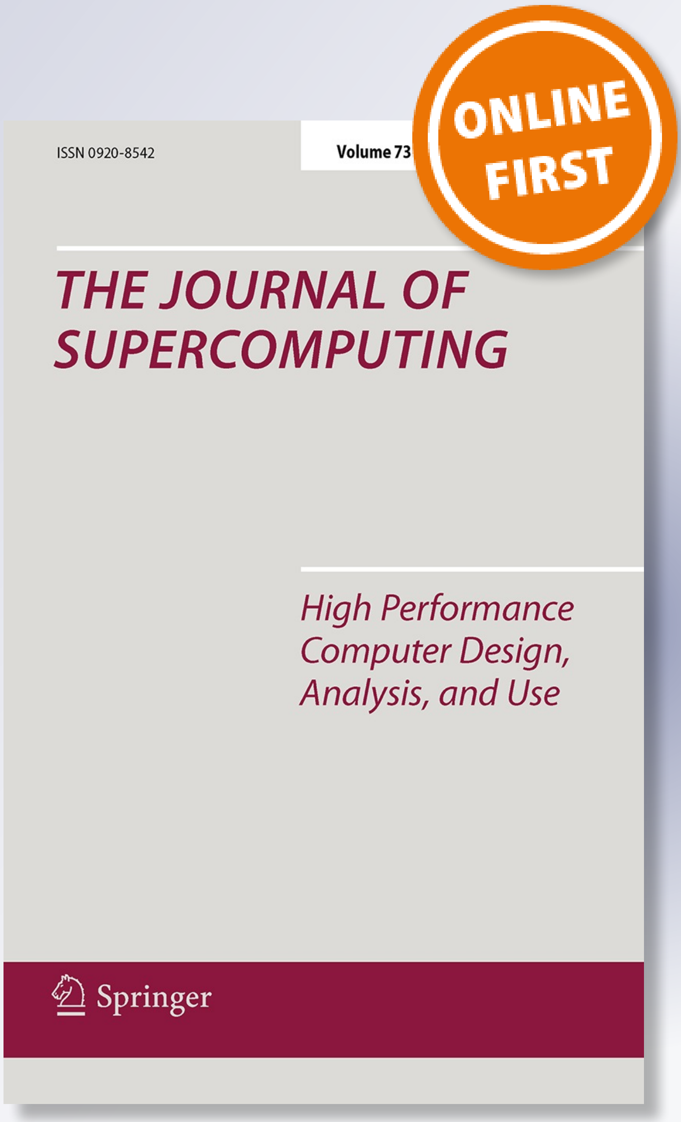
Change-driven development for scientific software

Mariano Méndez & Fernando G. Tinetti

The Journal of Supercomputing
An International Journal of High-Performance Computer Design, Analysis, and Use

ISSN 0920-8542

J Supercomput
DOI 10.1007/s11227-017-1966-1


ISSN 0920-8542
Volume 73
ONLINE FIRST
THE JOURNAL OF SUPERCOMPUTING
High Performance Computer Design, Analysis, and Use
Springer

J Supercomput
DOI 10.1007/s11227-017-1966-1

Change-driven development for scientific software

Mariano Méndez[1,2] · Fernando G. Tinetti[2,3]

Abstract Scientific software production dates back to the days before the computer science discipline obtained its own name. Over the past 76 years, scientists have been producing software, which means that most of the modern techniques and software engineering methods available these days did not exist while part of this process was taking place. Change-driven development was born as a new approach to maintain and develop scientific software. Founded on the principles of software essence (changeability, complexity, intangibility, and conformity), integrated development tools, and automated source code transformation. This new, agile approach takes change as a working unit devised to drive the entire development process, which is performed in a four-stage cycle. One of the most interesting approaches to apply change-driven development on scientific software is to update, modernize and even parallelize sequential programs that have been written 20 or 30 years ago and are still running in production environments. This process will be thoroughly described and implemented. Two successful case studies will be presented and analyzed in depth.

Keywords Change-driven development · Scientific programming · High performance computing

✉ Fernando G. Tinetti
fernando@info.unlp.edu.ar

Mariano Méndez
marianomendez@fi.uba.ar

1 Facultad de Ingeniería, Universidad de Buenos Aires, Buenos Aires, Argentina

2 III-LIDI, Fac. de Informática, UNLP, La Plata, Argentina

3 Comisión de Inv. Científicas de la Prov. de Buenos Aires, La Plata, Argentina

Published online: 31 January 2017

1 Introduction

The first attempts at scientific programming on computer machines were made in the 1940's and 1950's by the computational simulation of nuclear explosions of Von Newman [87], the ballistic trajectory computations of Dederick [26], the first stored-program computer [17,74], and the first weather forecast that ran in a computer based on Berjkess equations Jule Charney who performed the first automated regional weather forecast on a computer in 1949. This model ran on the Electronic Numerical Integrator And Computer (ENIAC) [30,54]. The only development process which was known at the time was "Code and Fix." Scientists had been producing software before the term "software" was coined by Tukey [86], or the concept of "byte" being mentioned by Brooks, Blaauw, and Buchholz [15], or the waterfall development process was created by Benington [11].

Nowadays, scientific programming is the ancient kind of software ever produced, and most of it was built in Fortran. It is the most long-lived programming language [4,56,60], which has been widely used by scientists to produce a profuse source code ever since it came into existence; it has come to be known as the "de facto" scientific programming language [6,43,72,76]. The first edition of the Fortran user manual was published on October 15, 1956, by a team from IBM run by Backus [4]. Throughout its long-lived existence, it has experienced a singular evolution which resulted in this programming language to be the first one to be ever standardized [2,75], among other things. Fortran has survived 60 years of changes within the computer science, it has adapted to programmers' needs in keeping with the times and technology. While according to "popular culture" of modern software development, Fortran is an outdated, fossilized, outmoded and obsolete programming language, its history and its evolution proves the polar opposite by boasting a dynamic language which has adapted to the needs of programmers. In order to perpetuate these positive qualities this language adopted a noteworthy evolutionary process that evolved on both fronts the formal one through the standards, and the pragmatically one through the industry. This process has been managed to maintain backward compatibility to such a degree so that programmers can compile a Fortran 66 program by using any modern compiler available.

One interesting affirmation about the role that scientific software plays has been proposed by Judith Segal [62]. She asserts that scientific software acts as a medium by which a erstwhile generation of scientists can transmit the knowledge encapsulated in software to a next generation of scientists. This pronouncement reflects the hereditary and legacy nature of knowledge and software. Throughout the last six decades computer science has evolved in a vast set of different fields including software engineering [81]. If the evolution of scientific software is compared with other kinds of software, it can be seen that the former has taken a different path in that it was slower than the latter. Furthermore, scientific software not only seems to stay one step behind other kinds of software but it also seems to have taken a totally different approach. Some authors describe this phenomenon as the "gap," the "chasm" between software engineering and scientific computing [8,44,58]. In this article, a new approach for scientific software development is introduced. This new technique has been called change-driven development (CDD). This new idea is founded on Brooks' four software essence: changeability, complexity, conformity and invisibility; especially on changeability [16]. Another fundamental approach that contributed to CDD foundations is Ralph Johnson viewpoint "Since most programmers are working on software that they did not start, their view of programming is that it is the process of converting one version of software to the next. In other words, software development is program transformation" [42], under this view point the idea that most software projects are built from nothing could at least be considered naive, specially when working with scientific software. change-driven development adopts change as a unit of work which is based on automated and integrated analysis and transformation tools and uses an iterative and incremental cycle to software development or maintenance.

In this article, change-driven development is introduced as an alternative software maintenance and development process for scientific software building. The process is thoroughly described and applied on an example.

The structure of this paper is as follows. Section 2 the theoretical docus and related works are described. Section 3 describes the main features that characterize change-driven development. Additionally, Sect. 4 presents a thorough descriptions of CDD and their components. Sections 5 and 6 present two step by step examples of CDD usage on Fortran source code. Finally, Sect. 7 presents conclusions and future work.

2 Theoretical focus

Traditional software development processes or the so-called plan-driven software development processes that "approaches along a spectrum of increasing emphasis on plans, ... In this context, the term "plan" includes documented process procedures that involve tasks and milestone plans, and product development strategies that involve requirements, designs, and architectural plans" [13] seems not to be widely used or at least frequently used by scientist to produce their software. One thought-provoking assertion about the role that scientific software plays has been proposed in [62] by Judith Segal. In this research work, it is asserted that scientific software acts as a medium by which a former generation of scientists can transmit the knowledge encapsulated in software to a succeeding generation of scientists. This assertion reflects the hereditary nature of knowledge and software. Additionally, it is important to note that a set of statements claiming the existence of a gap, chasm or significant divergence between scientific software production and commercial software production has been found in bibliography. If the evolution of scientific software is compared with other kind of software, it can be seen that the former has taken a different path in that it was slower than the latter. Furthermore, scientific software not only seems to stay one step behind other kind of software but it also seems to have taken a totally different approach. Some authors describe this phenomenon as the "gap" or the "chasm" between software engineering and scientific computing. The current software building practices used by computational scientists often bear little resemblance with those promoted by software engineers [8,20,28,37,44,79]. There exist a set of different reasons to explain this fact, a possible one could be that it is commonly found that the role of the end-user and programmer fall upon the same person [8,20,37,44,45,79,80,89].

Even when hardware and computational power is growing at an ever accelerating rate, computational science seems to be lagging behind [28,89]. Among some of the alleged reasons for the aforementioned mismatch in scientific software development, the complexity of the domain can be highlighted [79,80]. A second factor could be the low value scientists ascribe to software developing knowledge and skills [8,20,79]. Third, lack of software engineering training and the horizontal quality of teaching, where knowledge is passed down from scientist to scientist, both actors having the same level of programming competence [8,89].

Also, scientists are somewhat reluctant to use and apply modern programming tools such as integrated development environments (IDE) [8,19,20,89]. In spite of the existing mismatch between software engineering and scientific computing, the programs created by scientists tend to be successful [69,78].

2.1 Related works

The idea of applying automated transformation on existing source code to introduce changes on it has been studied by software maintenance for years. It would be naive to think that each new version of an existing program is built from the very beginning, as a result it can be thought that "software development is program transformation" [42]. Many development processes have been conceived to build software from nothing, such as: the spiral software development process, the cascade model, the rational unified process, among others. Under that premise, it is only logical to assume that a software system is built from scratch. On the other hand, applying changes to an existent program is called Software maintenance. The IEEE defines maintenance as "Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment" [40]. Taking into account this definition, software maintenance or software re-engineering seems to be a especial case and has to be treated as an exception within the development process [42]. In contrast with this declaration, it is widely studied that the maintenance stage is the most resource consuming stage in a project, so it can be considered that building software from the very start is an exceptional case of software development and modifying preexistent software is the norm [42].

With its onset in the 1970's, the maintenance stage began to acquire relevance within the software development process. This stage has been divided into three different dimensions [82]. Toward 1972, an author described software maintenance as an Iceberg, this is the start of the idea that software maintenance stage has hidden features inside [18]. En 1976, E. Burton Swanson described three software dimensions for maintenance [82]:

1. Adaptive maintenance: "*Maintenance performed in response to changes in data and processing environments may be termed adaptive maintenance. The timely anticipation of environmental change is necessary to insure effective performance of this type of maintenance.*"
2. Corrective maintenance: "*Maintenance performed in response to failures of the above types may be termed corrective maintenance. Especially where processing*

failures are concerned, a diagnosis of the causes of failure constitutes a significant portion of the task for this type of maintenance activity."

3. Perfective maintenance: "*Maintenance performed to eliminate processing inefficiencies, enhance performance, or improve maintainability may be termed perfective maintenance. Its aim is to make the program a more perfect design implementation. It is undertaken when "justified," i.e., when the improvements to be achieved outweigh the costs of making those improvements*."

A fourth dimension has been introduced in another research study, called Preventive Maintenance: "*the modification of a software product after delivery to detect and correct latent faults in the software product before they become operational faults*" [41]. For many years, researchers and the industry have focused on finding and studying possible ways to improve the software development process and its maintenance stage [51]. According to some research studies, software has been characterized by four essential properties: complexity, conformity, changeability and invisibility [16] as well as laws describing software evolution throughout time [50–52]. These properties directly affect the process of software maintenance. This stage in the development process has been taking relevance over the last years. Toward the year 1969, the calculated percentage of effort required in the maintenance step, within the software development process context, was between 40 and 60%. By 1978 [53], some authors made reference to those numbers published in another research article on software maintenance published by Rigs 10 years before [73]. In 1979 the relative cost of the maintenance stage according to [90] was near 67%, other research studies written in this decade hinted that this value would come close to 70% [12]. Toward 1981, the relative maintenance cost amounted to 50% in over 487 organizations, which positioned this cost between 50 and 75% according to [55],remaining within those levels according to [39,71] in the years 1988 and 1990, respectively. This trend was regarded as a gradually falling value as from the 1990's. Rather surprisingly, this tendency continued on its upward trend and amounted to 75% of the total relative cost [29]. It came to the point of reaching more than 90% of the total relative cost of project [24,32,35,61,77,88]. Although it is true that all these values have been calculated through different methods and techniques, it is also a fact that these numbers are also undeniable proof of how costly, complex and important the maintenance process is.

We have not found a complete development process identified, integrated, implemented, and tested on at least some example applications for scientific software evolution/enhancement. We do not claim to provide a closed and absolutely complete tool, but a proposal on a complete process with a proof of concept implementation, showing several important details, including an integration on an existing IDE (Photran [22,63]) on which it would be possible to test the whole process. Furthermore, we have selected an open source tool because we think the maintenance process can be highly enhanced by different developers, each one providing specific implementation details. Being the Fortran scientific software in production for decades, we think it is important to have a specific process for enhancing, updating, and including new optimizations and parallelization facilities in Fortran code.

2.1.1 Software maintenance numbers

One of the fundamental aspects on which this work is based are the numbers derived from software maintenance process. There are no instances of research work to estimate the exact number of lines of source code in existence nowadays. Some estimations are:

- G. Booch estimates that 1.000.000.000.000 of lines of source code new or modified were produced cumulatively between 1945 and 2005 with an annual increase of 35.000.000 of lines of source code. [14].
- Kontogiannis et al. [46] estimates the number of lines of source code at the beginning of year 2000 was 800.000.000.000 in the entire world.
- Lammel in [47] also estimates the number of lines to be 1.000.000.000.000. 30% of them written in COBOL (225 millions), 20% written in C/C++ (180 millions), 10% in Assembler (140 to 220 millions) and 40% written in other languages (280 millions).

The number of lines of source code in maintenance process was estimated to be 200.000.000.000 in 2001 by [32]. The software maintenance process cannot go unnoticed. The lack of information and up-to-date data on the software that is in maintenance process nowadays is unsettling.

2.2 Source code restructuring

Considering the cost and the complexity involved in the software maintenance stage, researchers have focused, throughout these last years, on the improvement of software development techniques and tools to be used in this process. In this research, the term “restructuring” has been tracked back to its first possible use as a concept. According to Arnold restructuring is defined as: “software restructuring is the modification of software to make the software easier to understand and to change, or less susceptible to error when future changes are made” [3]. A broader and more general definition for source code restructuring is the one given by Chikofsky [23] in his work “reverse engineering and design recovery: A taxonomy”: “restructuring is 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).” In the definition provided by Chikofsky, that the source code restructuring should be viewed as a transformation to be applied on the existent code to obtain another representation is made clear. In most cases, this representation is better than its preceding one in that it preserves both the abstraction level and the external behavior. According to Chikofsky, the term restructuring covers a wide spectrum of transformations that do not only serve as source code but they can also be used to transform data models, requirement structures, and design blueprints. For instance, data normalization process is, to this author, an exemplar of data transformation that implies improvements on the logical data model [23].

Restructuring comes into being as a necessary process behind the implementation of software maintenance due to its most essential characteristics (complexity, conformity, changeability, invisibility) [16] and with the purpose of reducing the costs stemming from maintenance. At the same time, the inherent complexity of software is exacerbated even further when the already existent software must undergo changes of any of the following kind: corrective, adaptive,perfective or preventive. This creates a vicious circle that turns software increasingly complex to be handled. Bearing in mind the aforementioned principles, restructuring has the objective, according to certain viewpoints, of lessening software complexity through the implementation of incremental improvements on its internal structure [3,36].

There exists a particular case in bibliography that refers to systems built under the object-oriented programming paradigm which is also labeled as Refactoring. This label is nothing but the same concept with a different name. Source code refactoring is basically the object-oriented variant of restructuring [59] whose definition according to Martin Fowler is: ‘the process of changing a [object-oriented] software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure” [33]. A problem that is typically associated with source code restructuring is the application of such restructurings manually. This manually applied process tends to be costly and error-prone [36], as a result the application of automated tools to carry out these source code transformations becomes paramount.

2.2.1 Restructuring or refactoring Fortran source code

One of the most salient features to Fortran is its longevity, its beginnings and the first source code restructurings studies date back to the late 1960’s and early 1970’s. Throughout its first stages there was an attempt to structure and restructure specifically for Fortan source code called “spaghetti code” [5,25,31,34,38]. Over the last few years, there have existed many research studies aimed at automatically restructuring Fortran sequential source code by applying automated tools to fulfill the objective of parallelizing it [7,27,27,49,70,83–85]. Other research studies have conducted Fortran source code restructuring [65–68] by implementing such transformations as source code refactorings. The approach proposed in this work will take advantage of AST (Abstract Syntax Tree) transformations as implemented in previous works.

3 The change-driven development process

While there are several proposals (some of them unrelated to each other and with different objectives) and implementations for Fortran source code transformations and there is not a complete process, we have decided to propose this one. Actually, we have worked for several years specifically on Fortran source code transformations for high performance computing [84] and this proposal can be considered as resulting from that experience and the related work mentioned above.

The approach, whose spirit entails accepting and embracing change, is the one at use within the so-called light-weight or Agile Methodologies. Almost all of them ar based on the “Manifesto for Agile Software Development” [10]. Agile methodologies have been gaining momentum in the past two decades, such as XP, Scrum, Test-Driven Development (TDD), to name but a few. Bearing in mind that changeability and the continuous pressure to change are two essential features of software [16] and the maintenance process is the most resources-consuming stage within, the need to define a change-driven software development process become a must. There exist many development processes; yet, none of them is specifically based on the essential feature of software: change for its own sake.

3.1 From the unknown to the well known

One of the initial tasks of the Fortran legacy software maintenance is focused on understanding the software that is being dealt with. Although this assertion appears to be trivial, it can become fundamental at the time that source code written by third parties comes into play. This comprehension process becomes even more complex when one is working with programs that are 20 or 30 years old in that even the way in which source code was written may be difficult to understand. Therefore, finding a process that comes to exist out of the unknown is fundamental. The unknown, in this case, is understood as Legacy Source code written by others. Such process should leave programmers with a well known, comprehensible, readable and up-to-date source code. It is sometimes troublesome to extract knowledge out of source code written by another programmer easily as this code may be cryptic and intricate, among others. As has been discussed above, new software development stemming from already existent one is more widespread than the one started from scratch. In fact, building libraries and components that can not be re-utilized would not make sense. Bearing in mind that changeability and the continuous pressure to change are two essential features of software [16] and the maintenance process is the most resources-consuming stage, the need to define a change-driven software development [56] process becomes a must. This approach, whose spirit entails accepting and embracing change, is the one at use within the so-called light-weight methodologies. Almost all of them ar based on the "Manifesto for Agile Software Development" which claims the following notions [10] This vision toward software construction is considered to be an approach driven by change as they run counter to the ones utilized by classic methodologies which are driven by a plan. An important aspect in the light of this vision to be borne in mind is that change should be viewed as an essential software feature [16]. Along the same lines, the project development process is centered on two of the four essential features of software [16]. To this end, this process should be guided by changes or transformations to be applied on software. It can be stated then that a change-driven development process is synonymous with a "Transformation-Driven Development." Although these two concepts seem to bear a high degree of resemblance throughout this research study, this process will be referred to as "change-driven development" or CDD.

3.2 Change-driven development process

A change-driven development Process could be thought of as an Agile Meth/-od/-ol/-ogy when it comes to maintenance and scientific software development-related purposes. There is a high likelihood that this process may be extended to any kind of software. It is characterized by:

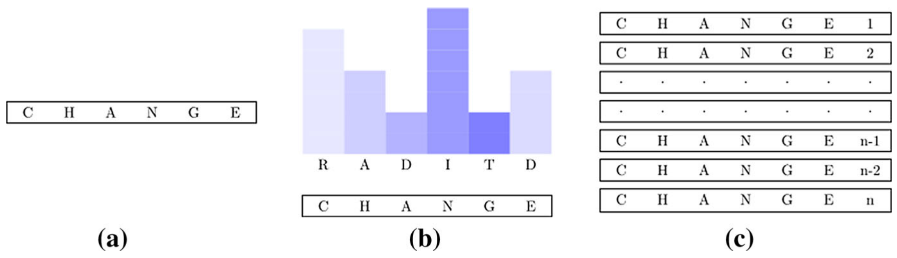


Fig. 1 Change as a working unit. **a** Unit of work, **b** core development activities and **c** list of changes

1. *It is focused on change* Given that change is one of the essential properties of software [16] CDD possesses change as a minimal unit of work. One change is to belong to one of these four characterizations: corrective change, adaptive change, perfective change, preventive change. Thus, a single change will be the unit of work in which several software development core activities will be performed: requirement, analysis, design, implementation, testing, and deployment, as shown in Fig. 1, where each one is identified by its initial letter (RADITD).
2. *It is tool centered* In this development process, tools are considered as important as change itself. It is almost inconceivable to believe that nowadays software can be constructed without making use of some kind of development tool. From this perspective, tools are conceived of as a means to apply the process itself. Programmers and other members of development teams must rely on such tools for the purposes of identification, comprehension, transformation and software verification.
3. *It is an iterative and incremental process* The concept of iterative and incremental process came into being in the 1930's and 1940's out of the initial work conducted by Walter Shewhart in the Bell Labs. The spirit of this concept involves performing small cycles of "Plan-Do-Study-Act" [48]. This concept was later on revisited in 1940's by a man who is considered the father of Quality W. Edwards Demings [48], who put forward his famous cycle "Plan-Do-Check-Act." Strangely enough both author are considered the fathers of the modern concept of quality. Broadly speaking, the spirit of this Iterative and Incremental Process resides in the building of a problem solving skeleton and in the fulfillment of small work cycles (called iterations). Out of performing these tasks, the solution to the problem will arise out of the application of the process in its entirety [9].

Taking into account the features that a change-driven development process must have, a first description of such process will be thoroughly described. Let us contemplate in this approach that a change is considered to be any variation in the state of any artifact that is involved in the software development process. The concept or vision of change or transformation is applicable to more artifacts other than source code. This process can be performed either from an already existent list of changes or a special iteration can be produced to obtain such list. As a following step, starting from a change to be applied to the system, this process comprises four stagese: comprehension, transformation, verification, and feedback. Once these four stages have been covered, a change will be applied into software. For this purpose, the central

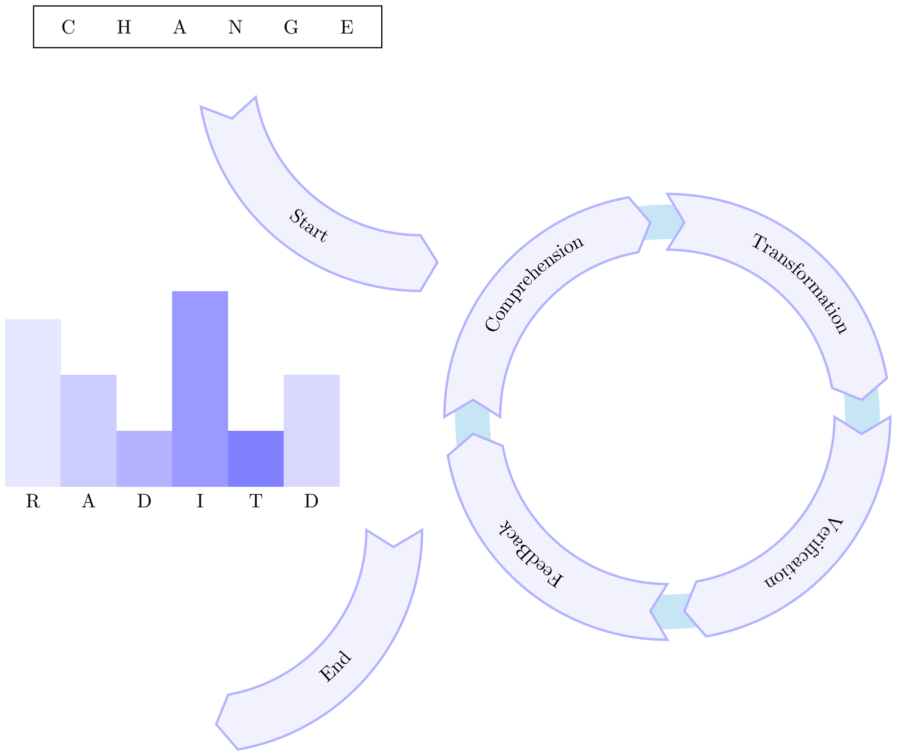


Fig. 2 A complete process cycle

core activities of software development (requirements, analysis, design, implementation, tests, and deployment) will be performed with different intensity in keeping with this specific change; undoubtedly some other supporting activities will be performed (documentation, management, software quality assurance, among others). At the end of such process, the change will be introduced into the system, as shown in Fig. 2.

Within the four stages of the change-driven development software process, a specific work flow has been defined. The work flow is viewed as the study of the operational aspects of an activity in which some facets are focused on, such as how tasks are structured, how tasks are performed, the sequence to be performed should be, how they are synchronized. In this case, when working with scientific software, the defined work flow comprises the following steps:

1. The establishment of an initial version of source code.
2. The transformation source code.
3. The verification of the obtained source code.
4. The validation of numerical results.
5. The acceptance/rejection of the applied change based on numerical results.
6. The documentation.

One fundamental feature of this process, as described above, entails the development tools to be utilized. It is of utmost importance that the biggest number of development tools available to be used are integrated in the same application. In other words to perform an operation on the source code it may not be necessary to change the current tool. For example, if it is necessary to run a source code analyzer, such tool should be integrated in the same program in which the programmer is editing. This assertion could seem trivial or insignificant but it is not unimportant. In the tool review performed in previous studies, most of these tools were standalone or independent programs. One distinctive feature of this kind of tools can be found in their complexity. To perform such type of tasks using the search and replace technique may not be fruitful.

4 The four stages

In the development process called change-driven development proposed in this research work, four stages are clearly defined: (1) comprehension, (2) transformation, (3) verification, and (4) feedback.

4.1 Comprehension stage

This stage is intended to understand and comprehend the source code which is being dealt with through the application of different development tools to such end. Some examples about the type of information that these tools should provide are: symbol table, static call tree, complex metrics (the longest, the most called, the most used routine), code coverage, among others. In the case of Fortran, more specific information about the programming language could be: common blocks areas, obsolete language features utilization, number of entry points within a routine, number and type of branch of GOTO statement (conditional forward branch, conditional backward branch, unconditional branch, among others). A list of changes to be applied to the source code will be obtained as a result of this stage.

4.2 Transformation stage

In this stage, the application of source code transformations detailed in the list obtained in the previous step that serves as a tool to improve and update such program will be performed. These changes will be introduced through automated source code restructuring tools that should be integrated and provided by the development environment. In the same way as within test-driven development, tests become paramount; within change-driven development, tools also become of utmost importance. In this stage, the required test will be built in order to verify the changes.

4.3 Verification stage

In this stage of the process, the applied changes must be verified in order to ensure the their correct application. It is important to highlight that the description of the process, activities and tasks belonging to this stage are by themselves an entire research area that exceeds the scope of this work. Regarding scientific software, verification involves a numerical validation task of the program results.

4.4 Feedback stage

This is the final stage (that every process should have) in which all the information regarding the process should be collected, processed and used to provide feedback to the next cycle of the process. This stage is also known as lessons learned or postmortem analysis. Basically, metrics and relevant information that can contribute to improve the process in the next iteration are collected. Additionally, in this stage, the list of changes can be modified by adding other changes to be performed in the next iterations.

4.5 The work flow

In the following subsection, a thorough description for the steps proposed for the work flow to be applied by change-driven development for scientific software will be presented.

4.5.1 The establishment of an initial version of source code

In this stage, a clearly outlined point of departure will be settled to apply the defined change. When working with preexistent source code, its current version will be fixed as the starting point. In fact, this is an instance of handling of legacy code. If there is no source code to work on, elicitation requirements and design techniques will be applied in order to obtain an initial list of changes to be performed. It must be taken into account that the starting point of the process is the absence of any kind of software and the change to be applied would be the analysis and design of the requirements to fulfill the program functionality.

4.5.2 The transformation source code

As a first task of this step, tests will be defined in order to verify that the source code transformation will be successfully carried out. The change defined in the previous step should be taken as a starting point for the source code transformation to be performed. Given the absence of source code, the design will be transformed into implementation which will leade to the creation of the new source code. Such transformations will be applied by using automated restructuring source code tools integrated within the development environment for the former scenario, and for the latter (absence of source code), the integrated development environment text editor will be used to implement new source code.

4.5.3 The verification of the obtained source code

All the tests that have been built in the previous stage in order to verify the source code transformed functionality must fulfill the initial requirements. Should any of these tests fail in this stage, the previous stage must be performed again.

4.5.4 The validation of numerical results

In this step, scientific software should require a numerical validation. This validation can be optional and it aims to contrast the obtained results with the expected ones. Although this task could be performed by applying tests, a specific numeric validation stage may also be required.

4.5.5 The acceptance/rejection of the applied change based on numerical results

Once the last two steps have yielded favorable results, they will be compared with the criteria set in order to accept or reject the new product version.

5 First case study

In this section, an example written in FORTRAN extracted from the book *Numerical mathematics and computing* [21] will be utilized. One relevant aspect of this example lies in the fact that in a latter edition of the book, from 2003 [21], the author made changes to his source code examples and updated them to a Fortran 90 version. In that way, those transformations or changes applied by the author can be used as contrast with the ones made by the proposed process in this research. The FORTRAN77 version of the program evaluates a function derivative at a certain point. This source code can be seen in Fig. 3, and the list of changes to be applied to the following program comprises: (1) Produce free format source code, (2) use lower-case Fortran statements, (3) use lower-case identifiers, (4) introduce implicit none, (5) replace old style DO loops, (6) remove unreferenced labels, (7) remove unnessesary statements, (8) add identifier to end program, (9) generate variable/s from DATA in the main program, and (10) generate parameter/s from DATA.

```
      PROGRAM FIRST
C NUMERICAL MATHEMATICS AND COMPUTING, CHENEY/KINCAID, (c) 1985
C
C FILE: first.f
C
C FIRST PROGRAMMING EXPERIMENT
C
      DATA N/25/, H/1.0/, X/0.5/
      F = SIN(X)
      G = COS(X)
      DO 2 I = 1,N
      H = 0.25*H
      D = SIN(X + H) - F
      Q = D/H
      E = ABS(G - Q)
      PRINT *,H,D,Q,E
 2    CONTINUE
      STOP
      END
```

Fig. 3 FIRST.f source code

```
      PROGRAM FIRST
!
! NUMERICAL MATHEMATICS AND COMPUTING, CHENEY/KINCAID, (c) 1985
!
! FILE: first.f
!
! FIRST PROGRAMMING EXPERIMENT
!
    ... identical sequence of lines as initial version
```

Fig. 4 Change 1

5.1 Process application

Taking into account the ten-changes list outlined above, an iterative process on of the process for each change of the list will be carried out. A well-known version control system tool, git, will be used, along with the Eclipse plug-in also known as Egit (http://eclipse.org/egit/).

5.1.1 Change #1: Produce free format source code

The initial version of the source code is that of Fig. 3 and Once the initial version has been set, the run output results of this version will be generated. After these steps have been fulfilled, the commit of the initial version of the source code for the first change/process iteration along with the results of such execution will be performed.

The Change to Free Format to be applied is closely connected to one characteristic of Fortran that dates back to its very origins [2,4], which renders the source code hard to read and understand. An additional important aspect is that fixed format allows for spacing through statements or through a variable name, for instance "D O100I = 1, 10" instead of "DO 100 I = 1, 10." This language feature played its part, for instance, in the failure of the space probe MARINER I. Even when this source code transformation has been referred to as a "pretty printing" one, the previous statement indicates it is not necessarilly the case.

The Transformation has been implemented as a Photran refactoring called *"Change to free format"*. The approach at use can parse the instruction of the language correctly as well as conduct pre and postvalidations to ensure behavior preservation. Such source code transformation has been implemented throughout this research study. It has been included in the IDE tool menu option. A view with differences is displayed in order to preview source code before and after changes. Along with this transformation, the comment characters in keeping with the free format of Fortran 90 standard have also been replaced. Actually, in this particular example, the new version of the program only has different character comment lines as can be seen in Fig. 4.

The following step entails the verification and validations of the results. In this case and due to the simplicity of the source code, it has not been necessary to write any type of unit test. If any unit test has been written, it must be executed in order to ensure that such change has not had an undesirable impact on software. The results of the program execution having undergone a transformation/change are compared with their prior version once the program execution results have been yielded. The numerical results

Fig. 5 Change 2

```
      program FIRST
      ...
      data  N/25/, H/1.0/, X/0.5/
      ...
      do 2 I = 1,N
      ...
      print *,H,D,Q,E
2     continue
      stop
      end
```

have not been altered after the transformation. The first iteration of the process has been performed satisfactorily and, thus, it is committed.

5.1.2 Change #2: Use lower-case Fortran statements

The initial version of the source code as well as the numerical results have been produced in the previous change, so they are directly available. This source code transformation is needed because in older versions of Fortran, programmers had to use only upper-case letters in their code. Such condition was removed from the Fortran standards at least since Fortran 90 standard [1]. Furthermore, Fortran language does not distinguish between reserved words and others identifiers. Programmers can find variables whose names are the same as a language statements, such transformation can not be performed by using a pattern matching or search and replace techniques. This source code transformation has been implemented as a Photran refactoring by traversing the program AST and by changing only those identifiers that fulfill the role of a Fortran statement from upper-case to lower-case, as schematically shown in Fig. 5 (take into account the initial version of Fig. 3).

The numerical results are identical to the previous version, thus the change is accepted and committed. Actually, this new source code version will be the initial version for the next change, and this situation will be repeated for every successive change/process iteration.

5.1.3 Change #3: Use lower-case identifiers

In this case, like in the previous iteration, the AST approach must be used to determine which names are identifiers and which names are program statements or literal expressions. The source code transformation has been implemented in this research work and contributed in the official Photran 8.1 distribution. Once the change has been established and performed, the source code is ready to be compiled and the program must be run in order to validate and verify the obtained numerical results. If a set of unit tests exist, they must be run in this step. The numerical results obtained have not shown any difference from the previous ones after the change has been applied. A little bug that consists in variables within a block in the DATA statement has not been modified in such transformation. In order to continue with the process, the bug was reported and the remaining change was performed manually within the DATA block, as shown in Fig. 6.

Before Change

```
      ...
      program FIRST
      data  N/25/, H/1.0/, X/0.5/
      F = SIN(X)
      G = COS(X)
      do 2 I = 1,N
      H = 0.25*H
      D = SIN(X + H) - F
      Q = D/H
      E = ABS(G - Q)
      print *,H,D,Q,E
2     continue
      stop
      end
```

After Change

```
      ...
      program FIRST
      data  n/25/, h/1.0/, x/0.5/
      f = SIN(x)
      g = COS(x)
      do 2 i = 1,n
      h = 0.25*h
      d = SIN(x + h) - f
      q = d/h
      e = ABS(g - q)
      print *,h,d,q,e
2     continue
      stop
      end
```

Fig. 6 Change 3

5.1.4 Change #4: Introduce implicit none

Implicit declaration of variables has been allowed since the first Fortran standard, and it implies that a variable name serves as an identifier and at the same time as a type definition: "The name employed to identify a datum or function carries the data type association." The implicit variable declaration carries a set of drawbacks such as bugs introduced by: typing errors, wrong data type assignment, and values and range control, among others. Since it is desirable a programming approach not allowing implicit declaration, the Fortran language has defined the implicit none statement, which forces programmers to declare all variables specifying their data type. The source code transformation will take advantage of the Photran VPG (Virtual Program Graph), which contains the symbol table. The symbol table has the necessary information to determine the data type, whether a token is an identifier or not, and finally whether such token has been implicitly or explicitly declared in order to modify the AST to be compliant with the data explicit declaration.

Once the source code was applied, and as a result of the feedback stage, it was noticed that variables were declared one per line. It may be interesting to introduce a new automated source code transformation which allows programmers to group or shrink a set of variable declarations split into different lines into a single line. Given that this new source code transformation does not exist and is not implemented, such shrinking will be manually performed. The resulting source code after introducing implicit none by declaring variables is shown in Fig. 7.

The change was successfully tested because it produced the same output of the previous version (which is the same as the original version, actually) and, thus, it is accepted and commited as part of the process.

5.1.5 Change #5: Replace old style DO loops

One of the most interesting statements from the scientific computing view point is precisely the DO statement. Basically, this statement usually performs more than the

Before Change

```
    program FIRST
    data  n/25/, h/1.0/, x/0.5/
    ...
```

After Change

```
program FIRST
    implicit none
    real:: d,e,f,g,h,q,x
    integer :: i,n
    data  n/25/, h/1.0/, x/0.5/
    ...
```

Fig. 7 Change 4

Before Change

```
        ...
        do 2 i = 1,n
        h = 0.25*h
        d = SIN(x + h) - f
        q = d/h
        e = ABS(g - q)
        print *,h,d,q,e
2       continue
        stop
        end
```

After Change

```
     ...
    do i = 1,n
        h = 0.25*h
        d = SIN(x + h) - f
        q = d/h
        e = ABS(g - q)
        print *,h,d,q,e
2       continue
    end do
    stop
    end
```

Fig. 8 Change 5

90% of the computing in a scientific application.There are many different syntactical ways to write the same Fortran loop, such as

```
....                      ....                 ....
DO 110 I=1,10             DO 100 I=1,10        DO I=1,10
DO 100 J=1,10             DO 100 J=1,10        DO J=1,10
MATRIX(I,J)=0         100 MATRIX(I,J)=0        MATRIX(I,J)=0
100 CONTINUE              ....                 END DO
110 CONTINUE              ....                 END DO
....                      ....                 ....
(A)                       (B)                  (C)
```

Versions (A) and (B) preceed the structuring programming concept, especially the (B) form, that is called Shared Do loop termination, and has been marked as obsolete in the Appendix B of the Fortran 90 standard. One of the most desirable features that source code should possess is that it must be easy to understand and easy to read. This fact is especially important when working with source code to be parallelized due to the fact that such parallelization is still performed (mostly) manually. In this research work, an automated source code transformation that changes old Fortran style Do loops into a more updated and structured format has been implemented [57]. This automated source code transformation has been part of the Photran 6.0 distribution. The source code transformation is schematically shown in Fig. 8.

Note that the code inside the Do loop is indented, so it becomes even more readable. It is worth mentioning that the continue statement could be deleted in this stage, but given that there are many different ways of using it in the context of Do loops (such

Fig. 9 Change 6

Before Change

```
       ...
       do  i  =  1,n
              ...
2             continue
       end  do
       ...
```

After Change

```
       ...
       do  i  =  1,n
              ...
              continue
       end  do
       ...
```

as in shared Do loop terminations) it has been decided to leave the continue to be analyzed by another source code transformation: Remove Unnecessary Statements. Old style Do loops are replaced successfully, since the program still generates the same output, so this new version is committed and taken as the initial version in the next change/iteration.

5.1.6 Change #6: Remove unreferenced labels

As a result of source code transformations some labels are not referenced anymore. This is the case specifically after replacing old style Do loops have been removed: the labels in continue statements or those in the statement with shared Do loop termination are probably not referenced anymore. Removing unreferenced labels has been implemented in this research work as part Photran, and the result of the automated source code transformation can be schematically seen in Fig. 9.

It has to be taken into account that every step toward removing labels is also a step to avoid spaghetti code, since goto statements necessarilly need labels. Even when in this simple case the analysis is straightforward, labels are hard to maintain/keep track in many lines of code. The source code change has been verified as correct by running the program and comparing the output with the original version.

5.1.7 Change #7: Remove unnecessary statements

In this stage, statements like continue and the final stop will be removed from source code due to the fact that none of them provides functionality to the program. More specifically, the continue statement is useles after replacing old style Do loops, where they become equivalent to the assembly language instruction *No Operation*. The final stop statement has been used in the oldest fortran standards as the last executable program instruction before the end of the program due to the fact that the end statement was not and executable statement until Fortran 90. This automated source code transformation has been identified trough the comprehension stage, and it is a good candidate to be implemented in order to contribute to the tool. Due to the fact that this source code transformation is not implemented, such transformation has been performed manually by following the work flow proposed for change-driven development. The resulting source code can be schematically seen in Fig. 10.

As expected, this source code transformation has not introduced any change in the program behavior and, thus, the change is committed.

Fig. 10 Change 7

Before Change

```
program FIRST
...
do i = 1,n
     ...
     continue
end do
stop
end
```

After Change

```
program FIRST
...
do i = 1,n
     ...
end do
end
```

Fig. 11 Change 8

Before Change

```
program FIRST
implicit none
...
end
```

After Change

```
program FIRST
  implicit none
  ...
end program FIRST
```

5.1.8 Change #8: Add identifier to end program

As explained above, the end statement has not been considered an executable statement until Fortran 90 standard. In addition, there did not exist any of these combinations: end program, end function, and end subroutine. Adding the identifier to the end of program helps in readability as well as being an initial step to add identifiers to the end of functions and subroutines. The resulting source code is shown in Fig. 11, where an indentation has been manually added.

5.1.9 Change #9: Generate variable/s from DATA in the main program

When a variable is declared within a DATA block, it implies that such variable has the SAVE attribute except if the declaration of this variable is performed within a named COMMON BLOCK. Variables with the SAVE attribute are equivalent to those of the C language with the *static* attribute. The program variables *n* and *h* of the example do not have any clear reason to be defined with the SAVE attribute, and they could be initialized in an assignment. Taking into account that explicit variable declarations have been made in Change #4: Introduce implicit none, there is only one more check to be made: if the variables are assigned in the program, then they should be initialized in the code, otherwise they are Fortran parameters. The resulting the source code transformation, applied manually, can be seen in Fig. 12.

Even when this change has been applied manually, the implementation is rather simple, since as explained in Change #4: Introduce implicit none, the VPG and more specifically its symbol table can be used in order to identify how the variables are used.

Before Change

```
program FIRST
     ...
     real:: d,e,f,g,h,q,x
     integer :: i,n
     data  n/25/, h/1.0/, x/0.5/
     ...
end program FIRST
```

After Change

```
program FIRST
     ...
     real:: d,e,f,g,h,q,x
     integer :: i,n
     data  n /25/
     x=0.5
     h=1.0
     ...
end program FIRST
```

Fig. 12 Change 9

Before Change

```
program FIRST
     ...
     data  n /25/
     ...
end program FIRST
```

After Change

```
program FIRST
     ...
     parameter ( n = 25 )
     ...
end program FIRST
```

Fig. 13 Change 10

5.1.10 Change #10: Generate parameter/s from DATA

The purpose of this source code transformation is to extract constant values from the DATA statements. In this case, the identifier *n* could be considered a constant/Fortran parameter. The Photran tool has already implemented and integrated this transformation, and the resulting source code can be seen in Fig. 13.

This last source code change was also successful: it did not introduce any program output change and, thus, it is accepted and committed.

5.2 Results

The initial program was entirely written in FORTRAN 77, and thanks to the application of change-driven development it was possible to generate an up-to-date Fortran 90 version (see Fig. 14, where source code comments have not been included in order to make easier the visual comparison). This process was conducted almost entirely through the application of automated source code transformation integrated in its entirety within a development tool implemented as refactorings. This process has been conducted by 10 iterations out of which 7 are fully implemented and the other 3 are *naturally* derived from the CDD utilization.

The remarkable aspect about this source code example is the fact that a new edition of the book was released in 2003 by the authors [21] in which exactly the same example was presented, but in this edition the authors changed it to Fortran 90. The Fortran 90 version provided by the authors in the new edition of the book is almost the same as that in Fig. 14, which was generated as a sequence of CDD iterations/source code changes.

FORTRAN 77

```
      PROGRAM FIRST
      DATA  N/25/, H/1.0/, X/0.5/
      F = SIN(X)
      G = COS(X)
      DO 2 I = 1,N
      H = 0.25*H
      D = SIN(X + H) - F
      Q = D/H
      E = ABS(G - Q)
      PRINT *,H,D,Q,E
2     CONTINUE
      STOP
      END
```

Fortran 90 obtained by CDD

```
program FIRST
    implicit none
    real:: d,e,f,g,h,q,x
    integer :: i,n
    parameter ( n = 25 )
    x=0.5
    h=1.0
    f = SIN(x)
    g = COS(x)
    do i = 1,n
        h = 0.25*h
        d = SIN(x + h) - f
        q = d/h
        e = ABS(g - q)
        print *,h,d,q,e
    end do
end program FIRST
```

Fig. 14 Initial version versus resulting version

6 Second case study, including parallelization

This case study is based on a program which estimates an integral by using an averaging technique. The function to be integrated is:

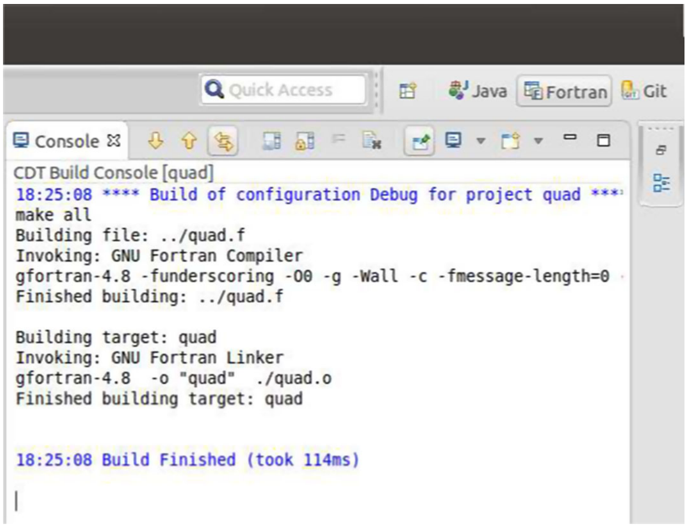
$$f(x) = \frac{50}{(pi * (2500 * x^2 + 1))}$$

from http://people.sc.fsu.edu/~jburkardt/f77_src/quad_serial/quad_serial.html.

Given that the focus is on parallelization with OpenMP [64], it has been decided to collect performance data of the initial version, so that it will be later available for performance analysis comparison. The performance data will be part of the data to be collected and used at least in the verification CDD phase. Thus, the next section will explain several simple steps to have an IDE project with this (legacy) code, for establishing an initial version with which performance data as well as numerical results will be collected.

6.1 Program install and profile

The program to be used can be considered as legacy code because it fits with the features described in the Legacy code definitions. Once source code has been downloaded, the Photran IDE along with PhotranLint (developed in this research work), will be the tool to carry out every step of each CDD iteration. Git has been used as control version software by using the eGit plug-in. Actually, the same environment used in the previous example/case study. A Fortran project is created within the Photran IDE: source code is loaded into the project. Once all the program files are loaded, some required compiler and linker options are set, as well as setting the version control in the IDE. Once all these steps were performed the program will be compiled for

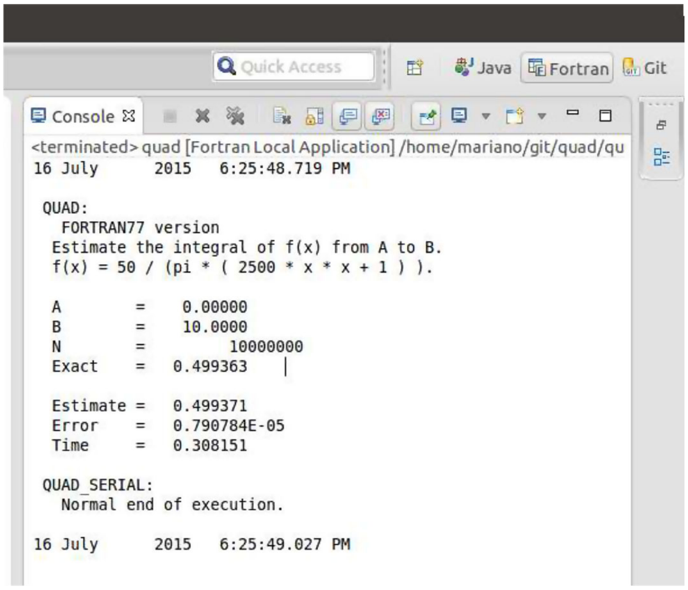


```
CDT Build Console [quad]
18:25:08 **** Build of configuration Debug for project quad ***
make all
Building file: ../quad.f
Invoking: GNU Fortran Compiler
gfortran-4.8 -funderscoring -O0 -g -Wall -c -fmessage-length=0
Finished building: ../quad.f

Building target: quad
Invoking: GNU Fortran Linker
gfortran-4.8  -o "quad"  ./quad.o
Finished building target: quad


18:25:08 Build Finished (took 114ms)
```

Fig. 15 Compilation results of Quad.f program



```
<terminated> quad [Fortran Local Application] /home/mariano/git/quad/qu
16 July      2015   6:25:48.719 PM

 QUAD:
   FORTRAN77 version
  Estimate the integral of f(x) from A to B.
  f(x) = 50 / (pi * ( 2500 * x * x + 1 ) ).

  A        =    0.00000
  B        =    10.0000
  N        =         10000000
  Exact    =   0.499363

  Estimate =   0.499371
  Error    =   0.790784E-05
  Time     =   0.308151

 QUAD_SERIAL:
   Normal end of execution.

16 July      2015   6:25:49.027 PM
```

Fig. 16 Execution program results

the first time. Figure 15 shows the IDE summary for the project compilation. Then, the execution step is performed in order to obtain the program results as well as the execution profile. The program output is shown in Fig. 16, which does not seem to have any runtime/numerical error.

As a matter of fact, this example has been selected because the programmer provided the original output results. In general, comparing program outputs can be made with

```
Original Execution                        Initial IDE execution
──────────────────────────────            ──────────────────────────────

14 December  2011   8:28:12.640 AM        16 July      2015   6:25:48.719 PM

QUAD:                                     QUAD:
FORTRAN77 version                         FORTRAN77 version
Estimate the integral of f(x) from A to B. Estimate the integral of f(x) from A to B.
f(x) = 50 / (pi * ( 2500 * x * x + 1 ) ). f(x) = 50 / (pi * ( 2500 * x * x + 1 ) ).

A        =    0.00000                     A        =    0.00000
B        =    10.0000                     B        =    10.0000
N        =    10000000                    N        =    10000000
Exact    =    0.499363                    Exact    =    0.499363

Estimate =   0.499371                     Estimate =   0.499371
Error    =   0.790784E-05                 Error    =   0.790784E-05
Time     =   0.338820                     Time     =   0.308151

QUAD_SERIAL:                              QUAD_SERIAL:
Normal end of execution.                  Normal end of execution.

14 December  2011   8:28:12.980 AM        16 July      2015   6:25:49.027 PM
```

Fig. 17 Output comparison

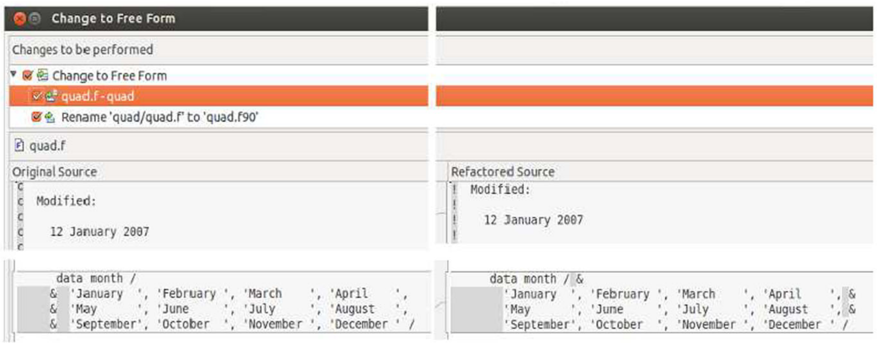
```
index % time    self  children    called       name
<spontaneous>
[1]      92.3    0.07    0.17                  MAIN__ [1]
0.17     0.00 10000000/10000000     timestamp_ [2]
0.00     0.00       1/2           register_tm_clones [5]
-----------------------------------------------
0.17     0.00 10000000/10000000     MAIN__ [1]
[2]      65.4    0.17    0.00 10000000         timestamp_ [2]
-----------------------------------------------
0.02     0.00       1/1           main [4]
[3]       7.7    0.02    0.00       1          f_ [3]
0.00     0.00       1/2           register_tm_clones [5]
-----------------------------------------------
<spontaneous>
[4]       7.7    0.00    0.02                  main [4]
0.02     0.00       1/1           f_ [3]
-----------------------------------------------
0.00     0.00       1/2           f_ [3]
0.00     0.00       1/2           MAIN__ [1]
[5]       0.0    0.00    0.00       2          register_tm_clones [5]
-----------------------------------------------
```

Fig. 18 Profile data

small specific programs or even with standard ones (e.g., diff). However, the program output of this example is simple as well as text-only, so that it is possible a quick visual comparison, as in Fig. 17. It is worth to mention that every comparison can be made also inside the IDE, which automatically highlights differences in case they are found.

Figure 18 shows the runtime profile data, which is always easy to obtain via specific compiler switch/es and widely available programs such as gprof.

Only two source code changes will be made in this case: (1) produce free format source code, in order to enhance program readability; and (2) Do Loop parallelization with OpenMP [64], in order to show parallelization capabilities as part of the CDD already implemented in the IDE.



Original Source:

```
c
c  Modified:
c
c    12 January 2007
c

      data month /
     & 'January  ', 'February ', 'March    ', 'April    ',
     & 'May      ', 'June     ', 'July     ', 'August   ',
     & 'September', 'October  ', 'November ', 'December ' /
```

Refactored Source:

```
!  Modified:
!
!    12 January 2007
!

      data month / &
       'January  ', 'February ', 'March    ', 'April    ', &
       'May      ', 'June     ', 'July     ', 'August   ', &
       'September', 'October  ', 'November ', 'December ' /
```

Fig. 19 Diff view, original code on the *left* and transformed code on the *right*

6.2 Change #1: Produce free format source code

The actual version of the source code is versioned by using the IDE plug-in, as in the previous case study. In this particular case, the profile data is considered part of the program output, which is not necessarily useful for this first change but for all changes related to performance evaluation. Figure 19 shows the resulting code, where different lines are highlighted by the IDE in order to aid the visual comparison. In this case study, two major changes have been made: (a) comment lines are changed to free format, i.e., preceding with the character! the source code comment, and (b) continuation lines are determined by adding the character & at the end of the line to be continued. As expected, the new version, in free format source code behaves as the original one.

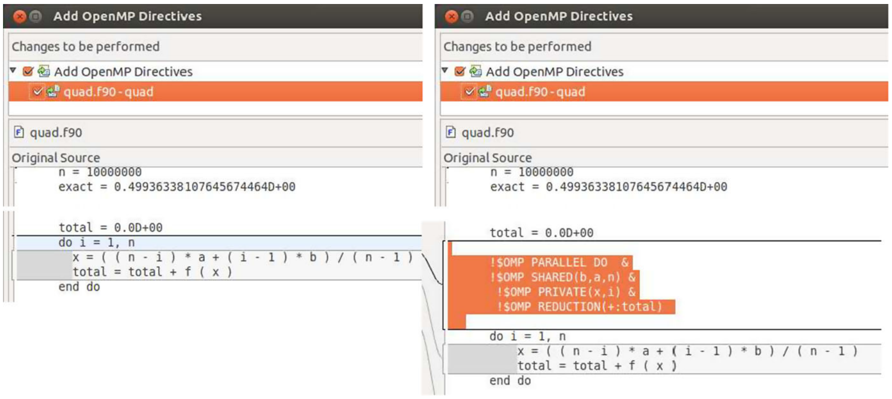
6.3 Change #2: Do loop parallelization with OpenMP

This change/iteration of the CDD starts with versioning the output source code from the previous change, as proposed in the CDD work flow. Given that this change is devoted to reduce runtime, the program profile (Fig. 18) is analyzed in order to find out the specific code to parallelize. Two very well-known general guidelines are taken into account in the analysis: (1) the profile provides the function in which to look for parallel processing, and (2) Do loops are identified in order to add the corresponding OpenMP directives. Taking into account those guidelines, the Do loop in Fig. 20 has been identified. The automated source code transformation to parallelize the Do loop has been implemented as "Add OpenMP Directives" in the IDE [83]. The implementation checks every requirement to parallelize the Do loop by performing a set of analysis on the source code using the AST. If the Do loop can be parallelized, the OpenMP directives are automatically included in the code and the results are shown to the programmer, as in Fig. 21.

More specifically, Fig. 22 shows the source code before and after the change, where it is possible to identify the OpenMP directives automatically inserted by the tool in capital letters. Currently, the tool is able to identify many data dependences as well

```
do i = 1, n
x = ( ( n − i ) ∗ a + ( i − 1 ) ∗ b ) / ( n − 1 )
total = total + f ( x )
end
```

Fig. 20 Do loop to be parallelized



```
n = 10000000
exact = 0.49936338107645674464D+00

total = 0.0D+00
do i = 1, n
   x = ( ( n - i ) * a + ( i - 1 ) * b ) / ( n - 1 )
   total = total + f ( x )
end do
```

```
n = 10000000
exact = 0.49936338107645674464D+00

total = 0.0D+00

!$OMP PARALLEL DO  &
!$OMP SHARED(b,a,n) &
 !$OMP PRIVATE(x,i) &
 !$OMP REDUCTION(+:total)

do i = 1, n
    x = ( ( n - i ) * a + ( i - 1 ) * b ) / ( n - 1 )
    total = total + f ( x )
end do
```

Fig. 21 Diff view for the Do loop parallelization with OpenMP

Code Before	Code After
`total = 0.0D+00`	`total = 0.0D+00`

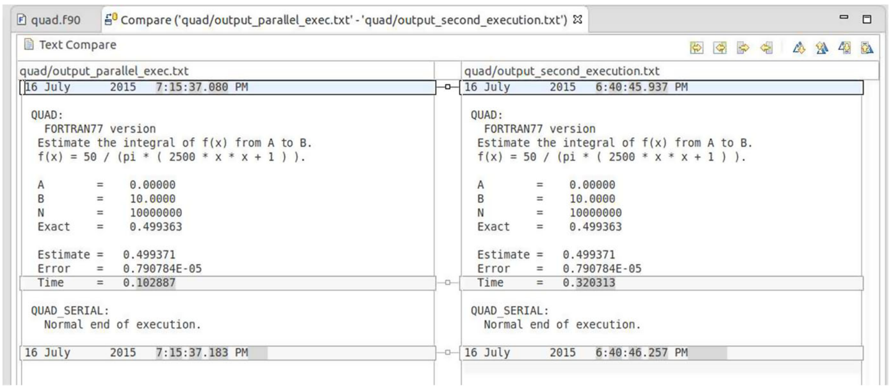
	`!$OMP PARALLEL DO &`
	`!$OMP SHARED(b,a,n) &`
	`!$OMP PRIVATE(x,i) &`
	`!$OMP REDUCTION(+:total)`

`do i = 1, n`
`x = ((n-i)*a+(i-1)*b)/(n-1)`
`total = total + f ( x )`
`end do`
	`do i = 1, n`
	`x = ((n-i)*a+(i-1)*b)/(n-1)`
	`total = total + f ( x )`
	`end do`
	`!$OMP END PARALLEL DO`

Fig. 22 Automated source code transformation: OpenMP Do loop parallelization

as some reduction variables. Once this source code change has been made and set the specific OpenMP compiler and linker options (in the IDE, as usual), the program will be able to run in parallel. This particular example was run in a dual processor computer, and Fig. 23 shows the diff view comparing sequential and parallel output runs.

The parallel runtime was 0.102887 s, while the sequential runtime was 0.320313 s, thus obtaining an improvement of about 33%, and the numerical results are the same. It is worth mentioning that the parallel version has been produced by the IDE by following a few and well-known and simple performance guidelines.


quad.f90
Compare ('quad/output_parallel_exec.txt' - 'quad/output_second_execution.txt')
Text Compare

quad/output_parallel_exec.txt

```
16 July      2015    7:15:37.080 PM

 QUAD:
   FORTRAN77 version
  Estimate the integral of f(x) from A to B.
  f(x) = 50 / (pi * ( 2500 * x * x + 1 ) ).

  A        =    0.00000
  B        =    10.0000
  N        =    10000000
  Exact    =    0.499363

  Estimate =   0.499371
  Error    =   0.790784E-05
  Time     =   0.102887

 QUAD_SERIAL:
   Normal end of execution.

16 July      2015    7:15:37.183 PM
```

quad/output_second_execution.txt

```
16 July      2015    6:40:45.937 PM

 QUAD:
   FORTRAN77 version
  Estimate the integral of f(x) from A to B.
  f(x) = 50 / (pi * ( 2500 * x * x + 1 ) ).

  A        =    0.00000
  B        =    10.0000
  N        =    10000000
  Exact    =    0.499363

  Estimate =   0.499371
  Error    =   0.790784E-05
  Time     =   0.320313

 QUAD_SERIAL:
   Normal end of execution.

16 July      2015    6:40:46.257 PM
```

Fig. 23 Diff view between serial and parallel results

7 Conclusion and further work

In this research work, change-driven development (CDD) has been presented as a new development process to maintain Scientific Software. This new development process has been adopted under the hypothesis that software development is based on transforming and changing programs. More specifically, CDD is funded on:

– Change as a unit of work.
– Integrated Development Tools.
– An iterative and incremental approach.

Even though CDD has been initially intended to be applied on already existent (legacy) software, building software form nothing is considered an exceptional occurrence in which the transformation consists in starting from scratch. Many of the current supercomputing software is actually Fortran legacy software, and this the main reason of the current proposal being focused on Fortran.

An entire cycle has been designed for CDD and a proposed work flow has been applied and performed on two study cases. A FORTRAN 77 program has been updated into a Fortran 90 version by applying CDD almost entirely by performing automated source code transformations, most of them implemented in the Photran IDE. It has been proven that making use of such transformations the source code has evolved into a more modern version of itself. Also, a serial program has been parallelized using CDD, improving its performance about 33% on a dual processor computer.

Further work includes adding more support to Fortran programmers as well as extending the application range of CDD to software other than scientific software. In the latter case, the proposed methodology could be improved so that it can be exploited by the software industry in general. Adding further case studies could provide new insights in the many details involved in legacy software. It is also essential to provide and gather information on the utilization of CCD by programmers, building and applying metrics along this development process. Experience should be contributed to Integrate CDD to other IDEs due to the fact that development tools are paramount to the whole proposed process.

References

1. American National Standards Institute and Computer and Business Equipment Manufacturers Association (1992) American National Standard for programming language, FORTRAN—extended: ANSI X3.198-1992: ISO/ IEC 1539: 1991 (E). American National Standards Institute
2. ANSI FORTRAN. X3. 9-1966. (1966) American National Standards Institute Incorporated, New York, p 40
3. Arnold RS (1989) Software restructuring. Proc IEEE 77(4):607–617
4. Backus J (1978) The history of Fortran I, II, and III. ACM SIGPLAN Not 13(8):165–180
5. Baker BS (1977) An algorithm for structuring flowgraphs. J ACM (JACM) 24(1):98–120
6. Balmer DW, Paul RJ (1986) Casm-the right environment for simulation. J Oper Res Soc 37:443–452
7. Basicevic I, Jovanovic S, Drapsin B, Popovic M, Vrtunski V (2009) An approach to parallelization of legacy software ECBS-EERC '09 Proceedings of the 2009 First IEEE Eastern European Conference on the Engineering of Computer Based Systems, Novi Sad, Serbia, pp 42–48. ISBN: 978-0-7695-3759-7
8. Basili VR, Cruzes D, Carver JC, Hochstein LM, Hollingsworth JK, Zelkowitz MV, Shull F (2008) Understanding the high-performance-computing community
9. Basili VR, Turner AJ (1975) Iterative enhancement: a practical technique for software development. Softw Eng IEEE Trans 4:390–396
10. Beck K, Beedle M, van Bennekum A, Cockburn A, Cunningham W, Fowler M, Grenning J, Highsmith J, Hunt A, Jeffries R, Jon K, Brian M, Martin RC, Steve M, Ken S, Jeff S (2001) The Agile Manifesto. Technical report, The Agile Alliance
11. Benington HD (1983) Production of large computer programs. IEEE Ann Hist Comput 5(4):350–361
12. Boehm BW (1975) The high cost of software. Practical strategies for developing large software systems, pp 3–15
13. Boehm B (2002) Get ready for agile methods, with care. Computer 35(1):64–69
14. Booch G (2005) The complexity of programming models. Keynote talk at AOSD, pp 14–18
15. Brooks FP Jr, Blaauw GA, Buchholz W (1959) Processing data in bits and pieces. IRE Trans Electron Comput 8(2):118–124
16. Brooks FP (1987) No silver bullet: essence and accidents of software engineering. IEEE Comput 20(4):10–19
17. Burks AW (1980) From ENIAC to the Stored-Program Computer. In: Metropolis N (ed) Two Revolutions in Computers History of Computing in the Twentieth Century, Academic Press, pp 311–344. ISBN: 0124916503
18. Canning RG (1972) That maintenance iceberg. EDP Anal 10(10):1–14
19. Carver JC, Kendall RP, Squires SE, Post DE (2007) Software development environments for scientific and engineering software: a series of case studies. In: Software Engineering, 2007. ICSE 2007. 29th International Conference on IEEE, pp 550–559
20. Carver JC, Hochstein L, Kendall RP, Nakamura T, Zelkowitz MV, Basili VR, Post DE (2006) Observations about software development for high end computing. CTWatch Q 2(4A):33–37
21. Cheney W, Kincaid D (1985) Numerical mathematics and computing. Cengage Learn
22. Chen N, Overbey JL (2013) Photran developer's guide. Part II: specialized topics
23. Chikofsky EJ, Cross II JH (1990) Reverse engineering and design recovery: a taxonomy. IEEE Softw 7(1):13–17
24. D'Ambros M (2008) Supporting software evolution analysis with historical dependencies and defect information. In: Software Maintenance, 2008. ICSM 2008. IEEE International Conference on IEEE, pp 412–415
25. de Balbine G (1975) Better manpower utilization using automatic restructuring. In: Proceedings of the May 19–22, 1975, National Computer Conference and Exposition, AFIPS '75. ACM, New York, NY, pp 319–327
26. Dederick LS (1940) The mathematics of exterior ballistic computations. Am Math Mon 47(9):628–634
27. D'Hollander EH, Zhang F, Wang Q (1998) The fortran parallel transformer and its programming environment. Inf Sci 106(3):293–317
28. Easterbrook SM, Johns TC (2009) Engineering the software for understanding climate change. Comput Sci Eng 11(6):65–74
29. Eastwood A (1993) Firm fires shots at legacy systems. Comput Can 19(2):17
30. Edwards PN (2001) A brief history of atmospheric general circulation modeling. Int Geophys 70:67–90

31. Eigenmann R, Hoeflinger J, Jaxon G, Li Z, Padua D (1991) Restructuring fortran programs for cedar. In: Proceedings of the 1991 International Conference on Parallel Processing, pp 57–66
32. Erlikh L (2000) Leveraging legacy system dollars for e-business. IT Prof 2(3):17–23
33. Fowler M, Beck K, Brant J, Opdyke W, Roberts D (1999) Refactoring: improving the design of existing code. Addison-Wesley, Boston
34. Gomez JE (1979) An interactive Fortran structuring aid. In: Proceedings of the 4th International Conference on Software Engineering. IEEE Press, pp 241–244
35. Gorla N (1991) Techniques for application software maintenance. Inf Softw Technol 33(1):65–73
36. Griswold WG, Notkin D (1993) Automated assistance for program restructuring. ACM Trans Softw Eng Methodol (TOSEM) 2(3):228–269
37. Hannay JE, MacLeod C, Singer J, Langtangen HP, Pfahl D, Wilson G (2009) How do scientists develop and use scientific software? In: Proceedings of the 2009 ICSE workshop on software engineering for computational science and engineering. IEEE Computer Society, pp 1–8
38. Horowitz E (1975) Fortran can it be structured-should it be? Computer 8(6):30–37
39. Huff S (1990) Information systems maintenance. Bus Q 55(1):30–32
40. IEEE (1999) IEEE Standard for Software Maintenance, IEEE Std., vol 2. IEEE Press, pp 1219–1998
41. ISO (2006) International Standard—ISO/IEC 14764 IEEE Std 14764-2006. ISO/IEC 14764:2006 (E) IEEE Std 14764-2006 Revision of IEEE Std 1219-1998), pp 1–46
42. Johnson RE (2010) Software development is program transformation. In: Proceedings of the FSE/SDP workshop on future of software engineering research. ACM, pp 177–180
43. Karp AH (1987) Programming for parallelism. Computer 20(5):43–57
44. Kelly DF (2007) A software chasm: software engineering and scientific computing. Softw IEEE 24(6):119–120
45. Kendall R, Carver JC, Fisher D, Henderson D, Mark A, Post D, Rhoades CE, Squires S (2008) Development of a weather forecasting code: a case study. IEEE Softw 25(4):59–65
46. Kontogiannis K, Patil P (1999) Evidence driven object identification in procedural code. In: Software Technology and Engineering Practice, 1999. STEP'99. Proceedings, IEEE, pp 12–21
47. Lammel R, Verhoef C (2001) Cracking the 500-language problem. IEEE Softw 18(6):78–88
48. Larman C, Basili VR (2003) Iterative and incremental development: a brief history. Computer 36(6):47–56
49. Lee G, Kruskal CP, Kuck DJ (1985) An empirical study of automatic restructuring of nonnumerical programs for parallel processors. IEEE Trans Comput 100(10):927–933
50. Lehman MM (1978) Laws of program evolution-rules and tools for programming management
51. Lehman MM, Ramil JF, Wernick PD, Perry DE, Turski WM (1997) Metrics and laws of software evolution-the nineties view. In: Software metrics symposium, 1997. Proceedings, Fourth International, IEEE, pp 20–32
52. Lehman MM (1980) On understanding laws, evolution, and conservation in the large-program life cycle. J Syst Softw 1:213–221
53. Lientz BP, Swanson EB, Tompkins GE (1978) Characteristics of application software maintenance. Commun ACM 21(6):466–471
54. Lynch P (2008) The origins of computer weather prediction and climate modeling. J Comput Phys 227(7):3431–3444
55. McKee JR (1984) Maintenance as a function of design. In: Proceedings of the July 9–12, 1984, National Computer Conference and Exposition. ACM, pp 187–193
56. Méndez M (2016) Aplicaciones del cómputo científico: mantenimiento del software heredado. PhD thesis, Facultad de Informática
57. Méndez M, Overbey J, Garrido A, Tinetti F, Johnson R (2010) A Catalog and two possible classifications of Fortran refactorings. Technical report
58. Méndez M, Tinetti FG, Overbey JL (2014) Climate models: challenges for fortran development tools. In: Proceedings of the 2nd international workshop on software engineering for high performance computing in computational science and engineering. IEEE Press, pp 6–12
59. Mens T, Tourwé T (2004) A survey of software refactoring. IEEE Trans Softw Eng 30(2):126–139
60. Metcalf M (2011) The seven ages of Fortran. J Comput Sci Technol 11(1):1–8
61. Moad J (1990) Maintaining the competitive edge. Datamation 36(4):61
62. Morris C, Segal J (2009) Some challenges facing scientific software developers: the case of molecular biology. In: e-Science, 2009. e-Science'09. Fifth IEEE International Conference on IEEE, pp 216–222
63. Nicholas C, Overbey JL (2013) Photran developer's guide. General information, Part I

64. OpenMP Architecture Review Board (2015) OpenMP application programming interface. http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
65. Overbey JL, Negara S, Johnson RE (2009) Refactoring and the evolution of Fortran. In: 2nd International workshop on software engineering for computational science and engineering (SECSE'09)
66. Overbey JL, Xanthos S, Johnson R, Foote B (2005) Refactorings for Fortran and high-performance computing. In SE-HPCS '05: Proceedings of the second international workshop on software engineering for high performance computing system applications. ACM, New York, NY, pp 37–39
67. Overbey JL, Negara S, Johnson RE (2009) Refactoring and the evolution of Fortran. Urbana 51:61801
68. Overbey J, Rasmussen C (2005) Instant IDEs: supporting new languages in the CDT. In: Proceedings of the 2005 OOPSLA workshop on Eclipse technology exchange. ACM, p 79
69. Pipitone J, Easterbrook S (2012) Assessing climate model software quality: a defect density analysis of three models. Geosci Model Dev Discuss 5(1):347–382
70. Polychronopoulos CD (1988) Automatic restructuring of Fortran programs for parallel execution. Springer, NewYork
71. Port O et al (1988) The software trap-automate or else. Bus Week 3051(9):142–154
72. Ricardo M, Braunschweig F, Leitao P, Neves R, Martins F, Santos A (2000) Mohid 2000, a coastal integrated object oriented model. Hydraulic engineering software VIII. WIT Press, Southampton
73. Riggs R (1969) Computer systems maintenance. Datamation 15(11):227
74. Rope C (2007) Eniac as a stored-program computer: a new look at the old records. IEEE Ann Hist Comput 29(4):82–87
75. Sammet JE, Garfunkel J (1985) Summary of changes in Cobol, 1960–1985. Ann Hist Comput 7(4):342–347
76. Schmidberger M, Brugge B (2012) Need of software engineering methods for high performance computing applications. In: Parallel and distributed computing (ISPDC), 2012 11th international symposium on IEEE, pp 40–46
77. Seacord RC, Plakosh D, Lewis GA (2003) Modernizing legacy systems: software technologies, engineering process and business practices. Addison-Wesley Longman Publishing Co., Inc., Boston, MA
78. Segal J (2008) Models of scientific software development. In: SECSE 08, first international workshop on software engineering in computational science and engineering, Leipzig
79. Segal J (2008) Scientists and software engineers: a tale of two cultures. In: Proceedings of the psychology of programming interest group, PPIG 08, pp 10–12
80. Segal J (2009) Some challenges facing software engineers developing software for scientists. In: 2nd International software engineering for computational scientists and engineers workshop (SECSE '09), ICSE 2009 Workshop, Vancouver, pp 9–14
81. Shaw M et al (2004) National Research Council Computer Science: Reflections on the Field, Reflections from the Field National Academies Press. ISBN-10:0309093015
82. Swanson EB (1976) The dimensions of maintenance. In: Proceedings of the 2nd International Conference on Software Engineering, IEEE. Computer Society Press, pp 492–497
83. Tinetti FG, Méndez M (2012) Fortran legacy software: source code update and possible parallelisation issues. In: ACM SIGPLAN Fortran Forum, vol 31. ACM, pp 5–22
84. Tinetti FG, Méndez M, Giusti AD (2013) Restructuring fortran legacy applications for parallel computing in multiprocessors. J Supercomput 64(2):638–659
85. Triolet R, Feautrier P, Irigoin F (1986) Automatic parallelization of fortran programs in the presence of procedure calls. In: ESOP 86, Springer, pp 210–222
86. Tukey JW (1958) The teaching of concrete mathematics. Am Math Mon 65(1):1–9
87. von Neumann J (1947) Chapter 2: The Point Source Solution. Wave B, Bethe HA, Fuchs K, Hirschfelder JO, Magee JL, Peierls RE, von Neumann J (eds) Los Alamos Scientific laboratory Report LA-2000
88. Ware MP, Wilkie FG, Shapcott M (2007) The application of product measures in directing software maintenance activity. J Softw Maint Evol Res Pract 19(2):133–154
89. Wilson GV (2006) Where's the real bottleneck in scientific computing? Am Sci 94(1):5
90. Zelkowitz MV, Shaw AC, Gannon JD (1979) Principles of Software Engineering and Design Prentice Hall Professional Technical Reference. ISBN:013710202X