COMPUTER AND COMPILER EFFECTS ON CODE RESULTS

STATUS REPORT

COMMITTEE ON THE SAFETY OF NUCLEAR INSTALLATIONS

OECD NUCLEAR ENERGY AGENCY
Le Seine Saint-Germain – 12, boulevard des Îles
F-92130 Issy-les-Moulineaux (France)
Tel. (33) 1 45 24 82 00 Fax (33) 1 45 24 11 10
COMPUTER AND COMPILER EFFECTS ON CODE RESULTS

Status Report

ORGANISATION FOR ECONOMIC CO-OPERATION AND DEVELOPMENT

Paris
ORGANISATION FOR ECONOMIC CO-OPERATION
AND DEVELOPMENT

Pursuant to Article 1 of the Convention signed in Paris on 14th December 1960, and which came into force on 30th September 1961, the Organisation for Economic Co-operation and Development (OECD) shall promote policies designed:

— to achieve the highest sustainable economic growth and employment and a rising standard of living in Member countries, while maintaining financial stability, and thus to contribute to the development of the world economy;
— to contribute to sound economic expansion in Member as well as non-member countries in the process of economic development; and
— to contribute to the expansion of world trade on a multilateral, non-discriminatory basis in accordance with international obligations.

The original Member countries of the OECD are Austria, Belgium, Canada, Denmark, France, Germany, Greece, Iceland, Ireland, Italy, Luxembourg, the Netherlands, Norway, Portugal, Spain, Sweden, Switzerland, Turkey, the United Kingdom and the United States. The following countries became Members subsequently through accession at the dates indicated hereafter: Japan (28th April 1964), Finland (28th January 1969), Australia (7th June 1971), New Zealand (29th May 1973), Mexico (18th May 1994) the Czech Republic (21st December 1995), Hungary (7th May 1996), Poland (22nd November 1996) and the Republic of Korea (12th December 1996). The Commission of the European Communities takes part in the work of the OECD (Article 13 of the OECD Convention).

NUCLEAR ENERGY AGENCY

The OECD Nuclear Energy Agency (NEA) was established on 1st February 1958 under the name of the OEEC European Nuclear Energy Agency. It received its present designation on 20th April 1972, when Japan became its first non-European full Member. NEA membership today consists of all European Member countries of OECD as well as Australia, Canada, Japan, Republic of Korea, Mexico and the United States. The Commission of the European Communities takes part in the work of the Agency.

The primary objective of NEA is to promote co-operation among the governments of its participating countries in furthering the development of nuclear power as a safe, environmentally acceptable and economic energy source.

This is achieved by:

— encouraging harmonization of national regulatory policies and practices, with particular reference to the safety of nuclear installations, protection of man against ionising radiation and preservation of the environment, radioactive waste management, and nuclear third party liability and insurance;
— assessing the contribution of nuclear power to the overall energy supply by keeping under review the technical and economic aspects of nuclear power growth and forecasting demand and supply for the different phases of the nuclear fuel cycle;
— developing exchanges of scientific and technical information particularly through participation in common services;
— setting up international research and development programmes and joint undertakings.

In these and related tasks, NEA works in close collaboration with the International Atomic Energy Agency in Vienna, with which it has concluded a Co-operation Agreement, as well as with other international organisations in the nuclear field.

© OECD 1997
Applications for permission to reproduce or translate all or part of this publication should be made to:
Head of Publications Service, OECD
2, rue André-Pascal, 75775 PARIS CEDEX 16, France
COMMITTEE ON THE SAFETY OF NUCLEAR INSTALLATIONS

The NEA Committee on the Safety of Nuclear Installations (CSNI) is an international committee made up of scientists and engineers. It was set up in 1973 to develop and co-ordinate the activities of the Nuclear Energy Agency concerning the technical aspects of the design, construction and operation of nuclear installations insofar as they affect the safety of such installations. The Committee's purpose is to foster international co-operation in nuclear safety amongst the OECD Member countries.

CSNI constitutes a forum for the exchange of technical information and for collaboration between organisations which can contribute, from their respective backgrounds in research, development, engineering or regulation, to these activities and to the definition of its programme of work. It also reviews the state of knowledge on selected topics of nuclear safety technology and safety assessment, including operating experience. It initiates and conducts programmes identified by these reviews and assessments in order to overcome discrepancies, develop improvements and reach international consensus in different projects and International Standard Problems, and assists in the feedback of the results to participating organisations. Full use is also made of traditional methods of co-operation, such as information exchanges, establishment of working groups and organisation of conferences and specialist meeting.

The greater part of CSNI's current programme of work is concerned with safety technology of water reactors. The principal areas covered are operating experience and the human factor, reactor coolant system behaviour, various aspects of reactor component integrity, the phenomenology of radioactive releases in reactor accidents and their confinement, containment performance, risk assessment and severe accidents. The Committee also studies the safety of the fuel cycle, conducts periodic surveys of reactor safety research programmes and operates an international mechanism for exchanging reports on nuclear power plant incidents.

In implementing its programme, CSNI establishes co-operative mechanisms with NEA's Committee on Nuclear Regulatory Activities (CNRA), responsible for the activities of the Agency concerning the regulation, licensing and inspection of nuclear installations with regard to safety. It also co-operates with NEA's Committee on Radiation Protection and Public Health and NEA's Radioactive Waste Management Committee on matters of common interest.
CSNI Status Report

Computer and Compiler Effects on Code Results

K. Trambauer, GRS Garching

Contents:

1. Introduction
2. Computer Dependent Results
   2.1 Numerical Precision
   2.2 Incorrect Programming
3. Compiler Errors
4. Software Product Quality
   4.1 Programming Rules
   4.2 Release Procedures
5. Installation Procedures
6. Conclusion
7. Acknowledgement
Appendix
1. Introduction

Within the framework of the international effort on the assessment of computer codes, which are designed to describe the overall reactor coolant system (RCS) thermalhydraulic response, core damage progression, and fission product release and transport during severe accidents, there has been a continuous debate as to whether the code results are influenced by different code users or by different computers or compilers. The first aspect, the "Code User Effect", has been investigated already /1/. In this paper the other aspects will be discussed and proposals are given how to make large system codes insensitive to different computers and compilers. Hardware errors and memory problems are not considered in this report.

The codes investigated herein are integrated code systems (e.g. ESTER, MELCOR) and thermalhydraulic system codes with extensions for severe accident simulation (e.g. SCDAP/RELAP, ICARE/CATHARE, ATHLET-CD), and codes to simulate fission product transport (e.g. TRAPMELT, SOPHAEROS).

Since all of these codes are programmed in Fortran 77, the discussion herein is based on this programming language although some remarks will be made about Fortran 90. Even if some code systems make use of data bank systems (e.g. SIGAL, RSYGAL), specific requirements for these will not be considered.

In the following some observations about different code results by using different computers are reported and possible reasons for this unexpected behaviour are listed. Then methods are discussed how to avoid portability problems.

2. Computer Dependent Results

It is generally accepted that computer dependence is caused by

- bugs (i.e. programming errors)
- bad programming practice
- different precision.
Some real bugs are a result of ignoring warning messages when compiling. Bad programming can result in errors, when high levels of optimisation are used. Errors due to precision will be discussed first.

2.1 Numerical Precision

Different computers provide different precisions as single, double or even quadruple precision. Most of the code systems require a word length of 8 bytes or 64 bits (REAL*8). The further discussion is focussed on this format.

Even if the word length is equal, the storage of floating point numbers can be different. This has been described already by Feinauer /2/ and Shepherd /3/. First, the exact word length might be different (60 or 64 bits). Second, the exponent and mantissa representation are different. Unless these attributes are defined by compiler options or by the program (KIND definition in Fortran 90), they are machine dependent.

In IBM-MVS system, the exponent has a size of 7 bits and represents a power of 16, which yields the range of the double-precision floating point number $F$

$$10^{-65} < |F| < 10^{78} - 10^{65}$$

The mantissa has a size of 56 bits but with up to 3 leading bits with zeros, which reduces the relative accuracy, i. e. the precision of $F$ divided by $|F|$, to

$$2^{-53} = 1.11 	imes 10^{-16}$$

In CRAY and CDC under NOS/VE systems, the exponent has a size of 14 bits and represents a power of 2 which yields the range of the single precision floating point of

$$2^{-8193} < |F| < 2^{2466} - 2^{-8193}$$
The mantissa has a size of 48 bits and is normalized, i.e. no leading zeros are present, which yields the relative accuracy of

\[ 2^{-48} = 3.55 \times 10^{-15} \]

or a factor of \(2^6 = 32\) less accurate than the previous mantissa.

The DEC OSF/1 AXP systems data representation \(\text{REAL*8}\) of floating point numbers consists of the sign bit, 11 bits for the exponent and 52 bits for a normalized 53 bit fraction including the redundant most significant fraction bit not represented. The value of data is in the approximate range

\[ 2^{-1025} - 2.22 \times 10^{-308} \leq |F| \leq 1.80 \times 10^{308} - 2^{1025} \]

with a precision of

\[ 2^{52} = 2.22 \times 10^{-16} \]

This means if computer codes are implemented on different computers, one has to consider the different precision, which can alter the results in three different ways:

The code performance is not interfered and the results lie within acceptable uncertainty band.

In some cases the course of event simulation can change (bifurcation or cliff edge effect). Let's assume that a safety valve opens exactly at 7 MPa and an arbitrary small \(\varepsilon\) (e.g. \(10^{-6}\)). In one calculation the maximum system pressure draws close to this value but does not pass over \((1-\varepsilon) \cdot 7\) MPa, in the other case the maximum system pressure reaches \((1+\varepsilon) \cdot 7\) MPa. If one assesses the calculation only regarding the activation of this valve - the calculation results differ essentially. But it has to be considered that this difference lies within the general calculation uncertainty and should not be rated as a precision problem.
An example of magnified uncertainty is the following. When transition from one correlation to the other is not smooth (discontinue) and the change from one correlation to the other is a result of numerical imprecision a significant divergence in the results can be observed.

The code performance is not interfered but the results differ significantly.

This arises if small differences of large numbers are amplified. As an example, the fission product release rate can be determined in two different ways:

\[ m = m \left(1 - \exp(-A \Delta t)\right) / \Delta t \]  
(1)

or after Taylor series development

\[ m = m A \left(1 - \frac{1}{2} A \Delta t + R\right) \]  
(2)

If \( A \Delta t \) draws close to zero (less than \( 10^{-16} \)) the imprecise result in eq. (1) is amplified by the timestep \( \Delta t \) if this value approaches zero too, while in eq. (2) the release rate is in the first order, independent of the time step size. The remainder \( 1/2A \Delta t - R \) is negligible for the absolute value of \( A \Delta t < 10^{-6} \). It is obvious that the result of eq. (1) for small arguments is strongly dependent on the precision, the execution of the intrinsic function, and the compiler.

Similar the expression

\[ y = (x^2 - 4)/(x - 2) \]  
(3)

is undefined for \( x = 2 \), but the singularity can be avoided by the transformation

\[ y = (x + 2) (x - 2)/(x - 2) = x + 2 \]  
(4)

This is clearly a very simple example. In more complex expressions singularities are not so obvious but should be avoided.
A comparable example is discussed in [2]. Feinauer illustrates the accuracy of the different computers with the expression

\[ E = \text{ABS} \left( X - 0.1 \times X \times 10.0 \right) \]  \hspace{1cm} (5)

or

\[ E = \text{ABS} \left( X - (0.1 \times X) \times 10.0 \right) \]  \hspace{1cm} (6)

If the computer code results deviate by the use of different compiler or computer due to this kind of programming, it should be considered as a result of bad programming practice and not as a precision problem.

The code performance is insufficient and the calculation is aborted. This has been observed by Shepherd [3] during the conversion of DRUFAN (which is a predecessor version of ATHLET) to run on a CRAY. Due to the less precise floating point number of the CRAY compared to the AMDAHL, an IBM-like computer under MVS, two iteration procedures failed. One was a Newton-Raphson type iteration with a normalised convergence limit of $10^{-19}$ for the determination of the initial enthalpy distribution in the coolant system (new subroutine names in ATHLET are DENTM3 and DENTM4). By changing the limit to $10^{-12}$ the scheme converged satisfactorily. The other was the generalised ordinary differential solver package FEBE, based on a Burlisch-Stoer algorithm, for solving the differential equations. The problem arises in the explicit part of the algorithm, which in general is not recommended to use. Based on a strict error control the system uses an increasing number of sub-timesteps to find a converged solution. In this procedure, a denominator was checked against a limit of $10^{-20}$. On the CRAY the procedure did not converge due to less precise floating point numbers. When replaced with a limit of $10^{-19}$ it arrived at the correct result.

These two examples illustrate that iteration procedures have to be adapted to the precision of the computer used, to assure the portability of the computer code. For example, ATHLET uses for the determination of the partial derivatives in the Jacobian Matrix a multiple value of the relative precision RUNKON [5] calculated at the beginning of the transient solution:
HALFU = 0.5D0
50 TEMP1 = 1.0D0 + HALFU
    IF (TEMP1 .LE. 1.0D0) GOTO 100
    HALFU = 0.5D0 * HALFU
    GOTO 50
100 RUNKON = 2.0D0 * HALFU

By this way the code performance is sufficient. The code results will be influenced by the precision, but this difference lies within the general code uncertainty.

2.2 Incorrect Programming

Some indication of bad programming has been given already in the previous paragraph. In the following further possible incorrectnesses are listed which might be a source of error.

Initialization of variables
The use of an uninitialized variable is faulty. If the computer generally presets memory to zero during the load sequence for a program, this fault will possibly not be detected. It is recommended /2/ to preset the memory explicitly to negative infinity or negative indefinite so that if variables are not properly initialized within the program, a total execution error will result.

Reuse of variables after subroutine exit
Some computers keep the value of a variable after subroutine or function exit but others not. The use of the statement SAVE is required to ensure that the variable remains unchanged.

Length of parameter list
To assure the portability of computer codes, the parameter list must have identical length and the type of data (REAL*8, REAL*4, INTEGER, etc.) must be identical for a specific variable in the subroutine itself and in all calling routines. It is recommended to sort the variables in the parameter list according to their type with decreasing word length to ensure that the passed arguments are naturally aligned, i.e. the word boundaries agree with the natural boundaries of the data structure.
Dimension of arrays

Arrays with static allocation must be defined during compilation with the maximum length ever used in the program. Otherwise data are overwritten or parts of the programme are destroyed. Fortran 90 allows the use of allocatable arrays with the statements `ALLOCATE` and `DEALLOCATE`. Then the maximum length of the array is defined in the block between these two statements and all array operations are performed for this actual dimension.

**INCLUDE statements and COMMON**

`INCLUDE` statements are not ANSI-Standard, therefore some programming rules do not allow them or the use of `BLOCKDATA` with the central dimensioning of the `COMMON` variables. Also Fortran 90 recommends the use of `MODULES` and `SAVE` as an alternative approach. Nevertheless some programs, e.g. ATHLET, use `COMMON` blocks to facilitate the data transfer between subroutines and via interface routines between different modules instead of data bank systems as other programs, e.g. ICARE. For a `COMMON` block it is essential that the word boundaries align with the data structure. To avoid misaligned data the following rules are recommended /5/:

- Always define the largest numeric data items first.
- Add small data items of the correct size (or padding) before otherwise misaligned data to ensure natural alignment for the data that follows.
- If your data includes a mixture of character and numeric data, place the numeric data first.

However ATHLET programming /6/ rules allow neither variables of different data type nor more than one array on one `COMMON` block.

**EQUIVALENCE statement**

An `EQUIVALENCE` statement is used to specify the sharing of storage units by two or more entities in a program unit. It does not equate the elements mathematically. Special care is needed to avoid data mismatch /6/. Most programming rules forbid the use of the `EQUIVALENCE` statement to facilitate the maintenance of the code. In Fortran 90 it is declared as obsolescent and it should be replaced by the `TRANSFER` function. Incorrect use of the `EQUIVALENCE` statement is difficult to detect and might be the reason for „computer effects on code results“.
Arithmetic Relational Expression

Arithmetic relational expressions (e1 relop e2) with data of real type are sensitive to the precision of the floating point numbers. The sensitivity is less if as relationed operators (relop) .LT., .LE., .GT., and .GE. are used and the difference of the two expressions (e1 - e2) covers a range

\[-E_1 \leq e1 - e2 \leq E_2\]

with E_1 and E_2 sufficiently large against the precision of the data. If one of the range boundaries (E_1 or E_2) is less than the precision of the data or if the relational operator is .EQ. or .NE., then the results depend in a high degree on the precision. To complete this deficiency, the comparison of two real data should be replaced by a logical function which considers an uncertainty spread as in the following example. This logical function compares two REAL*8 values considering a relative and an absolute tolerance of 10^-7, which is considerably larger than the precision of the data.

```
LOGICAL FUNCTION SEQR8 (R1, R2)
RU = R1 + DABS (R1) * 1.D-7 + 1.D-7
RL = R1 - DABS (R1) * 1.D-7 - 1.D-7
IF (R2.GE.RL .AND. R2.LE.RU) THEN
    SEQR8 = .TRUE.
ELSE
    SEQR8 = .FALSE.
ENDIF
RETURN
END
```

Double precision

Feinauer [2] also discusses the use of DOUBLE PRECISION. The solution of the differential equation system generally requires a high precision as provided by CDC machines or CRAY. To achieve equivalent precision on IBM machines (32-bit word), variables must be declared double precision. To explicitly change a single-precision program into a double-precision program, care must be taken to make sure that all floating point arithmetic is done in double precision where the accuracy is needed. Propagation of inaccuracies arising from "mixed-mode" arithmetic (performing arithmetic on a mixture of single- and double-precision values) can result in
divergence of numerical solutions. Avoiding such inaccuracies is especially important in the solution of nonlinear equations, where a small perturbation may grow to swamp out the true solution. With the statement `IMPLICIT DOUBLE PRECISION` all real variables are declared as `REAL*8`. But this is sufficient only for assignments of data which are representable in binary notation, i.e. by a series of $2^n$:

$$R = \sum_{n=-41}^{59} k_n 2^n$$

with

$k_n = 0$ or $1$ and $R$ with less than 19 significant digits, e.g.

$$\sum_{n=-13}^{11} 2^n = 16382.9998779296875$$

$$2^{-41} = 0.0000000000000454747350886464119$$

$$2^{19} = 576460752303423488.$$ 

Other numbers bear truncation errors and to avoid undesirable extension of the number representation the data must contain format descriptor „D“ as shown in the following table:

<table>
<thead>
<tr>
<th>INPUT</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.124</td>
<td>0.124000001287460327</td>
</tr>
<tr>
<td>0.124E0</td>
<td>0.12400001287460327</td>
</tr>
<tr>
<td>0.124D0</td>
<td>0.12400000000000000000</td>
</tr>
<tr>
<td>0.125</td>
<td>0.12500000000000000000</td>
</tr>
<tr>
<td>0.125E0</td>
<td>0.12500000000000000000</td>
</tr>
<tr>
<td>0.125D0</td>
<td>0.12500000000000000000</td>
</tr>
<tr>
<td>0.126</td>
<td>0.12599998712539673</td>
</tr>
<tr>
<td>0.126E0</td>
<td>0.12599998712539673</td>
</tr>
<tr>
<td>0.126D0</td>
<td>0.12600000000000000000</td>
</tr>
</tbody>
</table>

The different situation for 0.125 and the two other numbers is that

$$R = 0.125 = 2^3$$

results from above equation with $k_n = 0$ except $k_3 = 1$, while the two others are not representable in binary notation. The maximum relative truncation error corresponds to the precision of `REAL*4`

$$2^{-24} \sim 6 \times 10^{-8}$$
The problem for the code developer in converting a program from single to double precision is that the program may contain thousands of single-precision constants in DATA statements and in other assignment statements throughout the program. Changing all occurrences of single-precision constants to double-precision is very time-consuming, and it complicates the maintenance of two versions of the program. IBM supplies an alternate method for converting programs from single to double-precision through the automatic double-precision option (AUTODBL). Nevertheless, Feinauer /2/ recognized that this does not solve all conversion problems.

The incorrectness described in this paragraph may lead to different code results, if the code is transferred to a different machine independent of the compiler optimization. To avoid these incorrectnesses, the programmer should observe programming rules, which are common in software laboratories. They are described in paragraph 4.1.

3. Compiler Errors

One has to distinguish between real compiler errors and the incapability of compiler to convert specific parts of the program at higher optimisation level.

Problems arising during the conversion from single to double precision have been already discussed in the previous paragraph. Other problems may be generated by tricky or incompatible programming which should be recognised by the compiler and indicated by an error message.

Sometimes there are real errors in the compiler, namely if the compiler is new on the market and has not been sufficiently tested. There is no error message during compilation and the program is either incorrectly executed, which is not easy to detect, or terminated without indication what causes it. Either the compiler makes an error in the data management or it misinterprets a block of statements and does not report, that there are difficulties in the correct conversion. For illustration two examples are given.
1st Example:

```
DIMENSION A(N)
DO 10 I = 1, N-1
   A(I+1) = A(I+1) + A(I)
10    CONTINUE
```

This block of statements is correct, nevertheless changing from optimisation level 2 to 3 the range of the array A was overstepped and the following data at the position A(N+1) was overwritten. After changing the statements into

```
DIMENSION A(N)
DO 10 I = 2, N
   A(I) = A(I) + A(I-1)
10    CONTINUE
```

the execution was perfect.

2nd Example:

```
DIMENSION A(N), B(N), C(N)
DO 100 I=2, N
   IF(ABS(C(I)-A(I)) .LT. 1.D-6) THEN
      B(I)=C(I)
   ELSE
      GRAD=(C(I)-A(I))/(C(I)-A(I))
      B(I)=A(I)+(T-A(I)) * GRAD
   ENDIF
100   CONTINUE
```

At low optimisation, this block of statements was executed correctly. At higher level (Opt. 3) the statement

```
GRAD = (C(I) - A(I)) / (C(I) - A(I))
```

was executed before the IF block, which yields a data overflow for

```
C(I) - A(I) = 0.
```
The execution was correct after restructuring as follows:

```plaintext
DIMENSION A(N), B(N), C(N)
IF (ABS (C(1)-A(1)) .LT. 1.D-6) THEN
   DO 100 I=2, N
      B(I)=C(I)
   100 CONTINUE
ELSE
   DO 200 I=2, N
      GRAD=(C(I)-A(I))/(C(1)-A(1))
      B(I)=A(I)+(T-A(1))*GRAD
   200 CONTINUE
ENDIF
```

These two figures demonstrate that compiler errors are not easy to detect and general guidelines cannot be provided, except that clearly arranged, straightforward programming facilitate the compilation at higher optimisation levels. It is obvious that observation of the programming rules is mandatory.

4. Software Product Quality

Computer programs are used extensively in scientific and engineering applications involving the design of facilities or the prediction of results of physical phenomena. Since errors in such programs may have very costly consequences, it is necessary that the development of these programs exhibit a high level of reliability. Verification and validation (V&V) is a systematic approach to improving reliability. Although V & V activities may be conducted on existing programs, they are most effective when carried out in parallel with program development. A parallel approach allows for early detection and correction of errors in the software requirements, design, and coding.

This approach is generally laid out in guidelines and handbooks /7, 8, 9, 10, 11/ which should be referred to for further details. They are very useful for project management and the planning of development phases such as

- Initiation
- Definition
- Design
- Coding
- Integration and Testing
Installation

Operation and Maintenance.

An important problem in programme development and maintenance is version control, i.e. the task of keeping a software system consisting of many versions and configurations well organized. This task is assisted by specific software tools for configuration management, e.g. MAKE /12/ and revision control system, e.g. RCS /13/.

Today, powerful software development tools exist also on the market (e.g. Clear Case). These tools are used for version control, configuration management and process control. They allow to define software development politics and procedures. Such a tool is used for some phase in the development and maintenance of ESTER.

A helpful link to the high-level quality attributes of the International Standard for Software Product Evaluation ISO 9126 /14/ is provided by Dromey's model for Software Product Quality /15/. Dromey identifies the impact of different product defects on the seven quality attributes:

- functionality
- reliability
- usability
- efficiency
- maintainability
- portability
- reliability.

The quality-carrying properties associated with the structural forms of programs are grouped into four basic categories. These basic categories carry low-level quality properties, which are ranked from the highest to the lowest precedence:
• Correctness properties
  - Computable  
  - Complete  
  - Assigned  
  - Precise  
  - Initialized  
  - Progressive  
  - Variant  
  - Consistent  

• Structural properties
  - Structured  
  - Resolved  
  - Homogeneous  
  - Effective  
  - Nonredundant  
  - Direct  
  - Adjustable  
  - Range-independent  
  - Utilized  

• Modularity properties
  - Parameterized  
  - Loosely coupled  
  - Encapsulated  
  - Cohesive  
  - Generic  
  - Abstract  

• Descriptive properties
  - Specified  
  - Documented  
  - Self-descriptive  

Preconditions and postconditions provided

Comments associated with all blocks

Identifiers have meaningful names
To fulfill these quality properties, programming rules are established and to assure the quality of the product release procedures are performed, which are discussed in the following.

4.1 Programming Rules

Programming rules are established by GRS, IKE /16/ and IPSN. They are described in a generalised way in the appendix. It is assumed that they are similar in other software laboratories. These rules must be followed, as far as possible, by developers in charge of implementation of new modules and/or of old module development in existing versions, in order to keep a satisfactory homogeneity between all the modules of the code. This will make easier code maintenance by imposing correct software quality and to allow the portability of the code on various computers. For any exception, the allowance must be given in advance by the person responsible for quality assurance of the program.

Programming rules may be classified in four different categories:
– Code portability. They deal especially with ANSI standard programming.
– Code legibility. They deal with routine length, comment quantity and distribution, labels and format numbering, continuation characters and programming architecture.
– Code maintenance. They supply information on nomenclature, and prevent the use of several kinds of declarations and statements.
– Code usability. They deal with the interface to the user.

Besides these rules which concern the formal programming, there are many aspects to obey which concern the modelling and physical description of processes, but this not in the scope of this report.

4.2 Release Procedure

To assure the quality of a new code release different steps are considered. New models incorporated into the code should be subject to independent peer review, the theory and assessment results should be published in the code manuals for
each version. The programming can be scrutinized by employing tools like FTNCHEK /17/, LAXYM or FORGE /18/. These tools also support the documentation of the code by generation of flow charts, variable listing, etc..

The restart capability must be checked to ensure that all necessary data are transmitted. The code must continue after a restart with identical results, if the input is not altered.

Finally, the code has to be subject to portability tests by running different reference test cases on different computers with different levels of compiler optimization. If the results of the code deviate significantly, the reason must be selected and, if possible, eliminated. If there is a suspicion that one compiler optimization is incorrect, it should be documented and the manufacturer should be informed.

Input and output of selected reference test cases should be part of the software package when the code is transmitted to other users to facilitate the checking of proper installation of the code by the user.

5. Installation Procedure

The users point of view, how to reduce the so-called „Computer/Compiler Effects“, was lay out by Vanhorenacher /19/ who has a wide experience in running large codes on different computers:

In general, the thermalhydraulic expert is not familiar with all the computer techniques and relies on computer experts to get the code properly installed. Therefore some specifications have to be provided in order to judge if the installation is correct. The computer expert will generally compile the codes with the highest optimisation option. Sometimes, it doesn’t run and therefore the optimisation level has to be reduced. It is obvious that starting with the debug option is a better procedure. If the results of the test cases provided by the code developers are identical one can move to a higher level of optimisation. Should the next tentative not provide the same results, one can try to isolate the subroutine(s) that has to remain at a low level of optimisation. Proceeding this way, step by step, it is
possible to increase the performance (cputime) with an acceptable guarantee on
the results. It is recognised that this process could be long.

A very difficult task for the code user is to specify the criterion of acceptance. Often
the results are not strictly the same. The acceptance criteria should be oriented an
physically reasonable uncertainty based on experimental experience.

Often the workstations are shared and the code user is not always aware of some
computer modifications performed by the system analyst (cf. change of OS, of
mathematical libraries, ...). Therefore very strict QA procedures should be
implemented: for example at TRACTEBEL; beside the QA procedure each month a
set of runs are performed in order to verify that no changes have occurred.

6. Conclusion

The use of a computer code on different computers as well as the application of
different compiler or computer optimizations can lead to different results.

This is acceptable if the differences in the results are unessential or lie within the
model uncertainty of model correlations, which should be verified by code validation.

It is not acceptable, if the deviation of the results is significant or the code
performance gets worse. In this case, the cause of this deviation must be detected
and the incorrect programming improved or the use of the erroneous compiler
version must be excluded.

The dependency of code results on machine precision and optimization can be
minimized by correct and structured programming.
7. Acknowledgement

The author gratefully acknowledges the support of his colleagues, namely H. Austregesilo, T. Kullmann and P. Puhr-Westerheide from GRS, of E. W. Coryell (INEL), M. Delaval (JRC Ispra), and of the reviewers B. Adroguer (IPSN Cadarache), F. D’Auria (University Pisa), T. Haste (AEA Technology), I. Shepherd (IRC Ispra) and L. Vanhovenacker (TRACTEBEL).

The work has been sponsored by the German Federal Ministry for Education, Science, Research and Technology (BMBF) in the framework of ATHLET-CD code development and validation.
References

/1/ Aksan, S. N., D'Auria, F., and Städtke, H.
User Effects on the Transient System Code Calculation
NEA/CSN/R (94) 35, Jan. 1995

/2/ Feinauer, L. R.
Compiler Issues associated with Safety-related Software
Nuclear Technology Vol. 93, Jan. 1991

/3/ Daolio, B., Shepherd, I.
DRUFAN on the CRAY - An Analysis of CPU Usage, Parallel Processing
Possibilities and Comparison with other IRC Codes
CEC Luxembourg, Report EUR 11053 EN, 1987

/4/ DEC Fortran 90, User Manual for DEC OSF/1 AXP Systems
DEC Maynard, MAS, June 1994

/5/ Shampine, L. F., and Gordon, M. K.
Computer Solution of Ordinary Differential Equations: The Initial Value
Problem

/6/ ATHLET Developer's Manual
GRS Garching, June 1995

/7/ Military Standard DoD-STD-2167 Related DIDs
US DoD 1985

/8/ Bröhl, A. P., Dröschel, W.
Das V-Modell: Der Standard für die Softwareentwicklung mit Praxisleitfaden
Oldenbourg Verlag München, 1993

/9/ American National Standard Guidelines for the Verification and Validation of
Scientific and Engineering Computer Programs for the Nuclear Industry
ANSI/ANSI-10.4- 1987
Bryant, J. L., Wilburn, N. P.
Handbook of Software Quality Assurance Techniques applicable to the
Nuclear Industry
NUREG/CR-4640, Aug. 1987

Slater, J. A. et al.
A Guide to Software QMS Construction using ISO9001/EN29001/BS5750pt1
(1981)
UK Department of Trade and Industry, (February 1992)

Feldman, S. I.
MAKE - a Program for Maintaining Computer Programs
Software - Practice & Experience 9,3 (March 1979)

Tichy, W. F.
RCS - A System for Version Control
Software - Practice & Experience 15,7 (July 1985)
Updated 1993/10/07, Purdue University, West Lafayette, IN.

Software Product Evaluation - Quality Characteristics and Guidelines for
their Use

Dromey, R. G.
A Model for Software Product Quality

Bruder, M., Schatz, A.
Integration von KESS III-Modellen in ATHLET-CD und Beiträge zur
Programm verifikation
Appendix A, IKE 2-111, Universität Stuttgart (Juli 1994)

Moniot, R., et al.
FTNCEHK - Fortran program checker
Fordham University (moniot@mary.fordham.edu)
/18/ FORGE Explorer User's Guide
Applied Parallel Research (support@apri.com)

/19/ Vanhoenacker, private communication
Appendix: Programming Rules

Portability rules:
1. The ANSI standard must be applied.
2. The subroutines have to be successfully compiled with ANSI standard compilers on several computers.
3. Compiler warnings and informations have to be eliminated.

Legibility rules:
1. The size of each subroutine must not exceed N Fortran lines.
2. A special comment block must be present at the beginning of each subroutine and located just after the first statement (which is subroutine declaration). This comment block must obey to a given architecture and must contain several items. Each item may consist of several lines. These items must be supplied in the following mandatory order:
   - Subroutine name or function name
   - Version identifier
   - First author name (date)
   - Last modifier name (date)
   - Purpose of the subroutine/function
   - Arguments: name and definition with identification for input (I), output (O) or update (U)
   - List of calling subroutines
   - List of called subroutines
3. The labels are mandatory on CONTINUE and FORMAT statements, and forbidden on other statements.
4. The FORMAT statements must be grouped together just before the END statement. The labels on FORMAT statements must be greater than M.
5. The labels in each subroutine must be defined in increasing order.
6. The loops must appear clearly by doing a set back of K blank characters on all the statements located between the DO and the CONTINUE statements. The DO and CONTINUE statements must start in the same column.
7. The conditional tests must appear clearly by doing a set back of K blank characters on all the statements located between the IF and the ELSE, ELSEIF
or ENDIF statements. The IF, ELSE, ELSEIF and ENDIF statements must start in the same column.

8. Loops and conditional tests must be commented (purpose, conditions).

**Maintenance rules:**

1. A subroutine must not perform different tasks of various nature. The name of a subroutine refers to its task and must as far as possible have six letters. (ATHLET: The name must begin with the module-specific letter as defined in /G/).

2. All the subroutines of the program must be used.

3. Two procedures cannot have the same name in two different files.

4. Each file must contain only one subroutine.

5. A subroutine must have one and only one input point (no ENTRY).

6. A subroutine must contain at least one statement.

7. A subroutine must not contain unreachable statements.

8. The arguments number (and order) of each subroutine must be coherent between the declaration and the different calls.

9. The arguments number of a subroutine must not exceed M₁.

10. The local variables number of a subroutine must not exceed M₂.

11. Each subroutine must begin (just after the comment block previously defined) by the statement IMPLICIT DOUBLE PRECISION (A-H, O-Z). (Double precision is useful only for 32-bit word machines (see also point 12). Implicit assignment of real and integer variables is usual. In SCDAP explicit typing of variables is used (IMPLICIT NONE). The type of each and every variable must then be specified (see also point 13).)

12. The authorized types for variables are INTEGER, DOUBLE PRECISION and CHARACTER (OUTPUT: REAL*4).

13. Integer variables must begin by the letters I, J, K, L, M, N. Real variables must not begin by these letters.

14. No blank line is allowed. The letter C must be used in first column for each comment card. Comments within a FORTRAN statement with ! are not allowed.

15. The use of inclusion files is restricted to data bank systems.

16. The recursive calls, either direct or indirect, are forbidden.

17. A loop index must not be modified within the loop.

18. A loop control variable must not be modified within the loop.

19. A loop must have one and only one input point.
20. A **CONTINUE** statement, ending a **DO** loop, must apply to one and only one loop. If two loops are enclosed, two different labels must be used.

21. The label of a back **GOTO** statement must be commented (caution: do while).

22. A **COMMON** instruction must define only one common.

23. The use of the **BLOCKDATA** statement is forbidden (not in ATHLET).

24. The use of **EQUIVALENCE** statement is forbidden (except RSYT routines).

25. The same constant name must not refer to two different values.

26. Physical constants (e.g. pi, stefan constant) must be defined uniformly by **PARAMETER** statement.

27. The use of the arithmetic **IF** is forbidden.

28. The use of calculated **GOTO** is forbidden.

29. The use of assigned **GOTO** is forbidden.

30. The use of the **ASSIGN** statement is forbidden.

31. The use of the optional **RETURN** is forbidden.

32. The use of the **PRINT** statement is forbidden.

33. Unit numbers in **READ, WRITE, OPEN, CLOSE** must be declared by a variable which is defined at a central position (only ATHLET).

34. The common name must begin with the letter C, followed by the module specific letter as defined in /6/ (only ATHLET).

35. All common variables must be described in the include library (only ATHLET).

**Usability rules** (partly ATHLET-specific):

1. All input for a specific module must be controlled by one central input routine.

2. The input must be structured by **CONTROL** and **KEY** words.

3. Restart output and input must be controlled by one module-specific routine.

4. Transient print and plot output must be controlled by one module-specific routine.

5. The output data must be structured by **OBJECT, MODEL, VARIABLE** names and index number.

6. The use of standard output subroutines is mandatory (print output in tabular fashion).

7. All input and output variables must be described in the user's manual.

8. The name of the input and output variables must be meaningful, their physical units must be international standard units.