Reusing Code from FLUKA and GEANT4 Geometry *

M.Campanella, A.Ferrari, P.R.Sala, S.Vanini
INFN sez. di Milano, via Celoria 16, I-20133 Milano, Italy.

October 13, 1998

Abstract

The aim of the project is to investigate the feasibility of integrating the GEANT4 geometry facilities with FLUKA physics treatment, and eventually release a production application. This will allow to perform cross-checking of physics results, using completely independent physics packages (GEANT4 and FLUKA) tracking in the same geometry. This note summarises the basic requirements of Multiple Language Programming (with a special focus on the problems involved in integrating C++ and FORTRAN code). It also outlines the structure of the new application, and some positive results.

1 MLP: Multiple Language Programming

Reusing components means integrating in the same application components already coded, possibly in different programming languages. This may be a good alternative among recreating components or translating old ones. The following observations show the advantages ([1]).

- Components may be best coded in a particular language, depending on the task; indeed every language is created for a particular environment (e.g. Java for the network).
- Translating existing code is not straightforward, in particular:
  - due to the different characteristics of each language, a particular task may be ill suited to a particular language;

*Talk given at the "II Workshop INFN sul Software e Calcolo moderno", Perugia: 15-17 June 1998

- when viable, translating is difficult, and it easily introduces bugs;
- the users' and developers' background have to change radically. Reusing components allows a parallelism between the introduction and acquisition of different technologies, the development and maintenance of the whole of the code;
- the source code of the component may not be available.

1.1 Problems of Multiple Language Programming

The integration of components written in different languages cannot refer to any standard, and generally is not supported. The problems involved in integrating foreign routines ([1]) concern at least the following aspects:

- the name of the component is not obvious;
- passing input and output data may be complex;
- any component side-effect potentially causes environment troubles (anyway, is a good programming practice to avoid side-effects by *encapsulation*: e.g. substituting error messages with flags,etc.).

1.2 Types of Multiple Language Programming

Multiple Language Programming could be performed at different levels. The most common types of MLP are:

- **coarse-grained**: different executables (each coming from a component written in a particular language) work on a intermediate file of data;
- **CORBA = Common Object Request Broker Architecture**: communication between components (either on the same or between different platforms) is performed by the Interface Definition Language (IDL);
- **MLX = Multiple Languages Executables**: this is the most common form of MLP, even if, except for C++, few programming languages provide standard support ([2]). There are different types of MLX:
  1. *direct MLX*: for each foreign routine there is a routine-dependent call in the main code (this is platform-dependent!);
  2. *wrapper*: it provides a native interface to the foreign routine (but the use of the foreign routine via the wrapper isn't always platform-independent!). A wrapper scheme is shown in Figure 1.

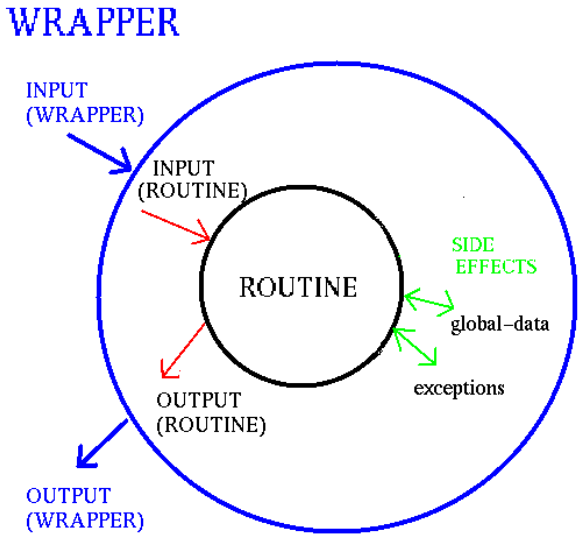

Figure 1: Wrapper scheme.

1.3 Wrappers for FORTRAN and C++

Mixing modules written in C++ with modules written in FORTRAN is not straightforward, due to the formal differences between the two languages ([3] and [4]). Such formal differences depend on the platform too. In particular for HP-UX these differences comprise the following points:

- **Routine names**: when compiling FORTRAN code with the `+ppu` option, an underscore is appended to the routine name. So, on HP platform, it is necessary to redefine, in C++ code, all routines called from FORTRAN code. For example:
```
#ifndef g1wr
#define g1wr g1wr_
```

- FORTRAN is not **case sensitive**, while the C++ compiler is. Therefore, all C++ global names accessed by FORTRAN routines must be lowercase. All FORTRAN external names are down-shifted by default on HP platform.

- **Function arguments:** there are two main methods of passing arguments: by reference or by value. Passing by reference means that the routine passes the argument address rather than the argument value. It's imperative to ensure

that both caller and called functions use the same method of argument passing for each individual argument. Furthermore, the calling convention for the order of arguments must be known. For maximum compatibility and portability, only simple data types should be passed to routines.

All C++ parameters are passed by value, except arrays and functions, which are passed as pointers (so they are actually passed by reference). FORTRAN passes all arguments by reference. So, the simplest way to reconcile these differences in argument-passing conventions is to use reference variables in C++ code, declaring all non-array formal arguments to be passed by reference; array arguments could be declared as arrays - or pointers (C++ converts a use of an array to a pointer to the array first element).

- **Arrays**: C++ stores arrays in row wise, whereas FORTRAN stores arrays in column wise (so matrixes are the transposed one to the other).
- **Array indexing**: The lower bound for C++ arrays is 0. The default lower bound for FORTRAN is 1. It's necessary to shift the array index going from a language to the other.
- **Strings**: C++ strings are not the same as FORTRAN strings. In FORTRAN the strings are not null terminated. Moreover, strings are passed as string descriptors in FORTRAN .
- **Boolean type**: C++ and FORTRAN do not always share a common definition of TRUE or FALSE. Sometimes C++ compiler does not have a FORTRAN LOGICAL type. Instead, C++ uses integers: any nonzero value is used to represent TRUE and 0 is used to represent FALSE.
- **File access**: FORTRAN I/O routines require a logical unit number to access a file, whereas C++ accesses files using subroutines and intrinsic functions, and requires a stream pointer. A FORTRAN logical unit cannot be passed to a C++ routine to perform I/O on the associated file. Nor can a C++ file pointer be used by a FORTRAN routine.

2 Interfacing FLUKA and GEANT4

Keeping in mind that reusing components is a simple and fast way to build new applications, an interface is being developed to allow the FLUKA MonteCarlo (written in FORTRAN) to call the geometry routines of the alpha version of the object-oriented HEP simulation software GEANT4 (written in C++). This new application

uses FLUKA to generate physical events and GEANT4 to handle geometry propagation of particles in the detector geometry. The key features of this operation are listed below.

- To simulate particles behaviour using all FLUKA physics treatment (like interaction models, charged particles tracking, biasing, etc.).
- To allow the transport code FLUKA to describe complex detector geometry, and to compute particle steps and particle location in the detector, by means of GEANT4 routines ([5]). This will allow to exploit a broad-range of facilities available in GEANT4 (like complex geometry handling, exchanging detector geometries with Computer Aided Design - CAD, detector and tracks visualisation, etc. ; see [6], [7]).
- To have two MonteCarlo simulations (FLUKA and GEANT4) within the same geometry description (built in GEANT4), in order to the cross - check simulation results, using completely independent physics packages.

2.1 FLUKA

FLUKA ([8]) is an *interaction and tracking MonteCarlo simulation*. It's a fully integrated code: it's an homogeneous code, developed as a whole and not an assembly of various programs. It's a multipurpose code: it can be used in different fields such as shielding, dosimetry, high energy experimental physics and engineering, cosmic ray studies, medical physics, etc., allowing analogue and biased transport. The physics capability include:

- hadron-hadron and hadron-nucleus 0-10 TeV interactions;
- 0-100 TeV μ and electro-magnetic interactions;
- charged particles tracking and energy loss by ionization, pair production and bremsstrahlung;
- multi-group neutrons interactions in the 0-20 MeV range;

2.2 GEANT4

GEANT4 ([9], [10]) is a *Object-Oriented Toolkit* for HEP simulation, now in its beta version. The use of Object Oriented design was employed to achieve the required

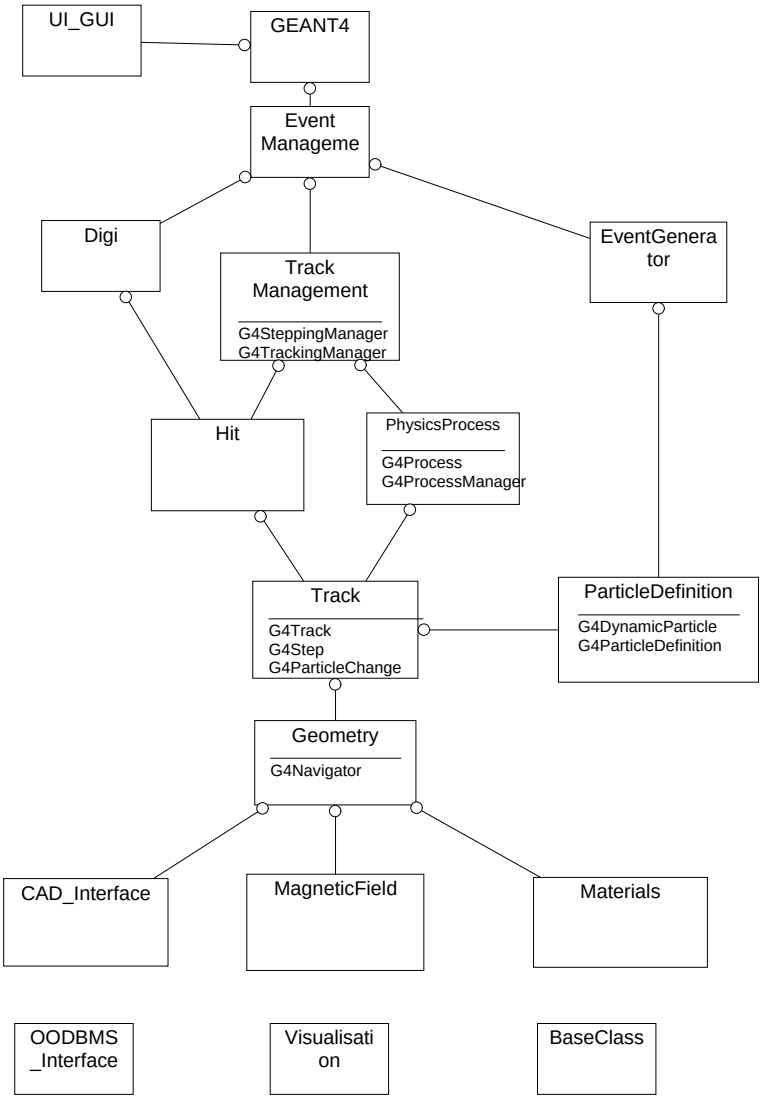

Figure 2: GEANT4 class category diagram.

level of flexibility and extensibility for the user, and the necessary transparency of the physics processes in the simulation. The description of detector geometries in GEANT4 is based on an ISO STEP compliant solid modeller. Logical volumes, physical volumes, and solids, are used to describe the properties of the geometrical structures of the detector. The class category diagram shown in Figure 2 is the result of the problem domain analysis for GEANT4. The physics capability of GEANT4 are not addresses here as they are outside the aim of this project.

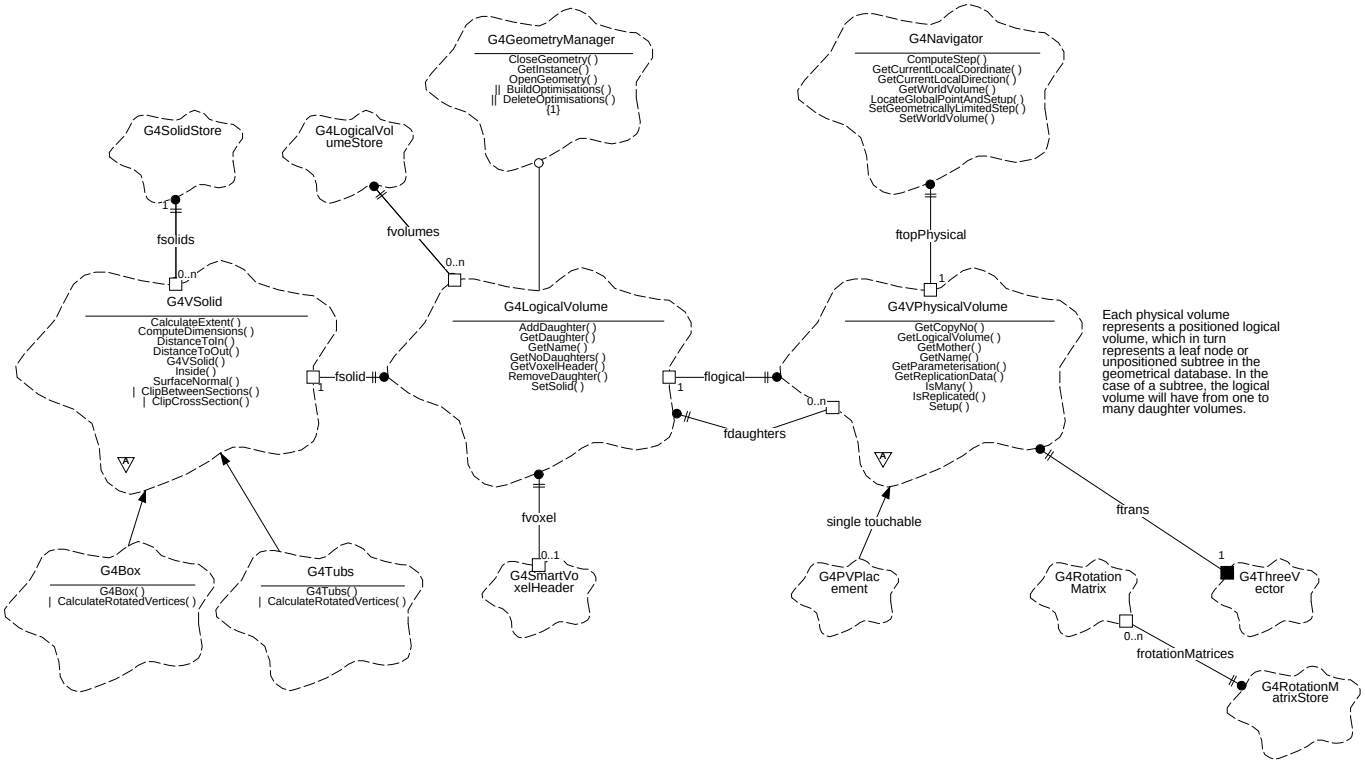

Figure 3: GEANT4 geometry classes.

2.3 Geometry Calls

The first step of the project was to find out the feasibility of decoupling physics and geometry in both FLUKA and GEANT4 packages. To achieve this goal, we identified and analysed each different FLUKA call to geometry subroutines, and we found that these are completely disjoin-ed from physical problems. Then we looked

for analogous geometry routines in GEANT4 and we discovered that these belong to G4Navigator class, and they are independent from physics too ([5]). So we decided to interface the programs at this very low level, *wrapping* into FLUKA code only GEANT4 classes dealing with the geometry tracking (these are shown in Figure 3).

The relevant FLUKA geometry calls are few, summarised in the following, together with their GEANT4 equivalent:

- **Tracking call**: it is the call for the control routine for combinatorial geometry named G1FLU, that calculates the distance travelled in the present zone/region and the number of the next zone/region to be entered by the particle. The analogous GEANT4 function is ComputeStep. G1FLU returns the important parameter DSNEAR, its value is less than or equal to the minimum of the distances between the particle and each boundary. Calculating such a parameter is very useful, because as long as the subsequent steps are shorter than DSNEAR, the program does not call the geometry at all, keeping all the information computed for the previous step. It is very important that ComputeStep returns a parameter that performs the same action (safety).

- **Look for region number calls**: various calls for geometry routines (LOOK*) which return the index of the region to which the particle belongs, according to the current position and direction, in particular situations (starting new track, error conditions, magnetic field, etc.). GEANT4 function LocateGlobalPointAndSetup performs similar actions.

- **Computing the normal to a boundary call**: this is a call for the NORML routine that returns the unit vector at the previous point of the tracking (in case a boundary crossing occurred), that is the intersection point between the path and the preceding boundary. The analogous GEANT4 function is getLocalExitNormal.

Obviously the input/output of GEANT4 member functions doesn't exactly match FLUKA routine input-outputs. A set of "wrappers" takes care of the problem.

2.4 Wrappers

When mixing C++ modules with FORTRAN modules, extern "C" linkage must be used to declare any C++ functions that is called from a non-C++ module and to declare the FORTRAN ones. So the wrappers must be defined like this:

```
extern "C" void g1wr(...);
```

Wrappers solve the formal differences between languages (arrays are translated, routine names redefined, etc.), and they adapt inputs and outputs of GEANT4 functions to make them match the FLUKA ones exactly. We could summarise wrapper functions as follows:

- adaptation of **input and output**:
  - **physics units** must be the same;
  - **data types** in the different languages must correspond (e.g. we had to convert double x[3] to G4ThreeVector pos(x,y,z);
  - **variable form**: the output data passed to FLUKA must match the exact request (e.g. FLUKA requires the number indicating a region, whereas GEANT4 employs volume pointers...)
- achieve formal compatibility between languages.

2.5 Main Program

In general, when mixing C++ modules with modules in FORTRAN the overall control of the program must be written in C++. In other words, the main() function must appear in some C++ module and no other outer block should be present. We wrote the C++ main, where:

- FLUKA is declared as an external routine with "external linkage":
  ```
  extern "C" void flukam(const G4int & GeoFlag);
  ```
- instances are made of those classes whose member functions are called from FLUKA (calling member functions of the class FGeometry - written for this purpose).
- FLUKA is called as a subroutine.

The executable file is created through a makefile. In this makefile, objects are created for GEANT4 classes used by FLUKA, and the FLUKA library is included. All these objects must be linked with the lisamstub library.

```
CC -lisamstub prog_CC.o prog_FORTRAN.o
```

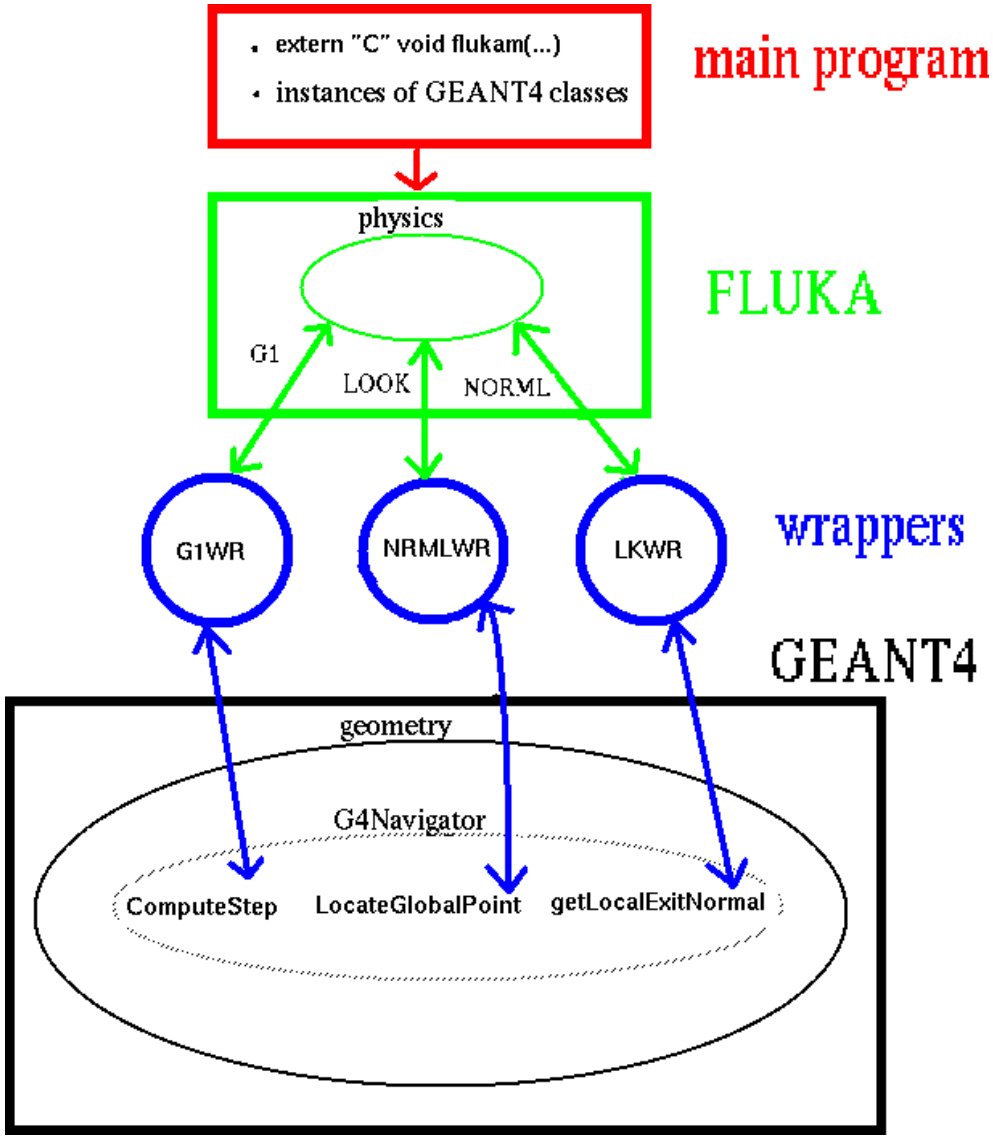

Figure 4: Application diagram.

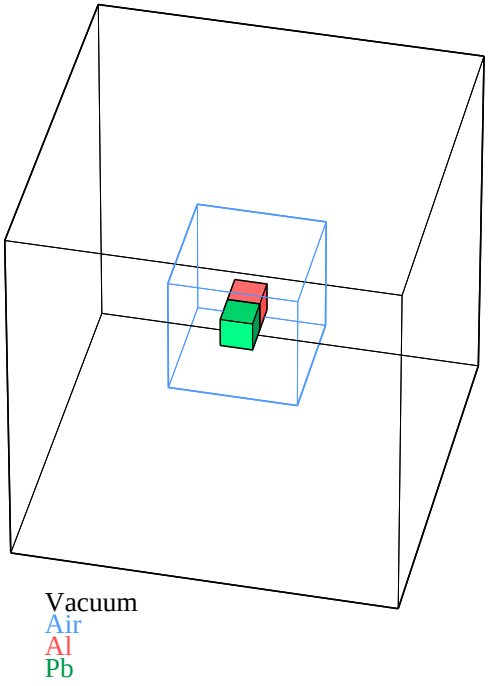

Figure 5: A simple geometry for testing the application.

3 Preliminary Results

3.1 Testing the application

After concluding the first draft of the application (its scheme is shown in Figure 4, we began testing the program, starting from the simplest cases.

1. Ray, neutrons, muons and electrons tracking in a simple geometry with boxes of Pb, Al, air (Figure 5).

2. Electrons tracking in a more complex geometry: a series of slab of different elements (Al, Au, Al), which total thickness is of the order of the electron range. As shown in Figure 6, electrons scatters in all directions. In this case the problem to face with concerns multiple scattering at boundaries, where a very good interface between geometry and transport is needed to handle delicate situations such as grazing angles, backscattering, deflections at boundaries.

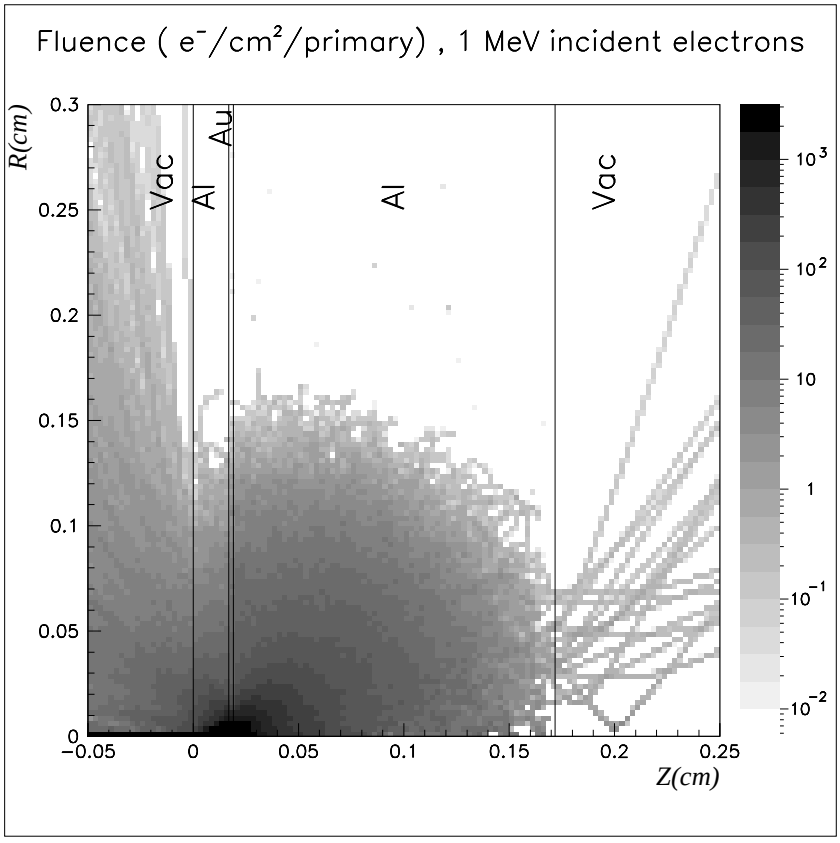

Figure 6: Electron flow in the detector - the "colour" is proportional to electron flux.

3. 500 MeV protons on a thick Cu target: the generated neutrons propagate in a concrete shield. In this case biasing techniques are applied to achieve variance reduction: particles are randomly splitted (cloned) at region boundaries according to user defined parameters in order to maintain a constant number of particles throughout the shield. To ensure the conservation of observable, the weight of splitted particles is reduced. From the geometry side, this requires continuous relocalization of particles sitting exactly on boundaries. An example of splitting at boundary is shown in Figure 7. It should be noted that the variance reduction is available in FLUKA, but not (yet) in the alpha version of GEANT4. As the figures 8 and 9 show, when simulating with biasing the percentage error on the neutron dose is constant through the shield depth. Without biasing, the percentage error grows exponentially with the thickness of the shield.

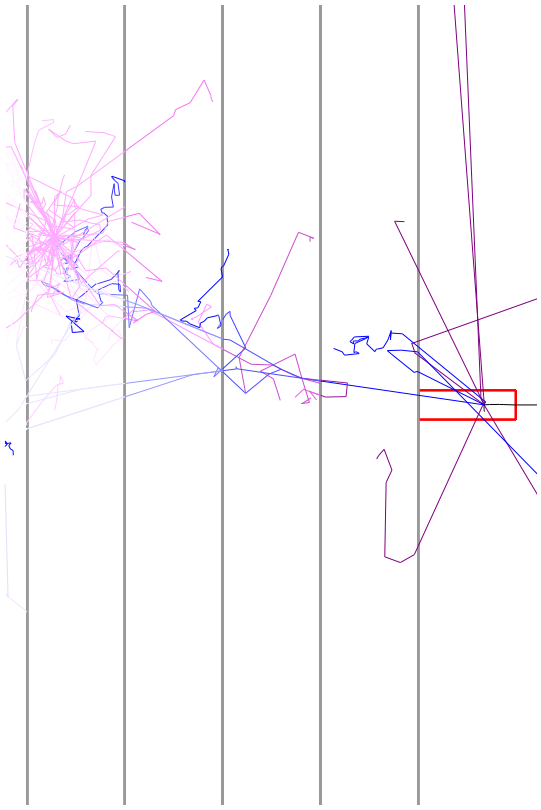

Figure 7: Particle tracks in concrete shield.

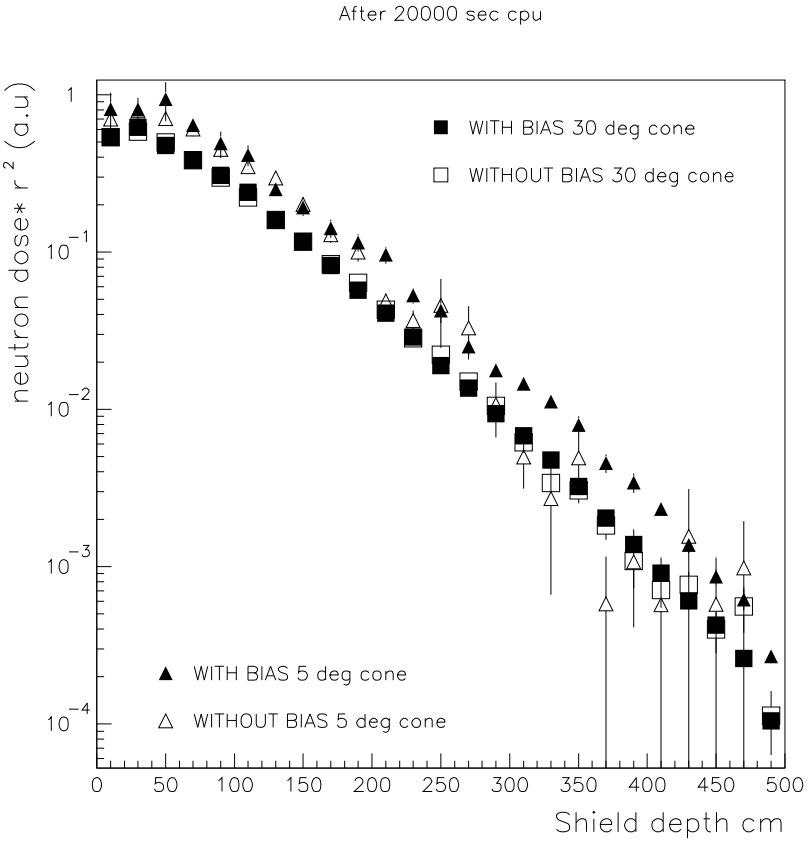

Figure 8: Neutron dose versus shield depth.

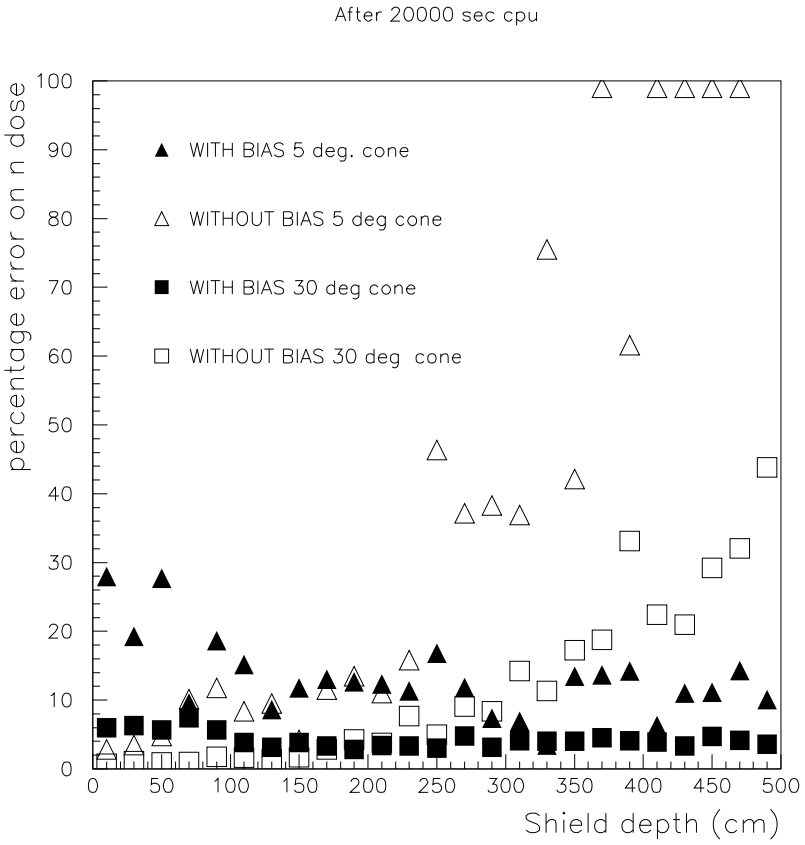

Figure 9: Percentage error on neutron dose.

3.2 CPU

We made a first comparison between the CPU time used running both FLUKA and the new application FLUKA+GEANT4. We tracked 100 primary particles in the simple geometries described before, and we obtained in the two cases **identical events**, i.e. events that start and end with the same random number and which are step by step equivalent in all quantities. Furthermore, as shown in the table, the average cpu time for each beam particle (in seconds) is almost the same for both runs (program initialisation and ending times are not taken into account in the computation of the time per event)!

	FLUKA	FLUKA+ GEANT4
"ray" (boxes)	$3.100 \cdot 10^{-3}$	$3.100 \cdot 10^{-3}$
n (boxes)	$1.124 \cdot 10^{-1}$	$1.053 \cdot 10^{-1}$
μ **(boxes)**	$6.980 \cdot 10^{-2}$	$7.100 \cdot 10^{-2}$
e^- **(boxes)**	$9.360 \cdot 10^{-2}$	$7.880 \cdot 10^{-2}$
e^- **(Al-Au-Al)**	$9.520 \cdot 10^{-2}$	$8.710 \cdot 10^{-2}$
n (concrete)	$3.991 \cdot 10^{-1}$	$4.497 \cdot 10^{-1}$

4 Future Developments

The activities for the coming months concern:

1. testing the code in more complex and critical cases (e.g. complex geometries with "replicated" and "parameterised" volumes, tracking in magnetic field, etc.);
2. visualising detector geometry and tracks with GEANT4 visualisation drivers;
3. installing the GEANT4 geometry debugger (DAVID);

4. storing in FLUKA stack history information (the position in geometry hierarchy) with created secondary particles. Giving the history back to GEANT4 Navigator on starting tracking of secondaries, will avoid recalculating it every time.

5. translating GEANT4 exceptions in FLUKA error flags;

6. mingling the material specification from FLUKA and GEANT4 input;

7. allowing the application to use some of the facilities of GEANT4 user interface (e.g. set visualisation parameters, etc.);

8. improving the application, for example by creating a C++ class (a sort of wrapper manager) with the wrappers as member functions, that instantiates all the classes needed.

9. adapting the application to run on several platforms.

5 Conclusions

We have shown that the project is feasible, and we built a first version of an application including FLUKA physics and the alpha version of GEANT4 geometry, with wrappers that encapsulate GEANT4 functions for making them callable from FLUKA physics routines. The first tests with simple geometries, and the comparison of CPU times are really encouraging. The project is therefore continuing in these months using the beta release of GEANT4 and improving the application as described in section §4.

6 Acknowledgements

We acknowledge the contribution of Laura Perini who participated in fruitful discussions and provided encouragement for this work.

References

[1] B.D.Burow, “Mixed Language Programming”, parallel paper CHEP 95

[2] J.J.Barton, and L.R.Nackman, “Scientific and Engineering C++ - an Introduction with Advanced Techniques and Examples”, Addison-Wesley Publishing Co., 1995

[3] B.Stroustrup, “The C++ Programming Language”, Addison-Wesley Publishing Co., 1997

[4] I.Pohl, “Object-Oriented Programming Using C++”, TheBenjamin/Cummings Publishing Company, Inc., 1993

[5] P.Kent, “Pure Tracking and Geometry in GEANT4”, April 1995 (unpublished)

[6] J.Apostolakis, “An Overview of GEANT-4’s Geometry”, RD44 collaboration, IT division, 28 April 1997

[7] P.Kent,S.Giani, “The GEANT4 Geometrical Model”, 23 April 1995

[8] A. Fassò, A. Ferrari, J. Ranft and P.R. Sala, Proceedings of the *IV International Conference on Calorimetry in High Energy Physics*, La Biodola (Elba) 1993, World Scientific, p. 493 (1994) ; A. Fassò, A. Ferrari, J. Ranft and P.R. Sala, Proceedings of the 3rd workshop on “Simulating Accelerator Radiation Environment”, SARE-3, KEK-Tsukuba, May 7–9 1997, H. Hirayama ed., KEK report Proceedings 97-5, p. 32 (1997); A. Ferrari, and P.R .Sala, Proceedings of the *International Conference on Nuclear Data for Science and Technology*, NDST-97, International Centre for Theoretical Physics, Miramare-Trieste 1997, SIF Atti e Conferenze, p. 247, Vol. I (1998) , and references therein.

[9] CERN/LHCC/97-40 “GEANT4: an Object-Oriented Toolkit for Simulation in HEP”, 1997

[10] CERN/LHCC/95-70 “GEANT4: an Object-Oriented Toolkit for Simulation in HEP”, 18 October 1995