EUROPEAN COMMISSION

Community Research

SEVENTH FRAMEWORK PROGRAMME

ARCHER

Collaborative Project

Co-funded by the European Commission under the
Euratom Research and Training Programme on Nuclear Energy
within the Seventh Framework Programme

Grant Agreement Number: 269892
Start date: 01/02/2011 Duration: 48 Months

www.archer-project.eu

Advanced Reactor for Cogeneration of Heat & Electricity-R&D

D13.21 - System Code Requirements

Authors: Stefan Kasselmann, Daniela Lambertz,
Sarah Scholthaus, Shubham Trivedi (FZJ)

ARCHER project – Contract Number: 269892
Advanced High-Temperature Reactors for Cogeneration of Heat and Electricity R&D
EC Scientific Officer: Dr. Panagiotis Manolatos

Document title	System Code Requirements
Author(s)	Stefan Kasselmann (FZJ), Daniela Lambertz (FZJ), Shubham Trivedi (FZJ)
Number of pages	19
Document type	Deliverable
Work Package	WP13
Document number	D13.21 – revision 1
Issued by	Forschungszentrum Jülich GmbH
Date of completion	28/06/2013
Dissemination level	Restricted to the partners of the ARCHER project

Summary

This document contains a description of tools and best practices for future simulation codes in the field of HTR safety research and reports about the requirements to successfully integrate different codes into a consistent code package by using an appropriate environment for a European HTR code development platform.

Approval

Rev.	Short description	First author	WP Leader	SP leader	Coordinator
0	First issue	S. Kasselmann (FZJ) 28/06/2013	S. Kasselmann (FZJ)	F. Roelofs (NRG)	S. Knol (NRG)

Distribution list

Name	Organisation	Comments
P. Manolatos	EC DG RTD	
All technical participants	ARCHER	
Steering Committee members	ARCHER	

Table of contents

1 Introduction

1.1 What is code integration?

The simulation of High Temperature Reactor (HTR) cores is a complex task which includes aspects like neutron diffusion or transport, time dependent nuclide inventories, fuel temperature and gas flow, fuel performance and fission product (FP) releases etc. Neutronics and fluid dynamics (FD) have a mutual feedback while other code modules (e.g. FP release) just use data from the FD model. In any case a vast amount of data has to be shared between the different code modules. In the last decades a variety of computer codes have been developed, validated and optimized to simulate the different safety and operational aspects of HTRs. In order to overcome present limitations of these codes and to exploit the advantages of modern computer clusters, several code integration projects have been initiated to integrate individual programs into consistent code packages. This deliverable reports about the requirements to successfully integrate different codes into a consistent code package suitable to be applied for a future HTR demonstrator.

The sharing of data among computer codes can be realized in different ways. The choice of one of the possible solutions depends for example on the following issues:

- Availability of source code
- Programming languages used
- Licence issues
- Documentation level
- Project timeline restrictions
- Skills of programmers

For example if no source code is available, the only way to share data between different codes is to make use of input/output (I/O) coupling. Another problem might be the usage of different programming languages which might not be appropriate to be coupled on source term level.
Despite common definitions of code coupling and cohesion in software engineering, there are basically three different ways to share data between separate code parts:

- I/O coupling
- Usage of Application Programming Interfaces (API)
- Code-to-Code coupling

I/O coupling is often used because the codes themselves have not to be modified. Instead, only an interface code is needed which is able to read the output of code A and prepare the input for code B. While this may be a good solution for smaller projects, it is not the preferred way for large code systems mainly for two reasons. Imagine the following situation: A reactor model has about 10,000 burnup regions. For each region the time dependent amounts of about 1,000 nuclides are to be calculated. The result for each region with its 1,000 nuclides is stored in a file. Applying I/O coupling in this scenario means we have to…

1. … deal with about 10 million files for each simulation time step (organizational issue)
2. … wait for about 20 million files to be written and read to and from disk for each simulation time step (performance issue)

Making use if APIs in general is a good way of sharing data between well-defined interfaces. It allows other users to connect to your code on source code level while hiding the implementation details in the libraries shipped with the APIs. The main problem here is that libraries are created for distinct software platforms (e.g. UNIX, Windows, MacOS) which leads to code fragmentation.

The preferred way to share data between different codes is to exchange data on the source code level. Here people can work on a common set of data containers which are well designed (with respect to the use case) and implemented according to the language standard. The data transfer via objects is also fast because all data can be kept in the main memory at any time during the calculation. More details about this approach are given in chapter 3.

So when we talk about code integration, we talk about designing a system which is sharing data on the source code level via well-defined data objects. How a system is well-designed is one question. Another question is how to organize the collaborative work of different people around the world from a technical point of view. This aspect is discussed in chapter 2.

1.2 Why code integration?

Many codes in the field of reactor physics share a long development history with origins that date back to the 60s of the last century. Due to the limitations of computer science at that time (e.g. memory) only parts of the actual model could be simulated at the same time. Input data sets were "stored" on punch cards, intermediate and final results were stored on magnetic tapes.

When the first personal computers came available in the 1980s programs were ported to those machines but the basic structure of the programs was kept almost the same. Different pieces of some codes still use so called temporary binary scratch files for communication. The "data structure" are usually large arrays which are subdivided as needed while the code parts are connected by "common blocks" or "modules". Punch cards were "translated" into fixed form and later on into free form input "cards". These inputs are basically text based (ASCII) input files. Finally the heterogeneous infrastructure of computer codes has led to some major limitations and drawbacks with respect to functionality and flexibility. The purely procedural structure of the codes which has evolved over the last decades does not meet the requirements of a modular and future-proof code system.

An example might explain the disadvantage of this approach: The former workflow to simulate FP release during a nuclear excursion accident scenario at FZJ included the generation of models for the codes VSOP, MGT, FRESCO-II and eventually PANAMA. Especially the codes VSOP and MGT share a lot of identical input data, for example the nuclear and fluid dynamics calculation grid, the materials and the nuclear data library. Another example is the definition of the (same) fuel element in VSOP and FRESCO-II. Also some codes contain similar physical models and apply similar algorithms, so there was a high level of code duplication.

This problem is aggravated when different codes are maintained by separate working groups where people have very limited knowledge about the capabilities and restrictions of other codes in use. In addition the lack of modern code development tools like version management, automated documentation, systematic unit and regression testing, and coding conventions for all developers lead to a huge loss of knowledge (input models) and code fragmentation.

So all this can be summarized by saying that an integrated code package is essential for future research in the field of nuclear reactor safety.

2 Code Development and Quality Assurance

2.1 General Aspects of Code Design

The code development process is structured in different steps: Planning, implementation, testing, documentation, deployment and maintenance. Different software development models are known as the waterfall model, the spiral model, agile development, rapid prototyping etc. All of them have their special strengths and weaknesses. A detailed description of these models can not be given within the scope of this report. Instead the authors want to report about the special situation of code development in the area of nuclear safety research.

Here people are facing the situation that a lot of legacy codes exists. The total sum of FORTRAN source code currently in use by the authors amounts to almost 100,000 lines of code (LOC) and represents several thousand person-months of work. At their time of creation, software engineering was still in its infancy. There were no special code libraries available, so programmers were relying on what the actual compiler in use would offer them. Also, people had to teach themselves how to program. Often there was just one author of a code. Detailed reports about how the code is actually

working are rarely available. This lead to the problem that many codes from a today's point of view are technically outdated and poorly documented. The purely procedural structure of the codes which has evolved over the last few decades does not meet the requirements for a modular and future-proof code system.

Therefore the question arose as to the kind of software design approach that should be chosen in the long run. It is widely accepted that software complexity and code maintenance costs can be reduced by choosing an appropriate object-oriented (OO) design approach. The main goal here is to divide the responsibilities of the program among its objects. An object-oriented software design thus strives to reduce the amount of coupling between the different objects, which reduces the complexity of the entire program. In the field of nuclear reactor codes, objects like *Mesh*, *Batch*, *FuelElement*, *Nuclide*, *Material* are appropriate, for example.

An object oriented (OO) code development approach is mandatory for collaboration with other research facilities in the field of nuclear reactor safety R&D.

2.2 Refactoring of Legacy Codes

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. It is a well-known and frequently used method in the field of computer science which has proven its worth. The goals of refactorization are to increase readability, to eliminate duplicate code and to speed up the implementation of new features.

The quality of any long-lived source code tends to degrade over time. Such systems often exhibit code duplication and global information sharing. Maintenance and expansion can become tedious, costly, and time-consuming. One solution to this gradual software decay is refactoring [2]. A refactoring process is advisable, if the following criteria do apply for a code (excerpt) :

- Usage of long methods and parameter lists
- Presence of duplicated or dead code parts
- Presence of patterns like *shotgun surgery*, *data clumps*, *temporary fields*, *primitive obsession*,…

Special care has to be taken in case of old legacy Fortran code. Here a lot of potential problems have to be removed first (excerpt) :

- Implicit data types
- Common blocks
- Equivalence statements
- Memory address calculations
- Static memory allocation
- Six character variable naming
- Highly nested loops and logical conditions
- Capitalization through the whole code
- Fixed format code layout
- Missing indentation

Because a detailed knowledge about the modeled physics is mandatory to perform extensive code rearrangements, refactoring needs a cooperation of scientists and software engineers. Before the actual refactoring can be started, the code has to be modified with respect to recent programming standards (e.g. Fortran 90/2003). Most of the problems during a software project arise due to non-compliant source code where bugs cannot automatically be detected by the compiler.

During the refactoring process, the simulation results which have been validated over a long period of time must be reproducible. This can be guaranteed by setting up a version control and documentation system where every modification of the code is tagged and cross-checked using a standardized set of input test samples. All code revisions are accessible through a file server by

every work group member. All revisions have to be checked to reproduce the results of the previous ones. This topic is discussed more in detail in chapter 2.3.

As an example **Fig. 1 (left)** shows a typical situation before the refactoring. The subroutine continues over several pages (not shown here), the different tasks of this subroutine are not clear from the source code itself. In **Fig. 1 (right)** the same piece of code after the refactoring process is shown. The subroutine has been subdivided according to its tasks. The names are self-descriptive and underline the sequence of workflow. This modularization process should be completed before any new feature is implemented.

```
      subroutine tispecrl (j2,kmatg,imatg,ljg,njg,lkg,nkg,
     >    libkart,fisg,satzq,satzr,satzs)

C     Subroutine TISPECRL erstellt die interne Kurz-Library fuer   !AL68
C     diesen Job.                                                  !AL68

CC    IMPLICIT REAL*8(A-H,O-Z)          ! darf hier fehlen         !AL68

      integer imatg(kmatg)
      real*4 fisg(62)
      real*4 satzq(193,kmatg)
      real*4 satzr(ljg,njg)
      real*4 satzs(lkg,nkg)
      real*4 x(193)

      CHARACTER*88  libkart(-1:njg+nkg)


      READ (J2,'(9A4,16x,3I4)') (x(I),I=1,9), NI,NJ,NK
      write (libkart(-1),'(9A4,16x,3I4)') (x(I),I=1,9), NI,NJ,NK
      READ (J2,'(7(6E12.5,/),2E12.5,/,3(6E12.5,/))') (fisg(K),K=1,62)

      DO I = 1,NI
              READ (J2,'(6A4,3X,F9.6,2E12.5)') (X(L),L=1,9)
              READ (J2,'(7(6E12.5,/),E12.5)') (X(L),L=22,107)
              IF (X(7)-1.0.GE.0) THEN
                  READ (J2,'(7(6E12.5,/),E12.5)') (X(L),L=108,193)
                  DO L = 10,21
                      X(L) = 0.0
                  END DO
              ELSE
                  DO L = 108,193
                      X(L) = 0.0
                  END DO
                  READ (J2,'(6E12.5)') (X(L),L=10,21)
              END IF
              do j=1,kmatg
                 if (imatg(j).eq.i) then
                   do l=1,193
                     satzq(l,j)=x(l)
```

```
!*********************************************************************
! builds interal compact library for this job
!
! called by StoreTispecAndTn4Data (Tispsub.f90)
!*********************************************************************
      subroutine StoreTispecData (kmatg,imatg,lrg,nrg,lsg,nsg,          &
     &                            libkart,fisg,satzq,satzr,satzs)

        use mgt_io_units, only: itispi

        implicit none

!---- variables from parameter list
        integer  (kind=4)                           :: kmatg,lrg,nrg,      &
     &                                                 lsg,nsg
        integer  (kind=4), dimension(kmatg)         :: imatg
        real     (kind=4), dimension(62)            :: fisg
        real     (kind=4), dimension(193,kmatg)     :: satzq
        real     (kind=4), dimension(lrg,nrg)       :: satzr
        real     (kind=4), dimension(lsg,nsg)       :: satzs
        character(len=88), dimension(-1:nrg+nsg) :: libkart

!---- local variables
        integer(kind=4)                             :: k,l          ! loops


        call StoreHeaderData (nrg,nsg,libkart)

        call StoreEnergyData (fisg)

        call StoreXSectionData (kmatg,imatg,satzq)

        call StoreResonanceData (nrg,lrg,nsg,libkart,kmatg,imatg,satzr)

        call StoreScatteringData (nrg,lsg,nsg,libkart,kmatg,imatg,satzs)

      end subroutine StoreTispecData
```

Figure 1 : Subroutine *tispecrl* before (left, excerpt) and after the refactoring process (right).

While refactoring is clearly a bottom-up process, designing completely new software is generally a top-down approach. The latter means forming a high-level structural picture of the code and then moving down from that high-level perspective to the details of implementation.

A crucial step of a successful reimplementation of existing legacy codes is a critical refactoring process carried out by a team of scientists and software engineers.

2.3 Version Management

In former times source code was exchanged between different developers by email or portable data storage devices. Because in a team different people are working on different code parts at the same time, this approach can lead to code fragmentation. In addition, descriptions of the modifications to the code may not be accessible by all team members during the process of forwarding modified code parts again and again without a central storage. For example it is supposed that over a dozen different versions of the code THERMIX is currently in use by different researchers around the world.

This is why a version management system is an essential part of modern code development. Version control not only inlcudes the source code but all documents belonging to the project, i.e. documentation, batch-scripts etc. The whole project is maintained in a central or distributed repository of files with monitored access. Each member of the developer team checks out a local copy of this repository in which local changes can be made.
The modified files can be uploaded to the repository and are then available for all other team members. The revision control system automatically traces who has made changes on which files and when they were uploaded along with comments on why the changes were made.
Conflicts due to the distributed collaborative development, e.g. two developers change the same file at the same time, are automatically recognized by the system and can be solved by the contributors.

Tracking all revisions of the documents brings the benefit of being able to restore old versions of the software and compare them to the current version, e.g. in terms of performance or bug tracking (see. Fig. 2).
This becomes essential when a specific code version is shipped to customers. A version management system guarantees that this specific version of code which was ditributed for example half a year ago can exactly be reproduced at a later stage of development.
A lot of tools are available for a number of operating systems to manage file repositories effectively. Well-known revision control systems are for example Subversion (SVN) [3] which is a centralized system with only one master copy of the project or Git [4] which is a distributed system.

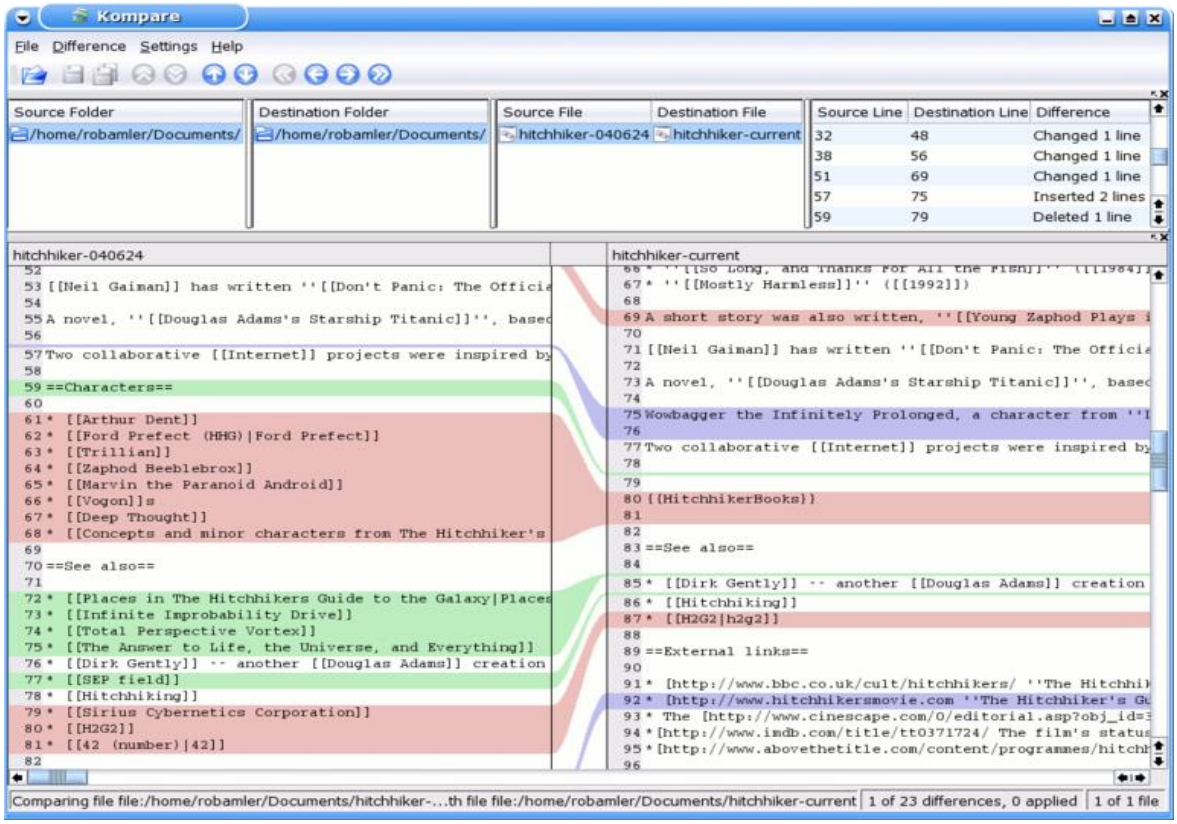

Figure 1: Tools like *Kompare* help to track changes throughout different code versions. Source : http ://www.oss-watch.ac.uk/resources/versioncontrol.

Applying a version management system for code development in a group of people is mandatory to avoid code fragmentation and to preserve the verification and validation strength of a code.

2.4 Coding Conventions

Coding conventions are a general set of rules obeyed by any member of the development team to help improve the readability of their source code and make software maintenance easier. To put it simply : Coding conventions ensure that a code system written by a dozen people finally looks like it has been written by a single person. Coding conventions are a set of recommendations and/or rules for a certain programming language concerning the style of programming, i.e. naming conventions, indentation, comments and other programming practices. All software developers should be advised to follow these rules because they help to increase the quality and reduce the costs of maintenance for software.
In a developer team, coding guidelines should be provided to the team members in a formalized, documented set of rules, if required divided into different levels of bindingness. For example three types of rules **May**, **Should** and **Shall** rules can be defined, indicated by a corresponding color :

- 

  May rules are advisory rules. They suggest the recommended way of doing things, but to some extend are a matter of taste. Deviations from these rules must not be documented.

- 

  Should rules are strong recommendations. It is expected that they will be followed. To break a should rule, the developer must receive an approval from the project leader and the reason for each deviation must be documented in the file that contains the deviation.

- 

  Shall rules are mandatory requirements. They must be followed in any case.

As an example Fig. 3 shows three examples of rules, each with a different level of bindingness.

NC15 **The prefix "is", "has", "may", "should" shall be used only for boolean variables and methods.**

```
isSet, isVisible, isFinished, isFound, isOpen

logical hasLicense
logical mayEvaluate
logical shouldSort
```

In line with common practice in the C++ development community and partially enforced in Java. Using these prefixes solves a common problem of choosing bad boolean names like "status" or "flag". "isStatus" or "isFlag" simply doesn't fit, and the programmer is forced to choose more meaningful names.

NC7 **Abbreviations of common words listed in a language dictionary should be avoided.**

```
call computeAverage() ! NOT: call compAvg()

command       ! NOT: cmd
copy          ! NOT: cp
point         ! NOT: pt
compute       ! NOT: comp
initialize    ! NOT: init
```

Readability.

NC24 **The term "compute" may be used in methods where something is computed.**

```
valueSet%computeAverage()
matrix%computeInverse()
```

Give the reader the immediate clue that this is a potentially time-consuming operation, and if used repeatedly, he might consider caching the result. Consistent use of the term enhances readability.

Figure 3: Three different kinds of rules in a set of coding conventions.

To ensure the readability and maintainability of a code developed in a group of people, coding conventions should be agreed upon by all team members.

2.5 Build Management

The proper way of generating reproducable and well-defined executables for different platforms is called build management. Build management defines the steps which are needed to create an executable from the source code, mainly compiling and linking. Concerning large software projects with many source files, this can be a complicated and time-consuming task if it is done by hand. In addition, different compiler flags used by different people in the development group can lead to problems or different results during the execution of a code.
Therefore, the build process should be automated, e.g. with the tool GNUMake [5]. For using Make, a so called *makefile* must be written. The *makefile* contains rules for the automated compiling and linking of the source files. Make automatically figures out which files have been updated so that only those are recompiled. It can be used for any programming language. Make can be used not only for the build process but also to perform other tasks, such as installing a package or running tests or generate documentation.
For cross-platform development however Make is not flexible enough, e.g. in terms of handling platform-dependent compiler flags (see [6]). A well known tool to overcome this is *Cmake* [7].

CMake is a cross platform open source program to build software in a platform independent fashion. It supports the native build environments such as Make, Apple's Xcode, and Microsoft Visual Studio. It has minimal dependencies, requiring only a C++ compiler on its own build system. But it also supports FORTRAN source code and is capable of building a program which is a mixture of C, C++ and FORTRAN programming language. CMake supports in-place and out-of-place builds. With out-of-place build, all the build files are placed in a single folder which may be present outside the source tree. So if the build directory is removed, the source code still remains unaffected. Complicated directory hierarchies and applications depending on several in-build as well as third-party libraries are well supported by CMake.

Each build project contains a *CmakeLists.txt* file which controls the build process of the program. The build is carried out in two steps. First the standard build files are created from *CmakeLists.txt* file(s). Then the native build tools compile these build files. With the integration of CMake in a large project, the program is build on Linux and Windows operating systems without any need to change the actual code. If the CMake's out-of-place build feature is used, the user can work on the code in his/her personal environment of choice.
In Linux, the compilation of the code is faster and the compilation messages are more readable than the self-made makefile messages. There is a progress status which shows the actual progress of the building process. Different types of messages are colored differently in order to increase their readability and grouping (see Fig. 4).

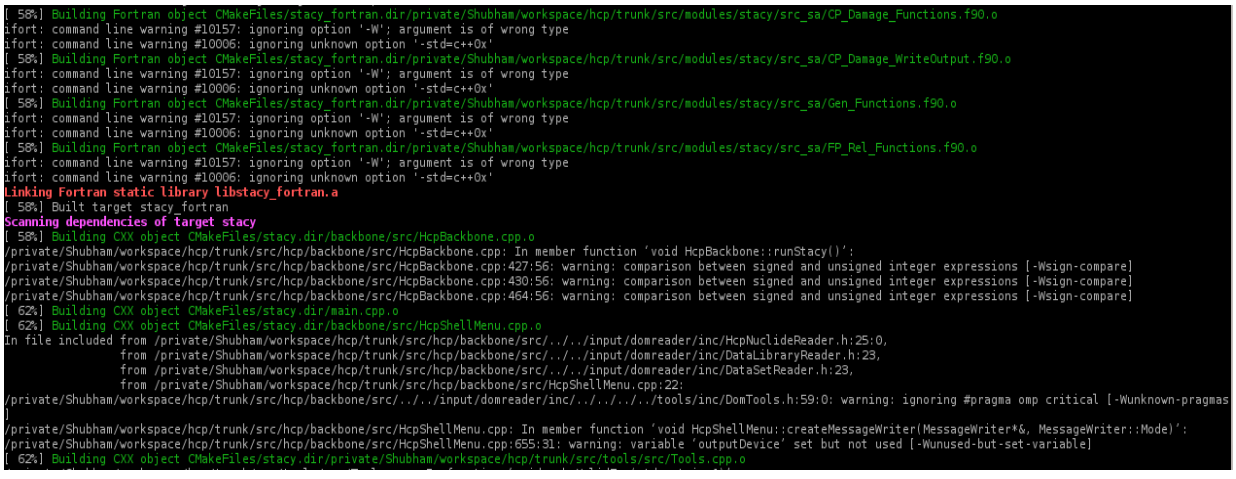

Figure 4 : Excerpt from a run of CMake in the HCP project.

Using a build system is an essential part of creating high quality executables on different software platforms.

2.6 Code Documentation

A proper documentation is essential for the quality of a software project. Code documentation not only includes comments within the source code but also external documentation like a developer guide, a user manual, auto generated source code documentation, UML diagrams and so on.

The developer documentation includes documentation of the code architecture and design as well as inline code documentation. The user documentation mainly consists of user manuals and/or tutorials which assist the user in the application of the software.

In terms of architecture and design, a widespread standard for the modeling of software is *UML (Unified Modeling Language)*. It was developed by the *OMG consortium* [8] and provides different types of diagrams to describe the structure, behaviour and interaction of software components. One diagram type, for example, are class diagrams. They describe a system's class-structure, the classes' attributes and functions as well as the relationship between them (see Fig. 5).

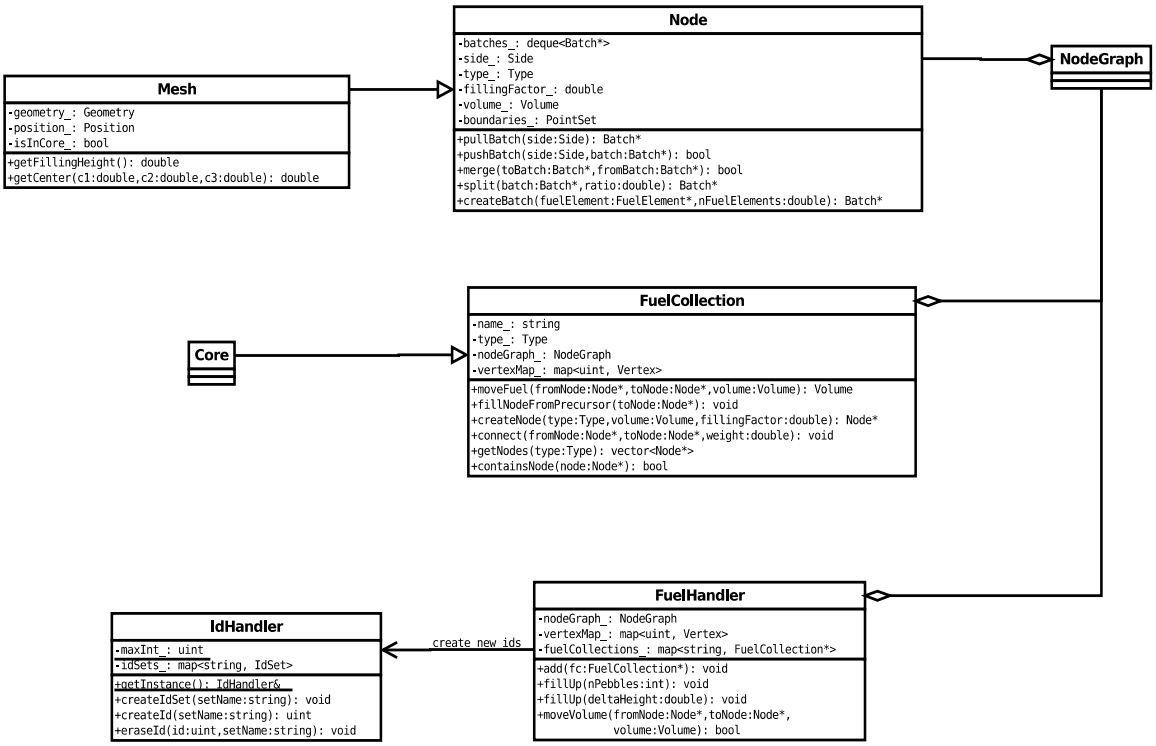

Figure 5 : Example for an UML diagram (excerpt) to define different objects in a nuclear reactor model.

Sequence diagrams are an example of interaction diagrams. They indicate the communication between different components as well as their life cycles (see Fig. 6).
Inline code documentation is very important to describe the general functionality and the intention of code segments. Each method or variable should at least come along with a short explanatory comment.
Doxygen [9] is a standard tool for the automated processing of code documentation into different formats. It generates, for example, online HTML-documentation (see Fig. 7) or offline reference manuals (in LateX, PDF, Unix man pages and other formats) directly from a given set of doxygen-commented source files. Besides, it is also possible to include pieces of source code into the documentation and to automatically generate dependency graphs, inheritance diagrams and collabration diagrams with *Doxygen* which can be helpful to get a general overview over large software projects.
To be able to process source code with Doxygen, some keywords must be added to the inline documentation to indicate the structure (examples can be found at [10]).

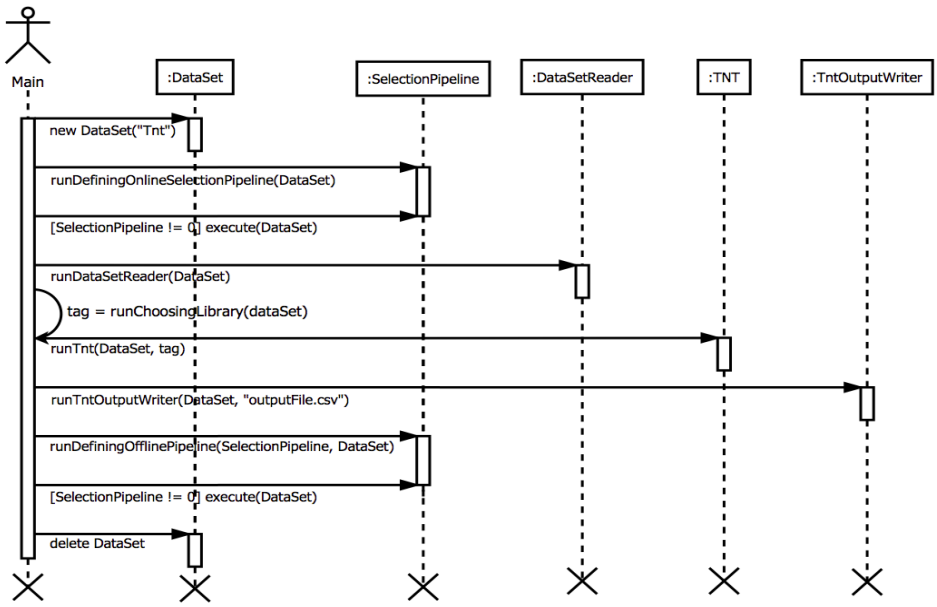

Figure 6 : Example for an sequence diagram as part of the code documentation.

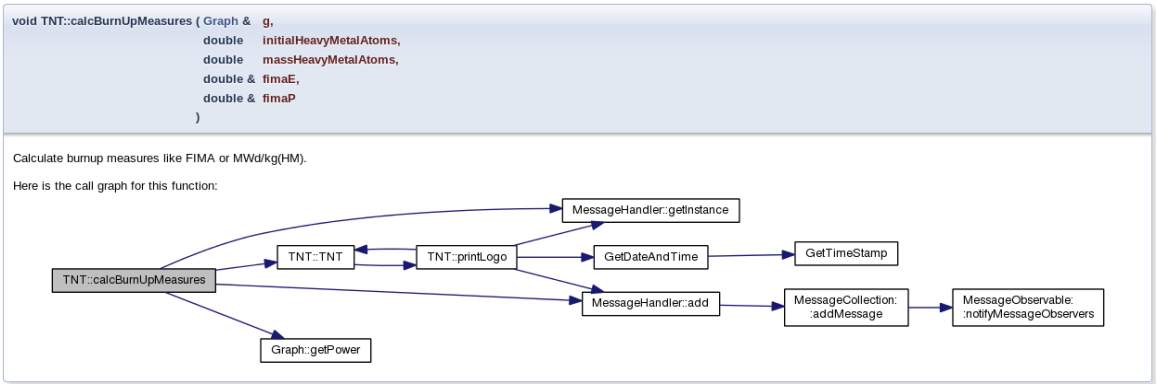

Figure 7 : Automated code documentation with *Doxygen*

Another fairly important part of software documentation is the documentation of decisions made by the developer team. This is important to be able to retrospectively understand the reason for the evolution of software. Decisions can be recorded in a seperate document or in the minutes of the regular project meetings. All project related documents can be stored in a so called *Wiki* page where the developer team and, if desired, also people outside of the project can find, in addition to the current project status, background information, such as presentation slides, and other project-related documents.

Proper documentation is more than commenting source code and writing a user manual. It includes automatic source code documentation, UML diagrams, sample input cases, minutes of code design meetings, developer guides, presentations, papers and all is under version control.

2.7 Integrated Development Environment (IDE)

An *IDE* is a collection of application programs with the intention of poviding quick access and optimal support for all regularly recurring tasks of software developers. Basic components of IDEs are text editor, compiler, linker, debugger and code formating functions. Additional components in more advanced IDEs could be an integrated revision control system, a GUI designer, a project control system or a UML modeling tool.

One well-known IDE is *Eclipse* [12] which was originally developed for Java but based on its plugin system it can be extended to other programming languages, such as C/C++ and Fortran, and to many other different tools. A typical screen while developing software with *Eclipse* can be seen in Fig. 8. In addition to the editor in the middle, it shows the organization of the project files on the left where modified files are marked and the outline of the class currently opened in the editor.

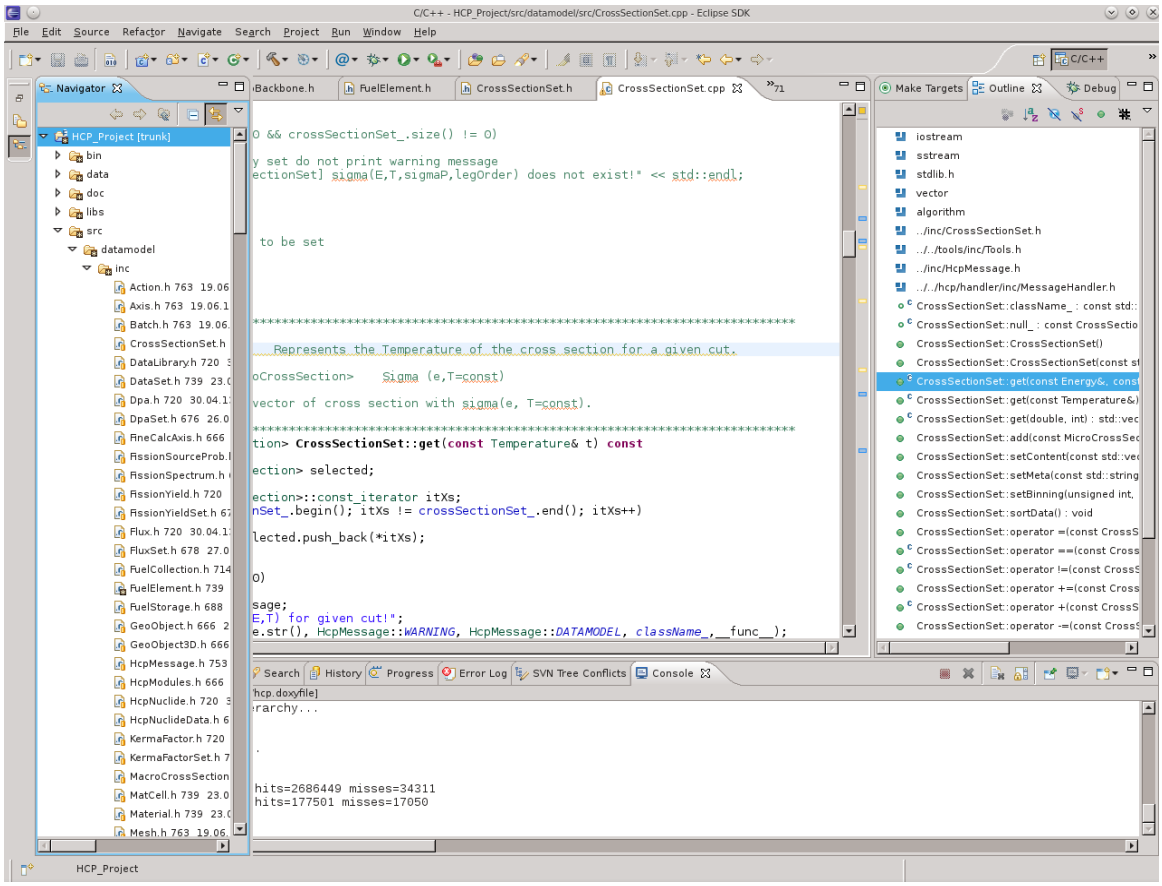

Figure 8 : Example for the IDE Eclipse

Making use of an IDE speeds up the implementation process by providing a toolbox for developers to quickly navigate, compile, test and debug the code in a uniform environment.

2.8 Unit and Regression Testing

Software testing is essential to guarantee a certain level of quality, especially in the field of nuclear safety. It helps to reveal programming errors and hence to verify and validate software. Testing should be applied during the development process as early as possible. These tests are usually performed on each new software revision before it is commited to the repository.

Before starting a test, it is helpful to define test cases which cover all variations of input parameters and all sorts of results when applying them. Defining so called « equivalence classes » is a common technique to reduce the amount of testing. It means, the input data of a software unit is divided into classes which cover a certain type of error. Thus, it is usually sufficient to define one test case for each equivalence class.

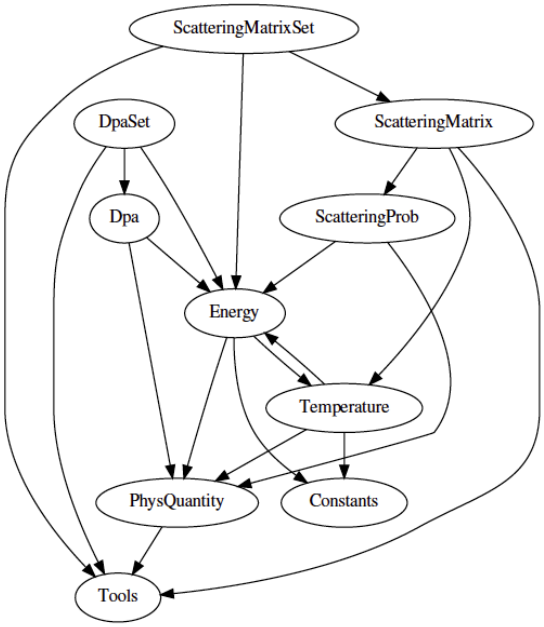

Figure 9 : Example for a file dependency graph used in the HCP unit test framework

As tests can be time-consuming, tests should only be performed for the objects which have been subject to changes. To do so, the unit test framework of the HCP project in a first step analyzes which code had been modified. In a second step a dependency graph of files is created (see Fig. 9). For example, if the class ScatteringMatrixSet is to be tested, all related classes will be tested as well. So it is guaranteed, that changes to one of the classes do not have any side effects to dependent classes.

There are various test types which can be used to test software [13]. Some important ones, especially for continuously evolving projects, are mentioned below.

Unit testing means testing an independent unit of software. It is often done by the programmers themselves to make sure that their piece of code is working correctly. Usually, a single function or even a whole class is subject to different test cases and verified against pre-defined reference results.

Regression tests are used to ensure that a part of software is still working correctly after a modification. Usually, all test cases must be run through again to prove that no unexpected results are produced by any module.

It is helpful to automate the testing process in order to reduce the effort to define, run and evaluate a test by using a test framework.Test frameworks are provided for a huge number of programming languages. *CppUnit*, for example, is a unit-testing framework for C++ [14].

Unit and regression tests are mandatory to assure the quality and validation power of a code during the development process.

2.9 Task Management

Task management is an issue not only for software projects but for projects in general. An effective task management gives project leaders a better overview over the current project status and the particular tasks ahead. It can also help to reveal potential problems and create new startegies for directing the team. With regard to the developers, it supports the optimization of their time-management by providing an individual task-list.
Task management is usually supported by software. *Trac* [11], for example, is an open-source Wiki and issue tracking system with the objective of supporting software developers in organizing their

projects. In addition to an integrated Wiki and Subversion access, it provides a ticketing subsystem which includes issue tracking, project roadmaps, and reports. The definition of milestones and the set up of a project timeline is also possible.

3 Example of Code Integration Aspects

In the following sections, selected examples of the HCP project at FZJ are given which are interesting from a code integration point of view.

3.1 Extended Markup Language (XML) for Data Input Files

To run 3D nuclear reactor simulations, a large volume of input data has to be provided by the user and a great deal of information is supplied by the codes to the user. The recent approach of setting up and printing out huge formatted text files has reached its limits with respect to efficiency and usability. Therefore a data input model based on *XML (Extensible Markup Language)* [15] was developed. The advantages are :

- Higher readability : element and attribute names are almost self-explaining, intrinsic hierachy of data blocks supports easier understanding of relations and dependencies
- Higher flexibility : In general there is no strict order of how the data is provided in an XML file
- Higher safety : data can be checked for correctness by XML schemes before the simulation is started

A more detailed discussion of an XML input concept in the field of nuclear safety research can be found in [17]. Fig. 10 shows a simplified example of the XML data representation of a nuclide as part of the nuclear data input.

```
<Nuclide>
  ...
  <Ids>
    <Id type="Symbol"> U-235 </Id>
  </Ids>
  <HalfLife error="3.155760e+13" unit="s">
    2.221650e+16
  </HalfLife>
  ...
</Nuclide>
```

Figure 10 : XML input example (excerpt) for nuclide properties in the data library.

Besides a better readability of the input, the input structure itself should be changed when integrating separate but related codes into a single code system. Imagine there are three codes as depicted in Fig. 11 (red part). Each of the code deals with a different part of the same fuel type. Let us call this kind of input « program based » input. Now in an integrated code we can either just lump all input files together or we can think about a new concept without any information duplication. This is when we talk about « object-based » input. Here we don´t think from the code point of view but from the object to describe in the simulation which is a fuel type (see Fig. 11, green part). Once read each code part just takes the corresponding part of interest from the input.

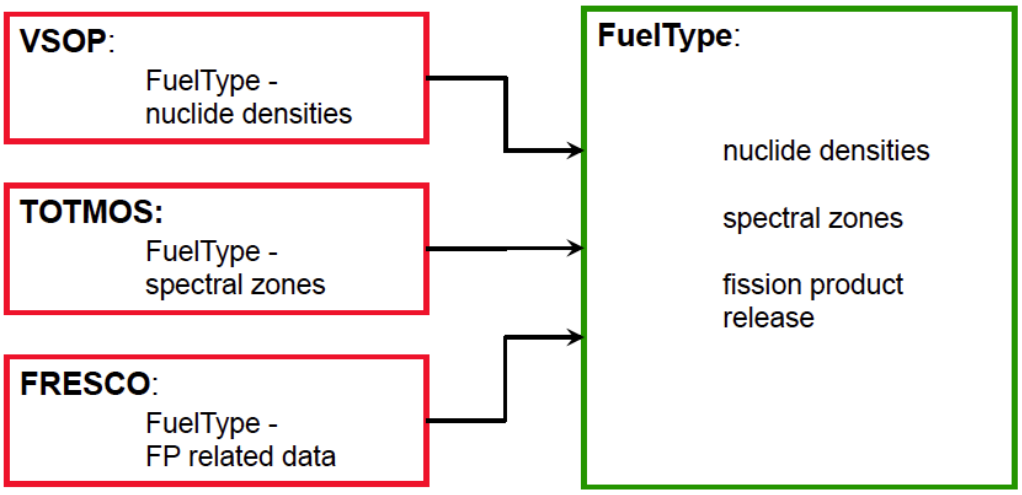

Figure 11 : Program-based (left) and object-based (right) input structure.

3.2 Hierarchical Object Oriented Data Model

Based on the structure of the XML input files an object oriented hierarchical data model was developed. The main focus here is to model the problem in a realistic way by using data objects like *CrossSection, Nuclide, Batch* or *Mesh* etc.

In general, the data model can be split into two parts. The one part (data model I) deals with basic variables in nuclear physics and is used to setup a data library for the nuclear properties. Another part (data model II) is a basis to store data for the reactor model or the time control itself (see Fig. 12).

How such a data model finally is used within the program is presented in figure 13. It can be seen that using problem specific objects increases the readability of a code a lot.

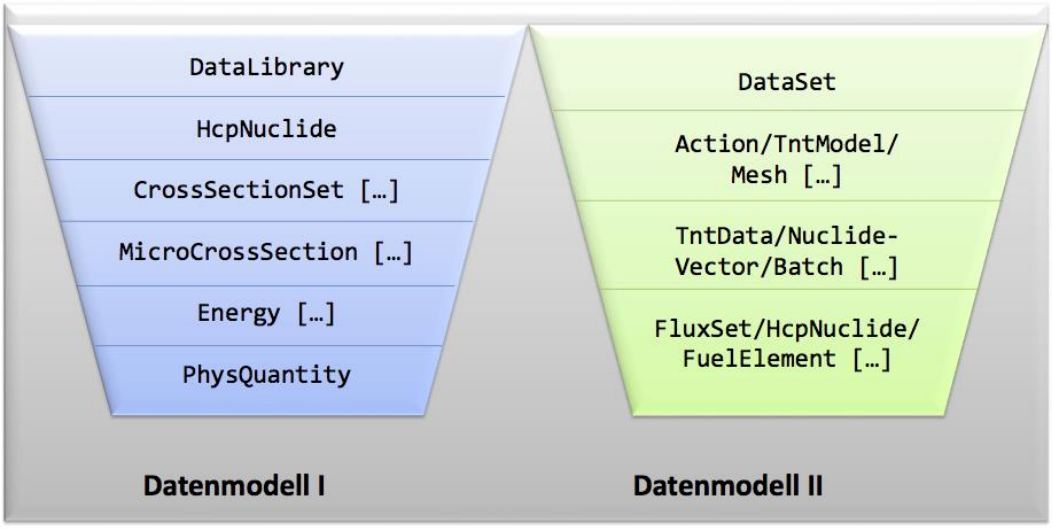

Fig. 12 : The hierachical data model used in HCP.

```
// set up nuclide by Z, A and M
HcpNuclide* u235 = new HcpNuclide(92,235,0);
u235->setSymbol("U-235");

// set half life for nuclide using Time object
Time halfLife(2.221e+16, Time::s);
u235->set(halfLife);

// average energy released in decay
Energy qvalue(4.68, Energy::MeV);

// define alpha decay (100%) into ground state with q value
Reaction reaction(Reaction::ALPHA, Reaction::GROUND, 1.0, qvalue);
u235->add(reaction);

// store nuclide in library, later find it by asking for "U-235"
DataLibrary lib("endfb70");
lib.addNuclide(u235);
```

Figure 13 : Example of how the data model is applied in the simulation codes

3.3 An Interface between C++ and Fortran

One of the major problems of huge simulation codes is data management. In fact, often large parts of the code are dealing with data while only a smaller part of the code contains the algorithms and physics models. This is why new codes dealing with huge amounts of data nowadays are implemented for example in C++ [Quelle CERN]. Nevertheless a lot of codes still in use have been written in Fortran. In some cases it makes sense to rewrite those codes from scratch because this takes less time than updating an non-standard legacy code, even if additional time for new verification & validation work has to be taken into account. In other cases the existing Fortran code is too big for a complete rewrite. Then a refactoring process should be started to bring the code in a shape to be internally coupled to codes written in C++.
Since Fortran 2003, the language standard includes an intrinsic module called *iso-c-binding* (see [16]) for interoperating with C/C++ (see Fig. 14). This module provides corresponding types for all primitive datatypes of C (e.g. `int – c_int`). For pointer-interoperability, the module contains the derived datatype `c_ptr` which corresponds to C-pointers of any type (as pointers are just addresses). Some functions are provided which makes working with pointers easier. There is a function which returns the C-like address of a `c_ptr` type (`c_loc`), a functions for testing if a pointer is associated (`c_associated`) and one for associating a Fortran-pointer with a `c_ptr` type (`c_f_pointer`).
It is also possible to make derived types interoperable by defining a derived Fortran-type with an additional `bind(C)` attribute and a corresponding struct in C.
For the interoperability of procedures, it is necessary to declare them in both languages along with the `bind(C)` attribute in an interface-block in Fortran and in an `extern "C"-block` in C. A Fortran-procedure can then be called from a C program and vice-versa. Arrays are also interoperable using `iso-c-binding`, but only those with fixed- or assumed shape. Allocatable or pointer-arrays are not natively supported. However C-arrays can be treated as pointers and therefore the pointer support can be used.
A huge benefit is that the `iso-c-binding` module handles the different storage order of arrays in C and Fortran internally.

Since the mechanism outlined above provides many standardized tools for interoperating with C, the coupling of Fortran codes to a C/C++ main program can be realized more easily than in the past. Access to data stored in a C/C++-datamodel, for example, can be enabled by gathering the needed data in a struct and defining the equivalent derived type in Fortran. Then, the derived type can be exchanged via an interface-function for example.

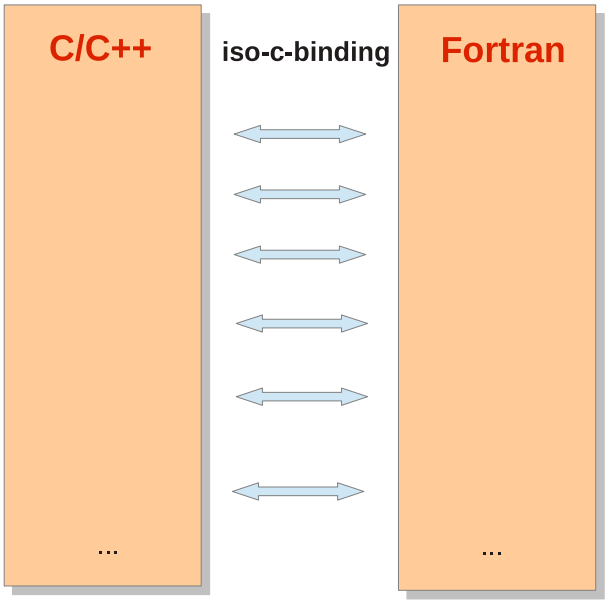

Fig. 14 : Overview of relations between C(++) and Fortran when using iso-c-binding.

4 Literature

[2] FOWLER, Martin: *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA : ADDISON-WESLEY, 1999. - ISBN 0-201-48567-2

[3] URL : http ://subversion.apache.org/, Last Access : 06/28/2013

[4] URL : http ://git-scm.com/, Last Access : 06/28/2013

[5] URL : http ://www.gnu.org/software/make/, Last Access : 06/28/2013

[6] URL : http ://en.wikipedia.org/wiki/GNU_Make, Last Access : 06/28/2013

[7] URL : http ://www.cmake.org/, Last Access : 06/28/2013

[8] URL : http ://www.uml.org/, Last Access : 06/28/2013

[9] URL : http ://www.stack.nl/~dimitri/doxygen/, Last Access : 06/28/2013

[10] URL : http ://www.stack.nl/~dimitri/doxygen/manual/docblocks.html, Last Access : 06/28/2013

[11] URL : http ://trac.edgewall.org/, Last Access : 06/28/2013

[12] URL : http ://www.eclipse.org/, Last Access : 06/28/2013

[13] URL :http ://www.codeproject.com/Tips/351122/What-is-software-testing-What-are-the-different-ty, Last Access : 06/28/2013

[14] URL :http ://sourceforge.net/apps/mediawiki/cppunit/index.php?title=Main_Page, Last Access : 06/28/2013

[15] URL : http://www.w3.org/XML/, Last Access : 06/28/2013

[16] URL : http://www.fortran.bcs.org/2002/interop.htm, Last Access : 06/28/2013