacy software application/program, which will be taken as the reference. Every change will be accepted/rejected according to its relationship to the reference program.
2. Select and apply a specific change/update to be applied to the reference program. A new program version should be produced from a well known and documented change/update the the current software version. Code transformation should guarantee the preservation of the external software behavior [10].
3. Check/verify the new program version by comparison with the previous one. Define and apply some software testing comparison criterion/criteria in order to accept or reject the new program version. This may/should include a set of test cases if necessary.
4. Accept/reject the change according to the previous comparison. An accepted program version will be the candidate as the current version for the next change. A rejected program version would be:
   - Discarded in order to avoid investing more time/effort in a possible useless change.
   - Reviewed in order to find out the problem/s and possible solution/s.

   The choice can be according to some cost evaluation, at this point there is some work already made, so the experience can be useful for the corresponding cost evaluation.
5. Document the accepted/rejected change. In case of an accepted change, documentation should include at least a general description of the change plus several (if not all) specific/actual changes. Specific changes are highly prone to be produced automatically (e.g. by a software version manager).

And the complete legacy software update process can be described as an iteration on these steps, each iteration for a different specific change, as shown in Fig. 4. This incremental succession of changes has several points in common with a task referred to as *reengineering step* in [12]. Moreover, the complete process shown in Fig. 4 should be integrated in a single tool or environment (IDE), which eventually uses other specific tool/s which can be identified for each step, such as

- Understanding and/or documentation tools.
- Software versioning and revision control system tools must be used to keep changes under control.
- Restructuring infrastructure and/or specific software/tools. The concept of code restructuring has existed for many years now, and some transformation tools have been built to apply transformation rules on a complete program in batch mode. An example of this kind of infrastructure is the DMS tool, which allows for re-engineering and migration of programs in many different programming languages [5].

– Testing infrastructure and/or specific tools.

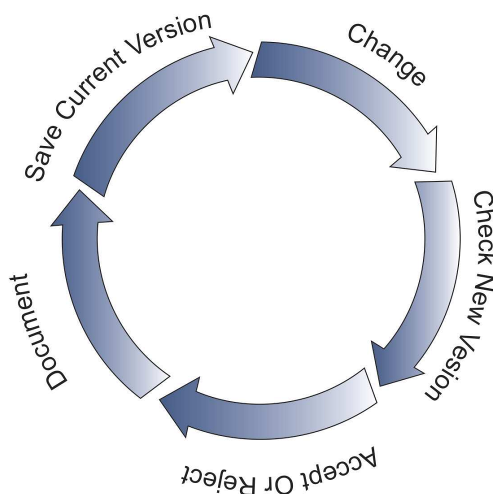

Fig. 4. Software Update Process

In the case of Fortran, the vast amount of existent lines of Fortran code and the investment made on them has encouraged the development of some tools to upgrade legacy Fortran code [20,21,19]. However, applying some transformation rules in batch mode may help updating the code by replacing outdated constructs, but that does not necessarily imply that a developer will gain a better understanding of the structure of code, nor a programmer will be able to clean, modularize, or remove duplicated code. Legacy code will still be (or have several characteristics of) legacy even if it is written in Java but with poor development practices.

4 The Understanding Stage

In this stage, information about the program should be gathered in order to obtain an abstract construction of "what" the program does and "how" it does it. In order to obtain this information a set of tools is used, and usually, these tools are completely disaggregated one from the other, they are not integrated to one another in the same environment on which programmers work such as an IDE (Integrated Development Environment). Some changes/updates can be applied without understanding of what the program does. On the other hand, there are some changes like transforming serial into parallel computing, implementing new requirements, correcting some bugs, and so forth, which need a more comprehensive understanding of the program/software.

When there is no further information about the system other than the source code, two different types of knowledge needed to understand the system can be identified/extracted from the source code. The first one is a language independent knowledge, which is required to understand the program as a whole. Under this characterisation it is suggested to obtain information such as:

- Static call graph.
- Dynamic call graph.
- A runtime profile (or an average of several runs).
- Input and output files/data.

The second type of knowledge which can be extracted from a program is closely related with its internal structure, such as source code dependencies and characteristics (standards, language obsolete features). In the case of Fortran specific source code, relevant features to be updated/changed, include a) Common blocks dependencies, b) Parameter dependencies, c) Module dependencies, d) Old style/standard code: fixed source form, GO TOs, arithmetic IF, etc.

Even when features above mentioned seem to stem from common sense, there exist very few tools that put all of them together into an integrated development environment, let alone Fortran community, in which IDEs are not broadly used. Several criteria or points of view could be used for ranking those features identified as candidate for updating the software

- Risk: (at least) wrong ways of GO TO usage could lead to spaghetti software, which in turn is usually highly prone to error, so replacing GO TOs by better structured code would reduce risks.
- Optimization: DO loops are commonly used for computationally intensive sections of code, so identifying and updating DO loops would produce better source code for optimization and, eventually, for parallelization.
- Frequency: a single outdated or obsolete feature (e.g. arithmetic IF) used too frequently often leads to unreadable code, so replacing such code enhances the legacy code quality (e.g. readability).

5 The Transformation Stage

Once the understanding stage has been initially approached (full legacy software comprehension can be hardly completed in general), the process in which changes are applied starts. This stage can be manually performed, but at least it would be tedious and error prone. Thus, these changes should be done in an automated way. There are tools such as NAGWare tools [20], SPAG [21], PlusFOR [21], Flint [19] most of then are command-line or stand-alone tools. In order to make changes easy to apply, a tool should provide a good transformation infrastructure, including handling the software as a complete project. Optional features include suggestions to the programmer, source code difference highlighting for each change, selection of code portions to upgrade (as opposed

to the whole project), etc. We consider restructuring [3,7] as the candidate technique for those changes, which are specific to each legacy application and we have described some of those changes for Fortran legacy code.

At this point, the understanding stage and the transformation stage were covered/described separately, but both of them are not integrated yet. To make the whole process agile and efficient, both stages should be performed with/in the same tool or environment. A good set of source code analysis tools can be found on internet, such as ROSE [18] and, also, open source IDEs such as Eclipse [22] are available. AST (Abstract Syntax Tree) representation of source code are considered extremely useful in this stage, since it allows an immediate identification of each program component/structure and, thus, allows more focused analysis and implementation tasks of each individual change.

6 A Tool For Legacy Systems

Some of the tools mentioned in Section 3 were integrated in order to show how helpful they can be when integrated directly into an IDE. An Eclipse plug-in was selected to be assembled with some of these tools: Photran, is defined as an IDE and refactoring tool for Fortran based on the Eclipse CDT. Photran refers to restructuring as *refactoring*, and several of them have been implemented for this tool, as shown in [16]. In the next subsections we show two specific tools integrated to the IDE.

6.1 Static Call Tree

The static call tree of a Fortran Program is shown as a directed graph, where each node represents a function or a subroutine and the edges show the caller-callee relationship between two nodes. This view was developed using Zest, a set of visualization components built for Eclipse. The main feature of Zest is that it makes graph programming easy [23]. The basic purpose of this view is to help programmers to understand easily the relationships between program components and to obtain a clearer picture of how complex the connections among these program components are (see Fig. 5).

The static call tree implementation has two major components. The first one is based on Photran VPG (Virtual Program Graph) and it is used in order to obtain the source code representation as an AST (Abstract Syntax Tree). A Visitor pattern [11] is used to build the Static Call Tree by visiting a set of nodes such as functions, subroutines, main/program, and call statements. The second component is used to show the static call tree once it was built from the AST. In order to achieve it, the JFace infrastructure of zest was used to allow this implementation (see Fig. 6).

6.2 Common Block Dependencies

Common blocks (for global data definitions) are widely used in legacy Fortran programs. It is also commonly found in practice one Fortran file per COMMON

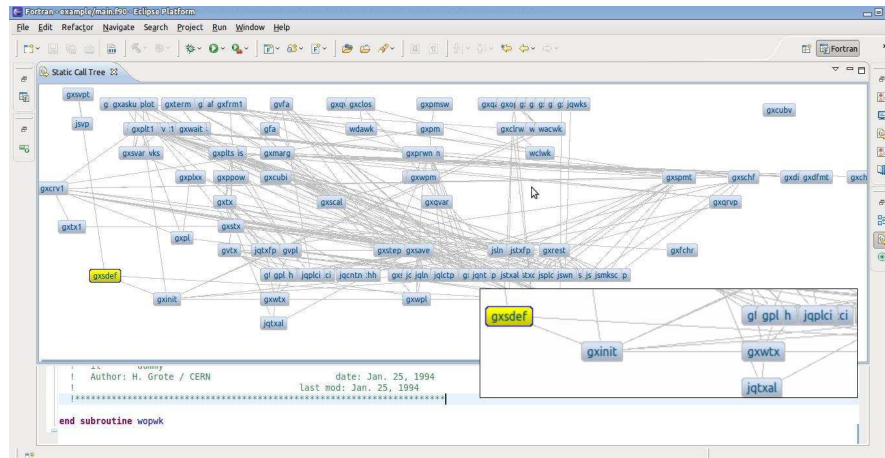

Fig. 5. Static Call Tree- Real Life Example

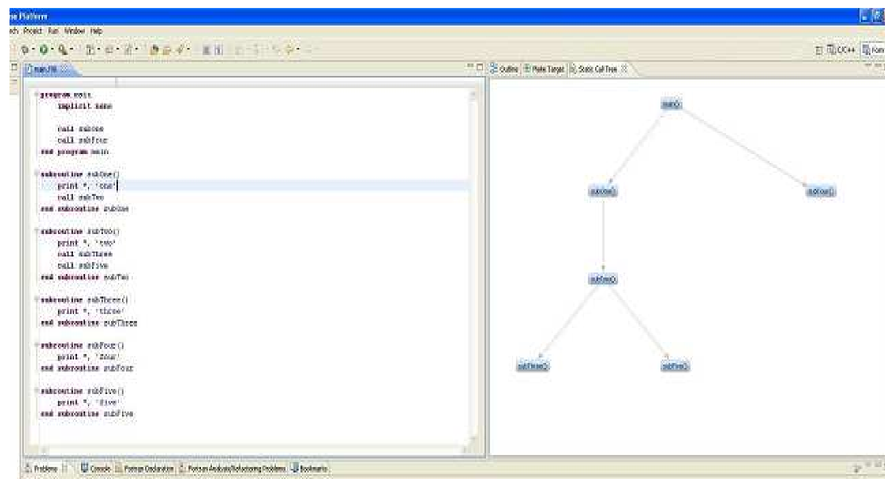

Fig. 6. Static Call Tree

BLOCK defined in the program. This practice can define a very intricate set of dependencies among program files, making the program hard to understand. Furthermore, it is worth noting that common blocks are almost directly related to parallelization tasks where shared/distributed memory has to be explicitly defined and used. Thus, every information about global access data is very useful not only for program legibility but also for further code transformation for parallel computing on shared as well as distributed memory parallel hardware. In order to obtain a global perspective of how they are related, a new feature to the IDE was added. This feature shows COMMON BLOCK dependencies among the

project files, as an HTML. Dependencies are shown as follows in Fig. 7. Dependencies among the files are obtained using some VPG features. These features are language-dependent due to the fact that COMMON BLOCKS are specifically define in the Fortran language.

```
MAIN.F90                        FILE1.F90
   COMMON_BLOCK_1                  COMMON_BLOCK_1
   COMMON_BLOCK_2                  COMMON_BLOCK_2
   COMMON_BLOCK_4                  COMMON_BLOCK_3
                                   COMMON_BLOCK_4
FILE2.F90                       . . .
   COMMON_BLOCK_2
   COMMON_BLOCK_3
```

Fig. 7. Common Blocks

7 Conclusions and Further Work

In general, software changes can not be trivially managed, and legacy software in particular entails one of the worst scenarios. To introduce a change or improvement in an already operational legacy program, a long list of constraints should be taken into account. This process can be extremely complex and a set of tools could be used to assess the changes. We propose the use of a single tool in order to decrease the process complexity and to limit the number of errors that can result from it.

The Legacy software change process should be divided into several stages, including the understanding stage and the transforming stage. These two stages are driven by change. A legacy tool should provide means for carrying out both stages: understanding and transforming tools which should be all integrated into a single tool. We have implemented two tools specifically focused to the understanding stage: the first one is a Static Call Tree View, and the second one is a COMMON BLOCK dependency analyzer. The Static Call Tree View can be used for understanding how the Fortran routines are related to one another. The COMMON BLOCK dependency analyzer shows how COMMON BLOCKS are used by the program routines or files. None of them has been previously integrated in a Fortran IDE.

Further improvements could be made to the Static Call Tree by adding extra information about call nodes, different view styles, and so forth. The COMMON BLOCK analyzer was submitted to be included as a new feature in Photran. Also, having integrated different tools into an IDE provides experience for other tools to be included, each focused to a specific step of the legacy software update process: document, control versioning, specific update/change, check/test.

References

1. American National Standards Institute. X3. 9-1978. *American National Standards Institute, New York*, 1978.
2. American National Standards Institute. *American National Standard for programming language, FORTRAN — extended: ANSI X3.198-1992: ISO/IEC 1539: 1991 (E)*. American National Standards Institute, September 1992.
3. R. S. Arnold. Software restructuring. *Proceedings of the IEEE*, 77(4):607–617, 1989.
4. J. Backus. The History of Fortran I, II, and III. *ACM SIGPLAN Notices*, 13(8):165–180, 1978.
5. I. Baxter, P. Pidgeon, and M. Mehlich. DMS: Program Transformations for Practical Scalable Software Evolution. In *Proceedings of the International Conference on Software Engineering, IEEE Press*, 2004.
6. F.P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE computer*, 20(4):10–19, 1987.
7. E.J. Chikofsky and J.H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE software*, 7(1):13–17, 1990.
8. E.W. Dijkstra. Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.
9. B. Foote and J. Yoder. Big ball of mud. *Pattern languages of program design*, 4(654-692):99, 2000.
10. M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.
11. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Reading, MA, 1995.
12. C. Greenough and D. Worth. The Transformation of Legacy Software: Some Tools and a Process. Technical report, RAL Technical Report TR-2003 012, 2004.
13. J. Hardebeck and P. Shearer. HASH1.2 Calculates earthquake focal mechanisms. http://earthquake.usgs.gov/research/software/HASH/hash.v1.2.tar.gz.
14. R.E. Johnson. Software development is program transformation. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 177–180. ACM, 2010.
15. Jr Kenneth L. Kipp. HST3D:A Computer Code for Simulation of Heat and Solute Transport in Three-Dimensional Ground-Water Flow Systems. http://wwwbrr.cr.usgs.gov/projects/GW_Solute/hst/.
16. M. Méndez, A. Garrido, J. Overbey, F.G. Tinetti, and R. Johnson. Refactorización en Código Fortran Heredado. In *VII Workshop on Software Engineering (WIS), CACIC 2010*, pages 546–555, 2010.
17. M. Metcalf. The seven ages of fortran. *Journal of Computer Science and Technology*, 11(1):1–8, 2011.
18. D. Quinlan. Rose: Compiler support for object-oriented frameworks. *Issues*, 2(3):215–226, 2000.
19. Fortran Lint - Home Page. http://legacy.cleanscape.net/products/fortranlint/.
20. NAGWare Tools - Reference Page. http://www.qaportal.cse.clrc.ac.uk/html/ NagWare/.
21. SPAG - Home Page. http://www.polyhedron.com/spag.html.
22. The eclipse foundation, eclipse.org home. http://www.eclipse.org/.
23. Zest - Home Page. http://www.eclipse.org/gef/zest/.