Fortran Legacy Software: Source Code Update and Possible Parallelization Issues

Fernando G. Tinetti *
Universidad Nacional de La Plata
fernando@info.unlp.edu.ar

Mariano Méndez
Universidad Nacional de La Plata
marianomendez@gmail.com

Abstract

We are working on a process for carrying out a set of transformations on Fortran legacy projects. We started our work for parallelization and reduction of runtime at least on multiprocessing systems, but we found necessary to update several old Fortran features and/or legacy software issues as a previous task. We present how to define and implement several source code transformations in order to enhance readability and, also, provide a source code that we think is more likely to be parallelized in subsequent work. Furthermore, we propose that some parallelization (e.g. for shared memory parallel computers) can be made at least as a tool-guided process, i.e. as *other* source code transformations. In the most simple cases, some transformations from sequential to parallel processing could be made automatically by a source code software analysis and transformation tool, which could provide the user (programmer/developer) a suggested way of paralellization. Source code transformations are initially approached as restructurings, and implemented by changing the abstract syntax tree (AST) program representation. We provide some comments on our preliminary work on the source code transformations directly focused on parallelization that we expect to implement almost automatically.

Keywords Legacy Software, Restructuring, Optimization, Parallelization

1. Introduction

The area of scientific computing in general and numerical processing in Fortran in particular has been producing software over many decades. It seems that throughout this period of time, most of the effort in the context of software engineering has been focused on the development process of brand new software rather than on other stages of the software life cycle. It is widely known that the maintenance stage takes up most of the cost involved in developing software, however this stage is not widely studied. By 1979, some studies found out that the software maintenance costs was about 67% of the total project costs [17]. By the first half of the 1990's the software maintenance cost grew to about to 75% of the total project costs [6], and in 2000 there is a report of this cost as about 90% of the entire project cost [7]. By 1983, studies conducted on software maintenance have revealed that the 50% of the time is devoted to understanding the program [8]. In the year 2000, the number of legacy source code lines was calculated to be 250.000 billions in [14].

It can be argued that Fortran projects do not have those high project maintenance costs mentioned above, or that percentages are different of those in software engineering "in general" or in the E-Business area. However, there are a large number of Fortran applications in actual HPC (High Performance Computing) production environments. For those applications, taking advantage of parallel processing in shared memory parallel hardware can be considered a high-priority requirement. Even small scale computing facilities are made up of multicore processors, so there is no way of taking advantage of current computing facilities beyond parallel computing. One of the main ways of parallel computing in shared memory parallel hardware is achieved by using libraries or the facilities provided by an OpenMP implementation

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

[13], for example. We started our work on Fortran legacy code directly focused on parallelization, but we found many legacy projects almost unreadable to say the least. Many legacy Fortran applications are strongly reactionary to changes, but even most complicated has been to understand the software design and implementation decisions, since we found that many legacy Fortran code is prone to have

- Out of date documentation, or no documentation at all. This is almost usual in legacy code in general and Fortran legacy code in particular.
- Old-style software design methodology or no software design methodology at all. This lack of software design methodology has been partially mitigated by the strong relationship among Fortran programs and mathematical/well defined numerical methods implemented.
- Different programming/programmers styles which has resulted in similar tasks written in different ways. The evolution of Fortran standards and specifically the Fortran standards backward compatibility plus the so-called Fortran compilers "extensions" make this situation even worse.
- Extensive usage of Fortran COMMON blocks in order to share data areas, making those areas statically assigned. Furthermore, in code dated before the 90's it is rather usual to find COMMON blocks for saving memory by overlapping data, which leads to numerous problems. Aliasing data makes it difficult to debug the code and, makes it difficult to identify data as global or local to each subroutine. And identifying local and global data is essential in the parallelization stage.

Thus, we started our work on legacy Fortran code as a complete process [15], mainly because we cannot parallelize what we cannot read/understand. As one of the initial tasks to be done on Fortran legacy code we focus on readability, which can be enhanced in a relatively easy way by means of restructuring.

2. Restructuring

Software characteristics such as changeability, conformity, intangibility, and complexity imply that software developers have to deal with [3]:

- Evolution: Like other human products, software has to change according to people's needs [10].
- Redesign: Unlike other human products, software can be modified even when its development process has been finished. If we take a closer look at manufactured products, like a pen: once it has been produced, the pen can not be changed and it will remain the same until the end of its days.
- Quality: As a consequence of its evolution and redesign, the software quality will be undermined [9].

Even when mathematical models implemented in many Fortran programs may have not changed in the last decades, the Fortran language has been greatly enhanced in almost every new standard [12]. Each new standard has introduced a lot of new *features* as compared to the previous one/s, and these new and better *features* are worth using from many points of view (most of them related to software engineering issues).

Restructuring was initially defined as the modification of software in order to improve readability and make subsequent changes easier and/or less prone to errors [2]. Other software changes like those made for optimization are explicitly excluded in [2] and restructuring is definitely included in the software maintenance process, which concerns not only to the source code but also to the software documentation. Software restructuring is considered a necessary task in the maintenance processes because of the essential features of software, in order to reduce software project (maintenance and future modification) costs. Even when parallelization is our initial objective on legacy code, we have to tackle source code restructuring in order to make easier/possible the parallelization. We are specifically focused on those restructurings directly related to: a) readability, and b) future source code changes for parallelization. Furthermore, we have found that changing old FORTRAN source code format (FORTRAN 66 and FORTRAN 77) and old definition of statements to their equivalents in newer standards (Fortran 90/95) enhances both: readability/understandability and further source code changes. Old source code is also likely to have bad coding practices such as unindented statements. We (and we think most of computer science/software engineering related researchers) find hard nowadays to read and understand Fortran source code with no indentation, lots of GO TOs, different ways of DO LOOPs (mainly those with no END DO and shared termination), etc. Thus, we propose to transform the source Fortran code to use those ways allowed/defined mainly in Fortran 90/95 standards and *enhancing* some programming styles too.

In [2] restructuring mainly consists in preserving or increasing software value. External software value can be increased by fully satisfying the users needs. For users, a good maintenance service means having software which is bugs free (or contains no visible bugs) and a swift response in the case of a request for change. Systems being subject to constant

maintenance may grow increasingly difficult to change. If this hardening should affect the users perception of the software's quality, the external software value will decrease. The internal software value may be measured by the following criteria: 1) the maintenance cost saving resulting from some other software form, 2) the cost savings emerging from reusing parts of the software in other systems, and 3) the cost savings as a consequence of an expanded software lifetime. Restructuring reduces cost of maintenance, as well as increases software re-usability and it extends system's life cycle.

Restructuring is defined in [4] as "... the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior (functionality and semantics)", which is broader in scope than the previous definition. A restructuring transformation is basically applied on the system's internal appearance, its aim is not to introduce new requirements. From this point of view, parallelization can be classified as a restructuring. However, we are not specifically interested in the conceptual definition of restructuring but in its practical application to actual legacy source code. We think that the kind of changes that are necessary for parallelization of some parts of code are strongly *similar* to those of traditional restructurings. By *similar* we mean the definition in terms of pre- and post- source code conditions, actual implementation by editing the program AST representation, etc. This practical view of restructuring lead us to almost always to deal with a restructuring tool, i.e. a software tool which implements and guides the users (mostly programmers) to apply source code transformations.

In view of this, a restructuring tool can be seen as a tool that can assist in solving significant problems that arise during the maintenance stage within the life cycle of the software development process. A summarized list of reasons for restructuring would be [2]:

- Reinforcing understandability of software by injecting software with known and easily decipherable structure, thus having other desirable side effects:
  - simpler documentation,
  - simpler testing tasks,
  - simpler auditing tasks,
  - substantial reduction of software's complexity .
- Reducing the necessary time for programmers to get acquainted with the system before implementing maintenance tasks.
- Making bugs easier to locate.
- Making it simpler to introduce new software functionality.

As a matter of fact, we started our work on restructuring as a stand-alone task [11] because we found it useful and big enough in order to avoid other related tasks. However, we soon realized that tackling changes on legacy software cannot be made independently of other essential software development tasks such as documentation, versioning, and, almost fundamental: testing (which in fact we usually take by granted). Currently, we are focusing our work in the context of a *complete* restructuring process as shown in Fig. 1 [16].

3. Fortran Restructuring for Legacy Systems: Proof of Concept Implementation

The real value of restructuring comes from its automation and, as a plus, its integration in development environments. Manual restructuring is time-consuming and prone to errors in general, and the situation is not different for Fortran. Restructuring (or at least many of the restructuring transformations) can be easily applied with a tool able to

- Analyse the source code preconditions under which a transformation is "safe".
- Perform the transformation. i.e. actually change the source code.
- Check that the source code has not been broken, i.e. that source code postconditions specifically defined for the transformation are met.

For this purpose, we propose and use restructuring engines which parse the code, create a program representation based on abstract syntax trees (AST), and perform analysis and transformations on the AST. Even when source code transformations could be implemented by simpler methods such as find-replace and other text processing tool/s, we think AST transformation is better in several ways:

- The AST provides direct access to program structures/definitions/constructs/*elements*, text-only tools have to incorporate processing for identifying those in a program. We actually found (and had to work with) a legacy program containing

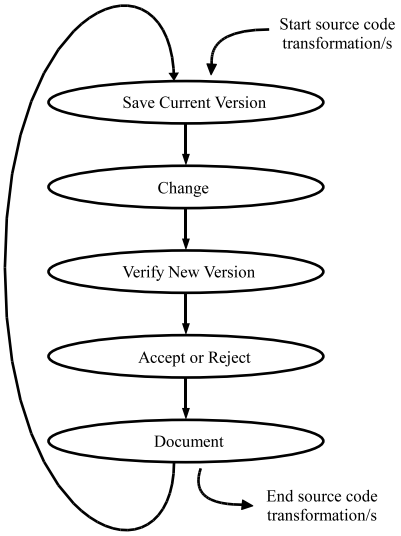

Figure 1. Incremental Process of Source Code Transformation

```
10 if = if + 5
   ifs = ifv(if)
   if(ifs-3)100,200,300
20 if(s1*7-s2)400,400,500
30 if(ifv(if+1)-10)600,700,600
```

on which the usage of "if" is very difficult to *classify* as a variable or the execution control IF construct (beyond the usage of the *obsolescent* arithmetic IF).

- The program analysis can be defined in terms of well-known tree traversals. In general, every software engineer is able to take advantage of the well known *tree* data type operations in order to define and implement source code transformations.

The AST could be constructed to contain enough editing information to be able, for example, to pretty-print the source code while maintaining the original semantics and syntactical elements. Even when the restructuring concept and Fortran source code transformations are independent of the implementation, we found that Photran has the necessary infrastructure for adding restructurings (implemented as *refactorings* in Photran) in the development process. Photran is a plug-in and Eclipse-based Fortran IDE (Integrated Development Environtment), and an open-source project with many contributors [19]. Fig. 2 shows the menu of Photran regarding source code transformation in version 6 and version 7. The Photran 6 menu (the left picture in Fig. 2) has many source code transformations available as a plain list. The Photran 7 menu (the right picture of Fig. 2) included so many new source code transformations (and many of them defined and implemented in the context of [11]) that it was necessary to define and construct submenus in the pull-down menu of Fortran source code transformations. It is worth noting that Photran 6 was released in June 2010 and Photran 7 in June 2011. The growing number of source code transformation shows the current interest of researchers for Fortran code update and enhancement and, if the growth is sustained, some kind of ranking by usefulness will be needed soon. We are currently working on some metrics and/or methodology for this specific issue: how many and in which order to apply source code transformations.

3.1 The Program Representation in Photran

Photran is an open source project implemented in Java which contains two very important data structures regarding Fortran source code:

- The first one is the AST, which maintains the entire representation of a Fortran program. The AST structure is made up of AST nodes, a set of classes that represent each possible element of the programming language, where each node of the tree denotes a construct occurring in the source code. The most remarkable aspect about applying changes to source code is that all transformations applied into the source code are performed by changing the AST structure.

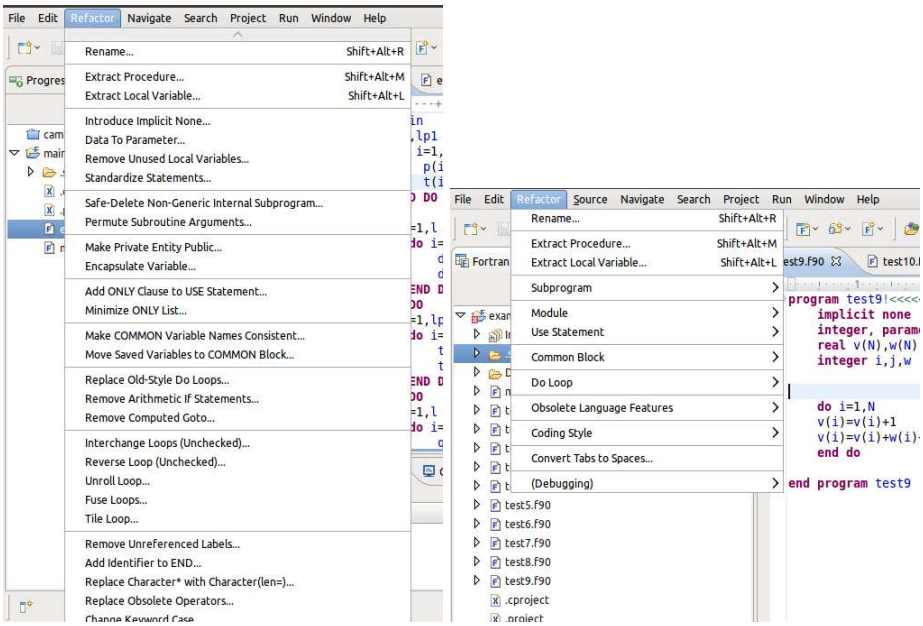

Figure 2. Photran 6 (Left) and Photran 7 (Right) Code Transformation Menu

- The second important structure is the VPG (Virtual Program Graph) which facilitates the handling of the AST and the embedded analysis information, acting as a facade between the AST and the programmer. Thus, the VPG allows restructuring programmers to acquire or release AST nodes, and it also sets off scope and binding analysis, while allowing the user to obtain variable definitions, and so forth.

3.2 Photran Restructuring Infrastructure

Every source code transformation in Photran is optional for the user, the programmer has to select and approve every transformation in particular. As an open source project it is possible to add new functionality to Photran, and in particular, new source code transformations. Photran divides source code transformations (called *refactorings* in Photran terms) into two categories:

- *Editor-based* transformations, which require the user to select part of a Fortran program in a text editor in order to initiate the transformation,
- *Resource* transformations, which apply to the files selected by the user.

The developer must decide whether it will be an editor-based or a resource one in order to create a new restructuring. Photran provides different super-classes for each. The developer then creates a concrete subclass and adds a line of XML to a configuration file to make Photran aware of the new Fortran source code transformation. The concrete subclass must define methods which first provide the name, which becomes its label in the Photran *Refactor* menu. It is also used to describe the transformation in the *Edit* > *Undo* menu item and in other user interface elements.

Once the transformation has been added as defined above, the initial preconditions have to be checked. These are usually simple checks which verify that the user selected the correct construct in the editor, that the file is not read-only, and so forth. Some source code transformations require user provided data, which is requested after the initial preconditions have been checked successfully. For example, a restructuring to add a parameter to a subprogram must ask the user to supply the new parameter's name and type. Then, the final preconditions are checked. These validate user input and perform any additional checks necessary to ensure that the transformation can be performed, the resulting code will compile, and it will retain the behavior of the original program.

Once successfully carried out every previous step, the transformation is actually made. Once all preconditions have been checked, this method determines what files will be changed, and how. Given the XML configuration file and Java's reflective facilities, much of the user interface for a restructuring comes "for free". When the previous steps have been implemented and included in the Photran source code, the source code transformation appears as a new Photran *refactoring* in the appropriate parts of the user interface. Also a wizard-style dialog box is provided, which allows the user to interact

with the source code transformation. This dialog includes a *diff*-like preview, which allows the user to see what changes will be made before committing it.

3.3 Actual Changes

We present in this section some of the source code transformations explained in [11], and some on which we are currently working. All of them are prone to be enhanced, and we are currently trying to define some metric/s in order to make a ranking of obsolescent statements/constructions or just those prone to be replaced by better statements/constructions.

3.3.1 DO Loops

DO constructs are among the first language constructs we were and are interested in, since our focus is parallelization of compute intensive sections of code, and those sections are usually found in DO constructs. One of the restructurings we implemented is the so called Replace Old-Style Do-Loops. There are many different ways to write a DO loop in Fortran, depending on the standard of Fortran being used. "Old-style" DO loops contain a numeric statement label in the loop header; the statement with that label constitutes the end of the loop as shown in Fig. 3. In contrast, "new-style" DO loops consist of matched DO END DO pairs as also shown in Fig. 3, which are generally preferred. In terms of the Fortran 95

```
    ....                    ....
    DO 100 I=1,30           DO 100 I=1,30        DO I = 1, 30
    V(I)=0             100  V(I)=0                 V(I) = 0
100 CONTINUE                ....                 END DO
    ....                    ....
           a) "Old-Style"                        b) "New-Style"
```

Figure 3. "Old-Style" and "New-stlye" Fortran Do Loops

Standard [1], we want to transform every *nonblock-do-construct* in a specific *block-do-construct*:

```
DO loop-control
  do-block
END DO
```

The "Replace Old-Style Do-Loops" restructuring was implemented as a source code transformation applied to files (a so called *resource refactoring* in Photran) as follows:

- Preconditions: The source code must have at least one DO.
- Transformation: The label is deleted, as well as the CONTINUE statement if present, an END DO is added in the corresponding place, and the *do-block* is re-indented. Figure 4 shows the diff-like preview of this source code transformation as currently implemented in Photran.

In Figure 5 the Photran's AST structures are displayed. The picture at the left of the figure shows the AST nodes before changes were applied, where the node labelled getEndDoStmt() is pointing to null. The picture at the right of the figure shows the AST of the resulting code where the same node, getEndDoStmt(), is pointing to an AstEndDoStmtNode.

3.3.2 Initial Work on Change Fixed to Free Form

One of the oldest and most tedious tasks to be performed is the transformation of Fortran source code written in Fixed Source Form into Free Source Form [18]. The Fortran fixed form was the only way to write a Fortran program for more than 35 years, until the Fortran 90 standard [1]. An important point regarding Fortran Fixed Form is that spaces can be found either in the middle of a statement or in the middle of a variable name, for example "D O100I=1 ,10" that is equivalent to "DO 100 I = 1, 10". Transforming these programs by hand can be a very error prone task. We are in the process of implementing the change of old source code form and, currently, we already have sucessfully tested two changes:

- Transform fixed form comments (lines starting at position 1 with character 'c', 'C', '*') into free form comments that starts with '!' character.
- Transform fixed form continuation lines (any character at position 6) into free form continuation lines ended with character '&'

both of which are explicitly defined in the standard as part of source code form. Change Fixed to Free Form is applied automatically on each file selected in the user interface by the programmer. Fig. 6 shows an example of our current implementation. We also change the name of source code files from .f77, .F77 or .f to .f90 as in [18]. We know that source

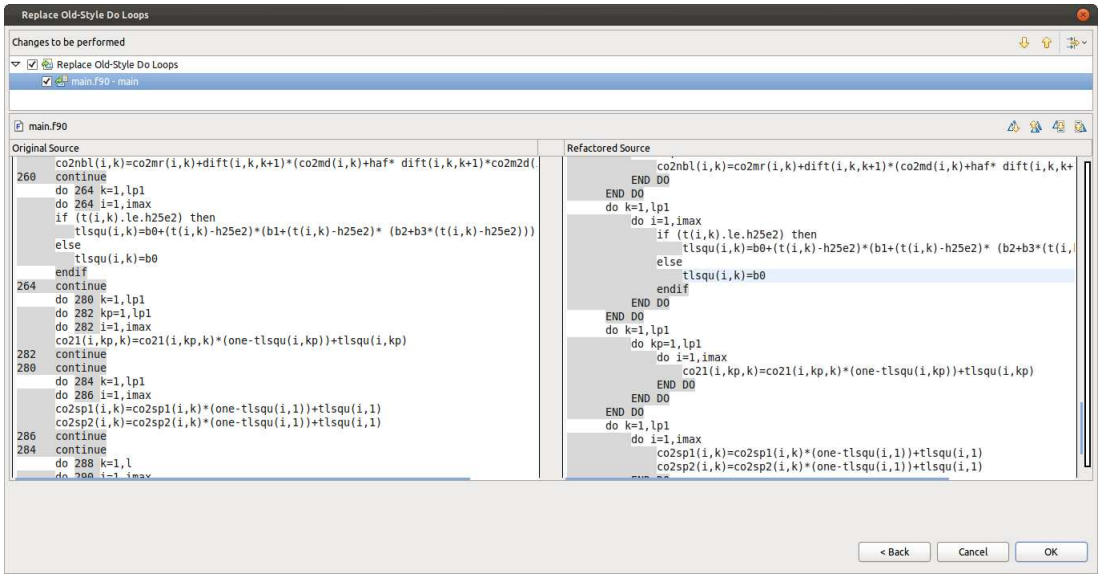

Figure 4. Replace Old-Style Do-Loops Transformation in Photran

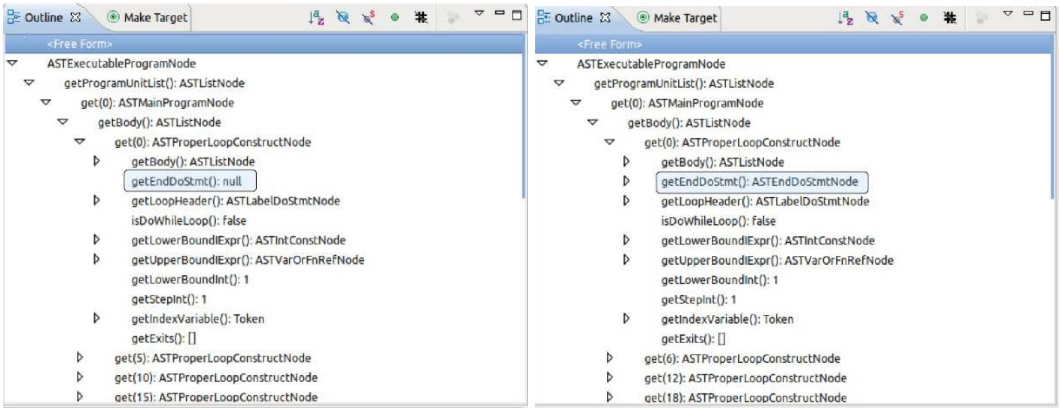

Figure 5. Photran AST Representation Before and After Source Code Transformation

code file name is not part of any Fortran standard, but we also know that every developer has to deal with source code project/s, source code files, etc. We have decided not to deal with indentation issues/decisions as part of this restructuring, instead, we are focusing on definitions/details included in the standard such as blank management in fixed form. It is possible that we deal with every indentation-related issue in a separate source code transformation (note, however, we have included a *pre-determined* indentation in the DO loop transformation as shown above in order to better visualize the do-blocks).

3.3.3 Remove Unreferenced Labels

The motivation of this restructuring is to remove labels no longer referenced from the source code. Unreferenced labels can be left in the source code because the label has never been referenced or because a previous source code change was made and the label is no longer referenced. To remove unreferenced labels, we must first recognize all the labels, which can be accomplished by constructing a list of labels gathered by traversing the AST. Additionally, we need to find out the number of references to each label. Finally, we will remove from the source code each label with the reference count equal to 0, and if the labelled statement is a CONTINUE, then it will be removed too. Fig. 7 shows an example of the Remove Unreferenced Labels source code transformation.

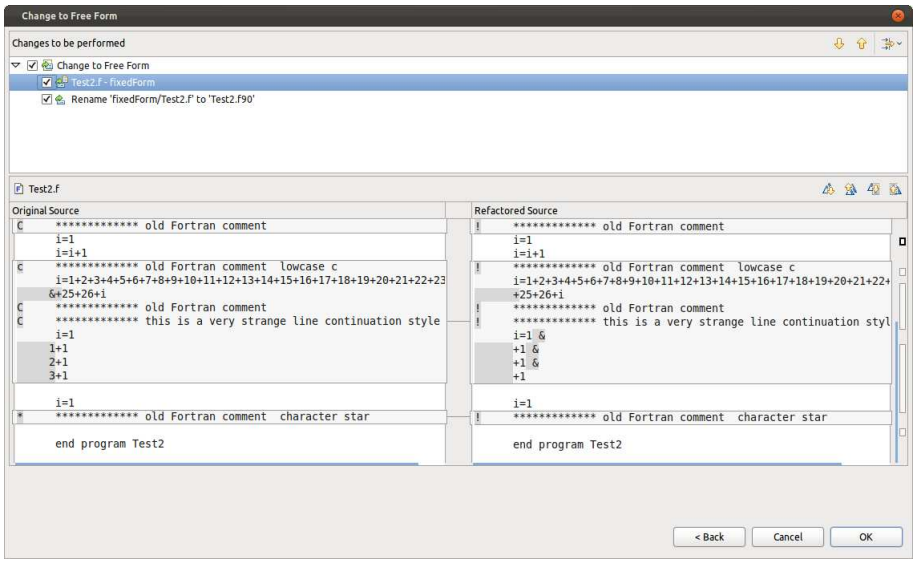

Figure 6. Photran Fixed to Free Form Transformation

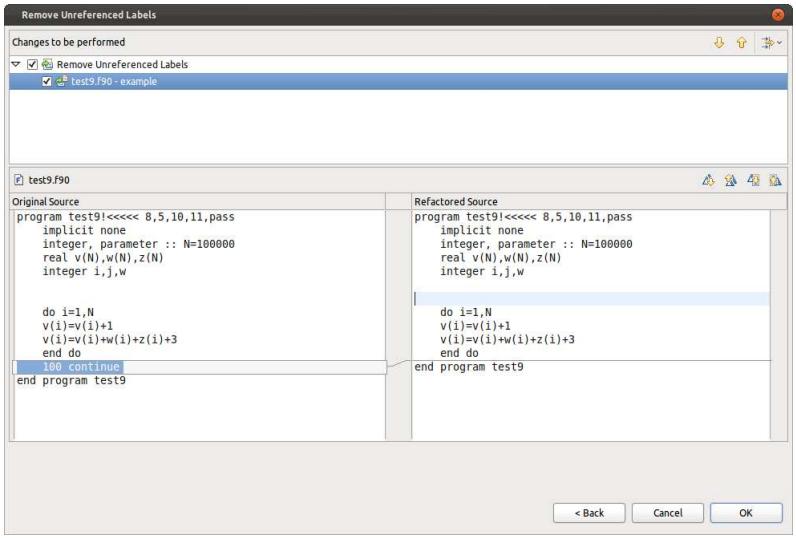

Figure 7. Photran Remove Unreferenced Labels

3.3.4 Remove Unused Parameters

We have found programs using the same Fortran parameters (constant values) declarations in almost every routine/source code file, these declarations may contain a large number (tens or thousands) of different parameters that seem to have been included in every program routine by copying and pasting the same set of declarations. This copy-paste process may lead to some declared parameters that are not currently used on every routine. Not used constants should be removed from each routine, at least. There are other possibilities, such as using a module for all the parameters or just a file included in every routine, but there are other issues involved such as defining what to do with parameters used in a fraction of source code files/units. Removal of unused parameters can be easily performed by traversing the entire AST structure and counting each referenced constant inside its own definition scope. After that, we can remove each constant that summarizes 0 in the set of references. Fig. 8 shows the current implementation. Note that the code of identifying and removing unused parameters can be almost re-used for removing unused *variables* and subprogram dummy arguments (subprogram *parameters* in most programming languages other than Fortran). We have currently implemented only the Remove Unused Parameters just because we have found more unused parameters than unused variables or subprogram dummy arguments.

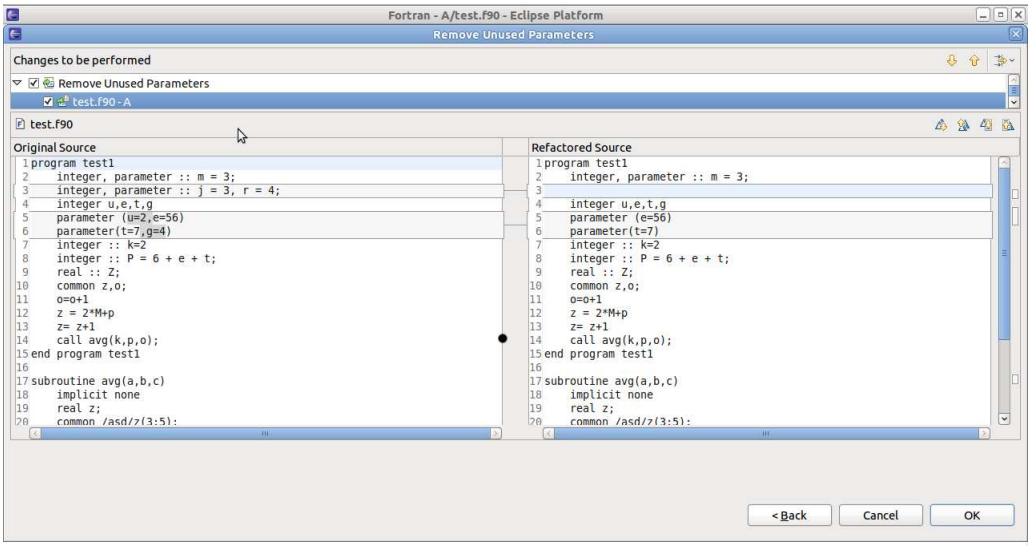

Figure 8. Photran Remove Unused Parameters Transformation

3.3.5 Change to Array Notation

Fortran 90 standard has introduced the ability to manipulate entire arrays or array sections [1], for example

V(1:N)=W(1:N)*SIN(1:N)

Array *notation* may facilitate and simplify some mathematical/numerical operations implemented in Fortran. Thus, an automatic transformation that changes a DO loop statement into array notation may enhance programmers readability/understandability. This change in source code notation imply to select a DO loop statement in the editor user interface (this would be an editor based transformation in Photran terms). As one of the current transformation preconditions, we have defined that every array in the DO should be accessed only via the *do-variable* (the loop control variable), and this transformation changes the entire source code of the *do-block* (that between DO - END DO), which will be written in array notation. Each reference to the *do-variable* is changed to LOWER BOUND:UPPER BOUND. An example is shown in Fig. 9. We are currently engaged in several analysis tasks related to this source code transformation, including performance issues,

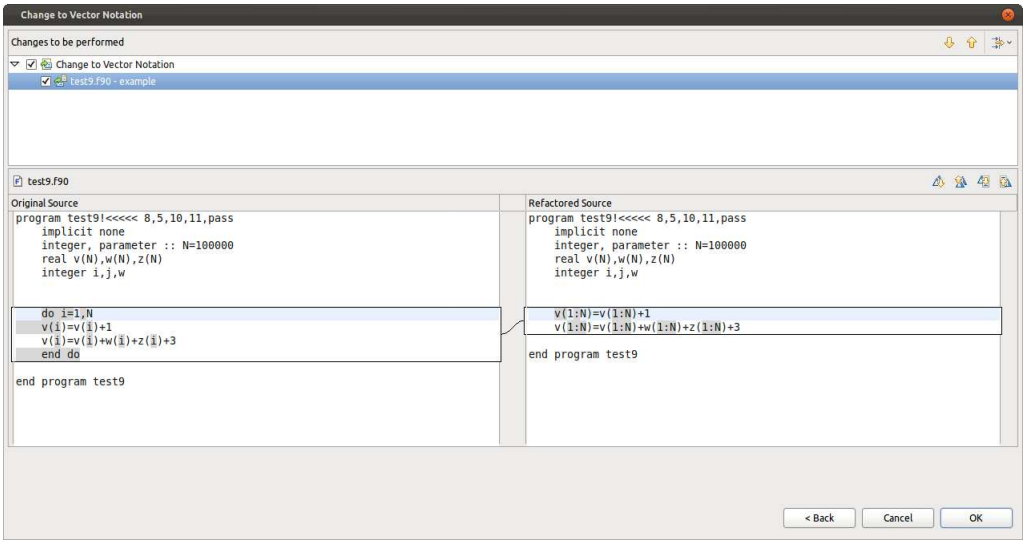

Figure 9. Photran Change to Array Notation Code Transformation

since several array expressions may lead to unnecessary array copies, and prevent some useful optimization/s and, thus, penalizing performance. On another hand, we found that the precondition checking for this source code transformation could be used for parallelization via OpenMP, for example.

4. A survey of restructuring tools

We think there are mainly two types maintenance tools available for Fortran programs:

1. Software for verifying programs: these tools verify and check language features, common programs errors and the program structure. The products that fall under this kind of tools are:
   - Forcheck
   - Fortran90-lint
2. Software for transforming programs: these tools verify and check language features, common programs errors and the program structure, and also apply source code transformations in order to update or improve the internal software structure. The products that fall under this kind of tool are:
   - Convert
   - FOR STRUCT
   - NAGWare Fortran Tools
   - Photran
   - PlusFORT
   - VAST/77to90

Both classes of tools assist in the process of restructuring and/or directly enhancing the legacy Fortran source code. While some of them directly restructure the code automatically, others suggest changes and/or identify parts of the source code to be enhanced/changed.

We have selected a set of tools listed in chapter 8 of [5]: "Fortran Analysis, Conversion, Maintenance, and Refactoring Tools". We also list some of the most significant functionalities of those tools regarding our Fortran restructuring and Fortran source code transformation point of view. The revision was based on features described in the web site or product specification published by each company/developer, but an experienced user may find some missing functionality. The tools and their functionalities are shown in table 1, where

- Y stands for "Yes, supported/implemented functionality".
- N stands for "Not (or N/A) supported/implemented functionality".
- chk stands for "The source code is checked and some warning is reported if needed, but no change is actually made to the software"
- Y^* stands for "Implemented in a development version of Photran, not present in the current Photran version". We have taken advantage of Photran as an open source project: we added the functionality we think is relevant for our research on Fortran legacy code management.

Some functionality of Photran is still in committing process. The static call tree, which is not part of the source code transformation options, but used for source code analysis/understanding can be seen in Figure 10. It is worth noting that there are not tools supporting the transformation of sequential code into parallel code. Parallelization is not an easy task in general and it seems to be even harder on Fortran legacy code.

5. Source Code Transformations and Parallelism

We do not forget our initial motivation for working on Fortran legacy code: parallelization. Some of the restructurings explained above are directly related to parallelization issues, such as those explained in the subsection "Change to Array Notation". While others are not so directly related, the impact on the parallelization job is significant. One of the most representative cases is that of data access/usage analysis necessary for parallel processing in distributed memory parallel computers. Even when we are now engaged in parallelization for shared memory parallel computers via OpenMP, we must take into account that parallel computing on distributed memory parallel platforms imply at least some way of data distribution and data exchange (communication) with minimum data replication. All (or most) of the analysis and source code transformation regarding Fortran *data objects* is strongly related to parallelization for distributed memory parallel processing. More specifically, COMMON blocks define, in fact, static memory for potential exchanging of data among subprograms, and if those subprograms are going to be executed in different processors/computers (as in a cluster) some way of communication and synchronization will be needed.

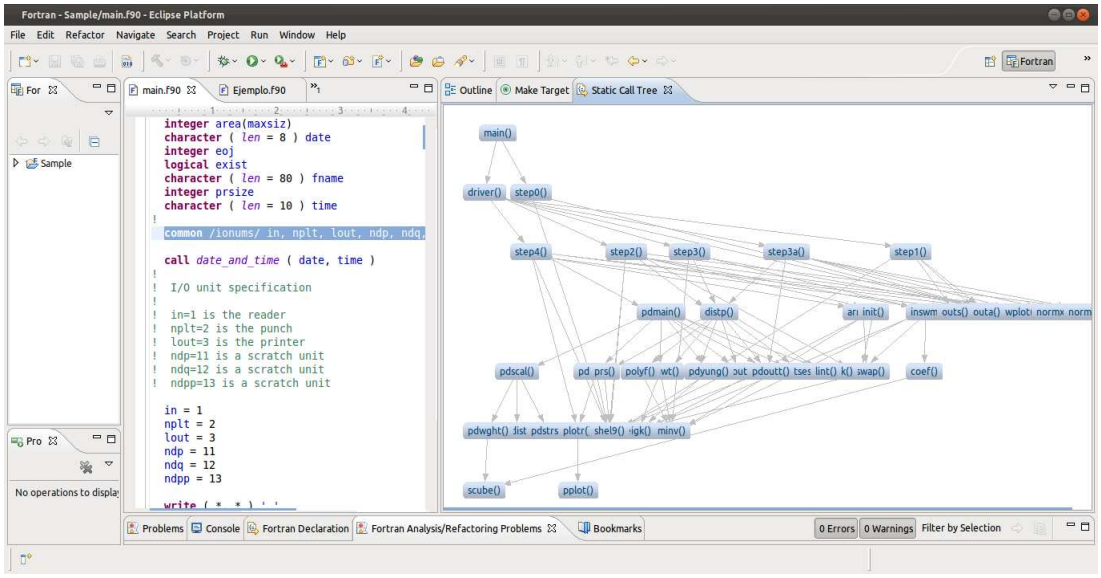

Figure 10. Photran Static Call Tree Proof of Concept

Our current approach to, and research on, Fortran legacy source code parallelization is based on source code transformations similar to those defined for restructuring. As a matter of fact, source code transformations for parallel processing can be considered as plain restructurings from the point of view of the definition given in [4]. Instead, we prefer to maintain the concept of parallelization as a different one, but, more important, the practical implementation is approached as the restructurings explained above: analysis and transformation of the Fortran AST program representation. Furthermore, we consider our approach to be independent of Photran and we expect to experiment using ROSE [20].

6. Conclusions and Further Work

This work has explored a way to update and enhance Fortran legacy software by using software transformation tools. Our first objective for restructuring is more readable and understandable legacy source code. Furthermore, we think that long-lived programming languages need tools for allowing them to evolve or, at least, to allow the evolution of the source code written in those languages. This kind of tools are not easily built, they require a rather complex engine to allow programmers to build transformations.

Fortran has had a particular evolutionary process through different versions across time, about ten language versions have been published in the last 50 years (six of them were standards). These versions have transformed Fortran into a language with a rich set of syntactical constructions. As a consequence, programs written (many) years ago need to be updated and improved with modern constructions and new language features. We found that analysis and modifications on the AST program representation are not only possible but make source code changes easier to be applied.

There is a lot of work still to be done, in *traditional* restructuring as well as in legacy software parallelization. In terms of restructuring, we are currently engaged in

- Defining some metrics and/or methodology for applying source code transformations. We included some features of restructuring tools in Table 1, but there is no methodological way of setting priorities for some source code transformation/s. We have our preferences, of course, but we think it is necessary some analysis and quantification on legacy source code in order to have a *roadmap* of restructurings to be applied to legacy source code. We would like to avoid an implicit “modify everything” rule which seems to be the rule so far.
- Advance on the analysis and source code transformation related specifically to data. We think that all we learn to analyse and handle data will help us in the parallelization stage. Just as a simple example: we will not start parallelizing Fortran source code without “Implicit None”, because “Implicit None”, since data should be analysed at least for possible distribution and data races in parallel processing.

In terms of parallelization, our current goal is that legacy software take advantage of current multiprocessing systems, made up of multi-core processor/s. Thus, parallelization for shared memory parallel platforms is our first goal, and we expect to directly use OpenMP for several reasons:

Functionality	Forcheck	Convert.f90	FortranLint	For_Struct	plusFORT	Photran	VAST/77to90	NAG Fortran Compiler
General Features								
Graphic interface	Y	N	Y	N	N	Y	Y	N
Command line interface	Y	Y	Y	Y	Y	N	Y	Y
IDE integration	Y	N	N	N	N	Y	N	Y
Batch/Interactive	B/I	B	B/I	B	B	I	B/I	B
Open source	N	Y	N	N	N	Y	N	N
Source Code Restructuring								
Replace COMMON by Module	N	N	N	N	Y	N	Y	N
Replace INCLUDE by Module	N	N	N	N	Y	N	Y	N
IMPLICIT None needed	chk	N	chk	N	Y	Y	N	chk
Rename Variables	N	N	N	N	Y	Y	N	N
Remove GO TO Statement	N	N	N	Y	Y	N	Y	N
Remove Arithmetic IF Statement	N	N	chk	Y	Y	Y	Y	N
Remove Computed GO TO Statement	N	N	chk	Y	Y	Y	Y	N
Combine sub-programs files in a single one	N	N	N	N	Y	N	N	N
Add END DO	N	Y	N	Y	N	Y	Y	Y
Transform fixed to free form	N	Y	N	Y	N	Y*	Y	Y
Change to Array Notation	N	N	N	N	N	Y*	Y	N
Upper/Lower case	N	N	N	Y	Y	Y	N	N
Remove unreferenced common blocks	chk	N	chk	N	Y	Y*	N	N
Common blocks inconsistencies	chk	N	chk	N	chk	N	N	N
Remove unused variables and functions	chk	N	chk	N	Y	Y	N	N
Variables used but not set	chk	N	chk	N	Y	N	N	N
Input/Output arguments	chk	N	chk	N	chk	chk	N	N
Source Code Analyses								
Static Call tree	Y	N	Y	N	N	Y*	N	Y
Symbol Table	Y	N	Y	N	Y	Y	N	Y
Automatic Shared Memory Parallelization								
Open MP Parallelization	N	N	chk	N	N	N	N	N
CUDA Parallelization	N	N	N	N	N	N	N	N

Table 1. Several Software Tools Assisting Fortran Source Code Maintenance

- OpenMP has been implemented by all major compilers, so it is widely available.
- OpenMP directives can be directly inserted in the source code, there is no need to call external libraries.
- We think that source code transformations via the AST such as those we explained above can be used for parallelization with OpenMP. We are working on some of them, specifically on DO loops.

Restructuring and parallelization is not the whole universe of our work on Fortran legacy software. We are also working on several analysis functionalities such as that shown in Fig. 10, which is directly oriented to documentation and helps in understanding the legacy software. We have not started our work on the testing phase of code, which is directly related to the "Accept or Reject" step in Fig. 1. Instead, we have used the *standard* black box input/output methodology which can be successfully exploited in numerical applications. However, we will define some other methodology/specific testing aids to developers for the decision of committing a specific change.

References

[1] American National Standards Institute, American National Standard for programming language, FORTRAN — extended: ANSI X3.198-1992: ISO/IEC 1539: 1991 (E), 1992

[2] R. S. Arnold, "Software restructuring", Proceedings of the IEEE, Vol. 77, No. 4, 1989.

[3] F. P. Brooks, "No silver bullet: Essence and accidents of software engineering", IEEE computer, Vol. 20, No. 4, 1987.

[4] E. J. Chikofsky, J. H. Cross, "Reverse Engineering and Design Recovery: A Taxonomy", IEEE software, Vol. 7, No. 1, 1990.

[5] I. D. Chivers, J. Sleightholme, Fortran Resources, 2012, `http://www.fortranplus.co.uk/resources/fortran_resources.pdf`

[6] A. Eastwood, "Firm Fires Shots at Legacy Systems", Computing Canada, Nol. 19, No. 2, 1993.

[7] L. Erlikh, "Leveraging Legacy System Dollars for E-Business", IT Professional, Vol. 2, No. 3, 2000, IEEE Computer Society.

[8] R. K. Fjeldstad, W. T. Hamlen, "Application Program Maintenance Study: Report to Our Respondents", Proceedings GUIDE 48, 1983.

[9] B. Foote, J. Yoder, "Big Ball of Mud", Pattern Languages of Program Design, Vol. 4, N. Harrison, B. Foote, H. Rohnert, (Eds.), Addison-Wesley, 2000.

[10] M. M. Lehman, "Programs, Life Cycles, and Laws of Software Evolution", Proceedings of the IEEE, Vol. 68, No. 9, 1980

[11] M. Méndez, Fortran Refactoring for Legacy Systems, MSc Thesis, Computer Science School, Universidad Nacional de La Plata, Aug. 2011. Available at `http://hpclinalg.webs.com/FRLS.pdf`

[12] M. Metcalf, "The Seven Ages of Fortran", Journal of Computer Science & Technology (ISSN 1666-6038), Vol. 11, No. 1, April 2011, pp. 1-8. Available at `http://journal.info.unlp.edu.ar/journal/journal30/papers.html`

[13] OpenMP Architecture Review Board, "OpenMP Application Program Interface - Version 3.1", July 2011. Available at `http://openmp.org/wp/`.

[14] I. Sommerville, Software Engineering, 6th Edition, 2000, Addison-Wesley.

[15] M. Méndez, F. G. Tinetti, "First Steps Towards a Tool for Legacy Systems", XVII Congreso Argentino de Ciencias de la Computación, UNLP, La Plata, Argentina, Oct. 2011. Available at `https://lidi.info.unlp.edu.ar/~fernando/publis/082.pdf`

[16] F. G. Tinetti, M. Méndez, M. A. Lopez, J. C. Labraga, and P. G. Cajaraville, "Update and Restructure Legacy Code for (or Before) Parallel Processing", Proceedings of the 2011 International Conf. on Parallel and Distributed Processing Techniques and Applications, Vol. 1, CSREA Press, July 2011, Las Vegas, USA, ISBN: 1-60132-193-7, pp.652-658.

[17] M. V. Zelkowitz, A. C. Shaw, J. D. Gannon, Principles of Software Engineering and Design, 1979, Prentice Hall Professional Technical Reference

[18] M. Metcalf, Program Convert, CERN, Geneva 1991, `http://www.nag.co.uk/nagware/Examples/convert.f90`

[19] Photran, an Integrated Development Environment and Refactoring Tool for Fortran, `http://www.eclipse.org/photran/`

[20] ROSE, `http://www.rosecompiler.org`