Testable physics by design

Chansoo Choi[1], Min Cheol Han[1], Gabriela Hoff[3], Chan Hyeong Kim[1], Sung Hun Kim[1], Maria Grazia Pia[3], Paolo Saracco[3], Georg Weidenspointner[4]

[1] Hanyang University, Seoul 133-791, Korea
[2] CAPES, Brasilia, Brazil.
[3] INFN Sezione di Genova, Genova 16146, Italy
[4] Max-Planck-Institut für extraterrestrische Physik, 85740 Garching, Germany

E-mail: `MariaGrazia.Pia@ge.infn.it`

Abstract. The ability to test scientific software needs to be supported by adequate software design. Legacy software systems are often characterized by the difficulty to test parts of the software, mainly due to existing dependencies on other parts. Methods to improve the testability of physics software are discussed, along with open issues specific to physics software for Monte Carlo particle transport. The discussion is supported by examples drawn from the experience with validating Geant4 physics.

1. Introduction

The validation of physics calculations requires the capability to thoroughly test them.

Physics observables calculated by complex software systems are often the result of concurrent effects of multiple parts of the code: for instance, the simulated energy deposition in a detector is the result of several physics processes modeling particle interactions with matter. Establishing confidence in the outcome of the simulation in a variety of experimental conditions requires the capability of appraising the reliability of individual components that contribute to shaping experimental observables, and of quantifying their relative role in a given experimental scenario. The difficulty of exposing parts of the software to adequate testing can be the source of incorrect physics functionality, which in turn may generate hard to identify systematic effects in physics observables produced by experimental applications of the software.

Our ongoing, extensive effort on the validation of Geant4 [1, 2] physics models has been a playground for accumulating experience on how software design choices may affect the ability to test basic aspects of physics functionality and to monitor their evolution in the course of software life-cycle.

This paper introduces the subject and presents some examples drawn from this ongoing activity. It discusses how improved physics functionality could be achieved by improving the transparency of software design: for this purpose, one can apply established techniques, such as refactoring [3] and other methods suitable to deal with legacy software [4]. Guidelines to introduce testability into the software design since the early phases of the software development, and to preserve it in the course of the product life-cycle, are briefly summarized. Relevant issues in this process, which are specific to the domain of physics software, are highlighted.

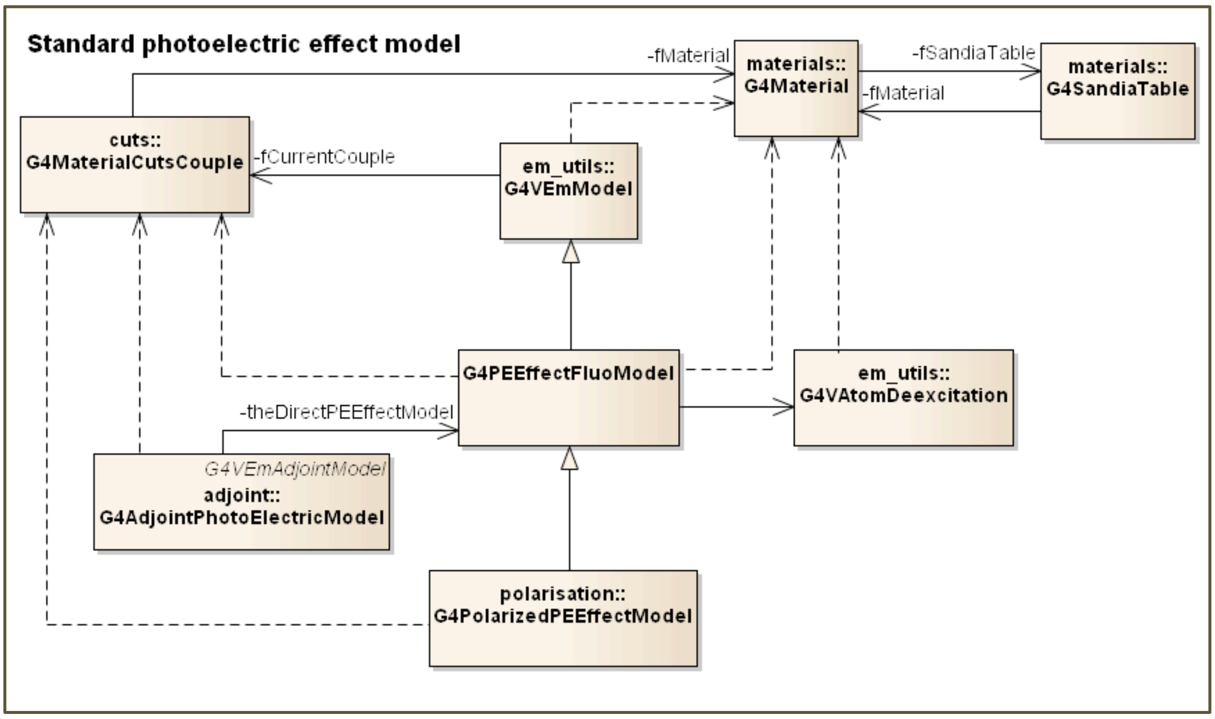

Figure 1. Overview of the software design of Geant4 standard photoelectric effect model. The UML class diagram highlights some dependencies on other Geant4 classes and packages.

The experience acquired in the context of Geant4 is relevant not only to its own future improvement, but also in the context of ongoing R&D on future simulation systems and, in general, of physics software development for high energy physics experiments.

This paper provides a brief overview of the main concepts and ongoing activities related to the design of testable physics, consistent with the constraints of the limited page allocation in these conference proceedings; more extensive details will be included in forthcoming journal articles.

2. An example: testing Geant4 photon interaction cross sections

A real-life example, derived from the experience with validating Geant4 physics, is a convenient approach to introduce the concept of testable physics in high energy physics software systems.

We stress that the study reported here is just part of a scientific research project, which documents the effort to compare basic physics modelling functionality embedded in public versions of an open source code (Geant4) with respect to experimental data available in the literature on objective ground, based on sound statistical methods: in doing this, there is no hidden intent to exalt, nor to denigrate any persons, institutes, computer codes, theoretical, phenomenological or empirical physics models.

A fundamental requirement for any Monte Carlo code is the validation of the basic foundation of the physics models that it applies in the course of particle transport through matter, e.g. the cross sections governing particle interactions with matter. According to the guidelines for software verification and validation summarized in the IEEE Standard 1012 [5], which in turn is related to other standard processes pertinent to software development [6], the validation of cross sections implemented in radiation transport codes implies their comparison with experimental measurements.

As an example, we consider here the validation of the total cross section for the photoelectric

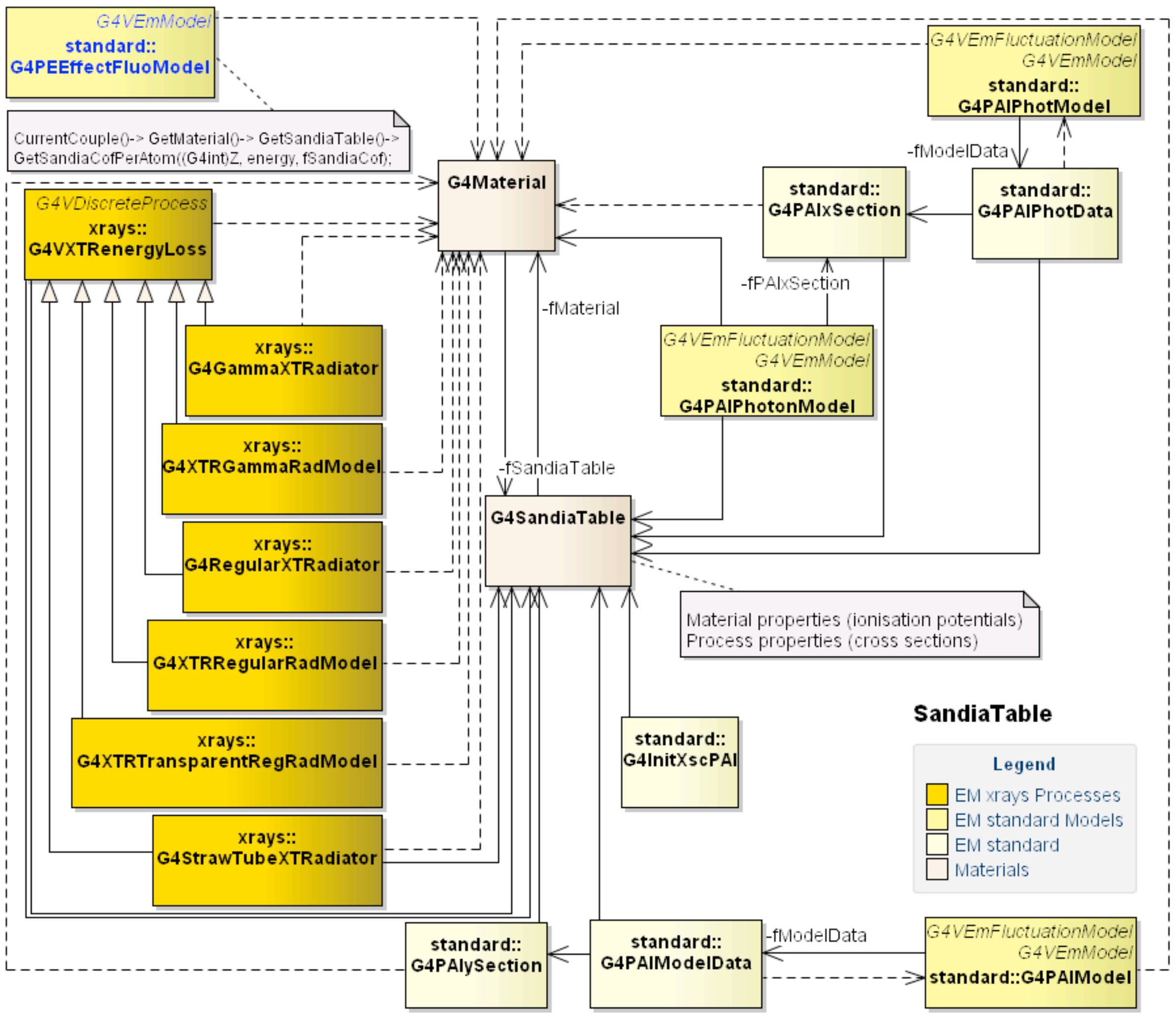

Figure 2. Overview of similar dependencies shared by other Geant4 electromagnetic models.

effect at the atomic level, i.e. concerning the interaction of a photon with an atom of the target material. The conceptual conclusions that can be drawn from the evaluation of this example are applicable also to other Geant4 models of the photoelectric effect, and to cross sections other than those pertaining to the photoelectric effect.

Geant4 encompasses a variety of models to deal with the simulation of the photoelectric effect in its electromagnetic processes package; in this example we focus on the *G4PEEffectFluoModel*, belonging to the *standard* electromagnetic package, which is commonly selected in simulations of high energy physics experiments.

The *G4PEEffectFluoModel* class encompasses a member function

```
G4double ComputeCrossSectionPerAtom(const G4ParticleDefinition*,
                                    G4double kinEnergy,
                                    G4double Z,
                                    G4double A,
                                    G4double, G4double);
```

which appears responsible for calculating the photoelectric cross section at the atomic level.

Since the atomic cross section of an electromagnetic process does not conceptually depend on any specific experimental configuration of a simulation application, one expects that such a basic physics modelling entity should be testable in a simple unit test, where one would instantiate a *G4PEEffectFluoModel* object and would invoke its ComputeCrossSectionPerAtom function with appropriate arguments: a pointer to a photon ParticleDefinition, the photon energy, and the atomic and mass numbers of the interacting atom.

The result of such a calculation should be compared with experimental data measured in the same conditions of photon energy and target element to estimate the validity of the software.

Nevertheless, the attempt to execute such a simple unit test fails, due to the extensive set of dependencies of *G4PEEffectFluoModel* on other parts of Geant4 code, which de facto require it to operate in the context of a full Geant4-based simulation: these extensive dependencies prevent elementary tests of physics modelling features, even if one only wants to calculate a fundamental modelling entity, such as the total cross section of a photon of a given energy for interaction with a given atom, which is conceptually independent from any experimental scenario.

A UML (Unified Modeling Language) class diagram shown in Figure 1 illustrates these dependencies. It is worthwhile to note that several other Geant4 models of photon interactions share a similar software design, which involves similar dependencies on the same classes, as can be seen in figure 2.

The software design of Geant4 electromagnetic abstract base classes, illustrated in figure 3, is the underlying reason for the difficulty of testing fundamental physics entities, such as atomic cross sections, in simple unit tests: these classes are characterized by a large number of data members and member functions, which are a source of risks of introducing dependencies that hinder their testability.

Regretfully, the reviewer's request of a substantial improvement in quality of figure 3 cannot be satisfied for practical reasons: the constraints of the page size of these conference proceedings prevent the inclusion of a sufficiently large figure to render the details of the depicted diagram in a readable font size. A larger scale, higher resolution version of figure 3 is available at http://www.ge.infn.it/geant4/papers/chep2015/chep2015-r1-em-inheritance.pdf for readers interested in visualizing the diagram with larger font size. Nevertheless it is worthwhile to stress that the role of this figure does not rest on the appraisal of the details of class attributes and operations; rather, this figure serves the purpose of conveying an overall message about a salient characteristic of the software design of Geant4 electromagnetic physics, i.e. the presence of base classes with a large number of data members and member functions. This key feature of the software design is effectively communicated by figure 3 in a visual manner. The reviewer's alternative recommendation of dropping this figure entirely would hinder the comprehension of the software design evolution in the scenario illustrated in this paper.

Class attributes in are listed in red in figure 3, class operations in green. Although the large number of attributes and operations in some base classes (including abstract ones) prevents appraising their details in this figure due to the small font size needed to fit within the page format of these conference proceedings, the reader can appraise the risk associated with a software design that involves classes with a large number of data members and member functions: this approach is prone to introducing dependencies, not necessarily related to physics, which hinder the testability of physics modelling elements embedded in those classes.

The apparent complexity of Geant4 electromagnetic design is consistent with the evolution of software systems that are actively used, as is summarized in Lehman's laws [7].

3. Methods for improving the testability of physics models

3.1. Refactoring and other advanced methods

Established methods exist to improve the design of existing software, which can contribute to improve also its testability.

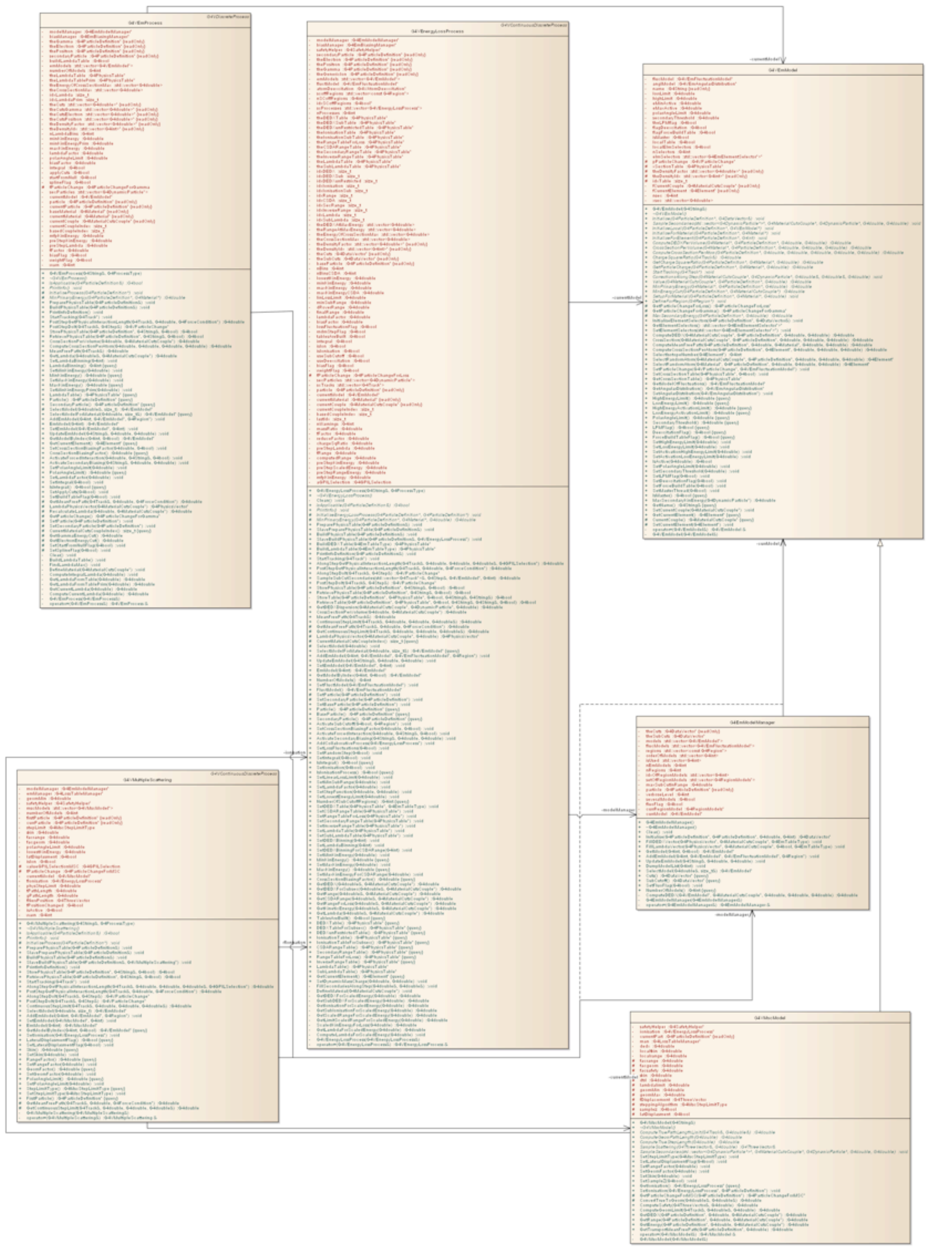

Figure 3. Overview of base classes of Geant4 electromagnetic physics: UML class diagram encompassing base classes for processes and models. Further explanation about this figure can be found in the body of the paper.

Refactoring [3] is a disciplined technique for improving the design of existing code; it is the most commonly adopted approach for the improvement of existing code, which is often the result of several development cycles, which may have fogged its pristine design (if it ever existed).

More advanced methods to deal with legacy software are documented in [4]: they focus on techniques that help breaking external and internal dependencies in the code, with the purpose of making it testable. These techniques are often the starting point for applying refactoring to the parts of the software that have become testable.

It is worthwhile to note that the concept of "testable" software has different meanings in the context of refactoring and of validation of physics software. In the context of refactoring unit tests are an essential tool to assess that the functionality of the refactored software is not altered by the refactoring process: i.e., the tests verify that the original and refactored software have identical behaviour. However, these tests do not address the issue whether the functionality of the software actually reflects the real physical world and satisfies the requirements of the intended use of the software. This task is the responsibility of validation tests.

3.2. An example: improving the testability of Geant4 physics

The example illustrated in section 2 showed that the appraisal of basic physics behaviour in Geant4 was hidden by software design that prevented its testability. In this respect it is worthwhile to note that a systematic validation of Geant4 photoelectric cross sections, spanning all its various modelling options, is not yet documented in the literature: therefore the test case considered in section 2 is not a pointless exercise, rather it responds to the scientific need of documenting the capability of Geant4 photoelectric effect simulation and quantifying the respective merits of the available modelling options.

In the logical framework of the previously discussed example, the validation of the photolectric cross sections implemented in *G4PEEffectFluoModel* requires establishing their consistency with measurements by means of appropriate statistical methods: making them testable in a simple unit test environment, independent from any application scenario, is the first step towards achieving this goal.

The approach towards making Geant4 photon interaction modelling testable (not limited to *G4PEEffectFluoModel*) consists of breaking the dependencies present in the current Geant4 classes pertaining to photon physics. This goal is achieved by performing in-depth analysis of the problem domain, which leads to decomposing the physics of the problem at a fine granular level into classes with sharply defined, minimal responsibilities.

Regarding the photoelectric effect, such a problem domain analysis leads to the identification of classes responsible for its intrinsic physics features: total cross section calculations, which determine the occurrence of a process, partial cross section calculations, which determine the creation of a vacancy in the atomic shell occupancy as a result of photoionization, differential cross sections, which determine the angular distribution of the photoelectron, etc. These classes, which are responsible for modelling fundamental atomic physics behaviour, are independent from any specific features of the experimental scenario; their design and implementation involve minimal or no dependency on other parts of Geant4. As a result, they are testable in a simple unit test environment. Indeed testability becomes a guideline in the object oriented analysis and design (OOAD) process: in this context, a software design candidate is acceptable, if the physics is testable. Once the testable candidate design is finalized, the functionality of existing Geant4 models is refactored. The minimalist character of the software design also facilitates the implementation of additional models not yet available in Geant4. An example of the result of the OOAD process concerning total photoelectric cross sections is illustrated in figure 4.

Thanks to this OOAD process, total photoelectric cross sections calculated by the various modelling options (each one encapsulated in a policy class [8]) can be compared with experimental references. Figures 5 and 6 illustrate how making physics testable by design allows the appraisal both of existing Geant4 physics modelling capabilities and of alternative approaches: the cross sections identified as “EPDL” and “BiggsG4” correspond to currently implemented models, while those identified as “Biggs” and “Chantler” identify alternative options, which have been considered in the course of an extensive validation project concerning the simulation of photoionisation. The full set of quantitative results deriving from the validation of Geant4 photoelectric cross sections will be documented in a forthcoming dedicated publication.

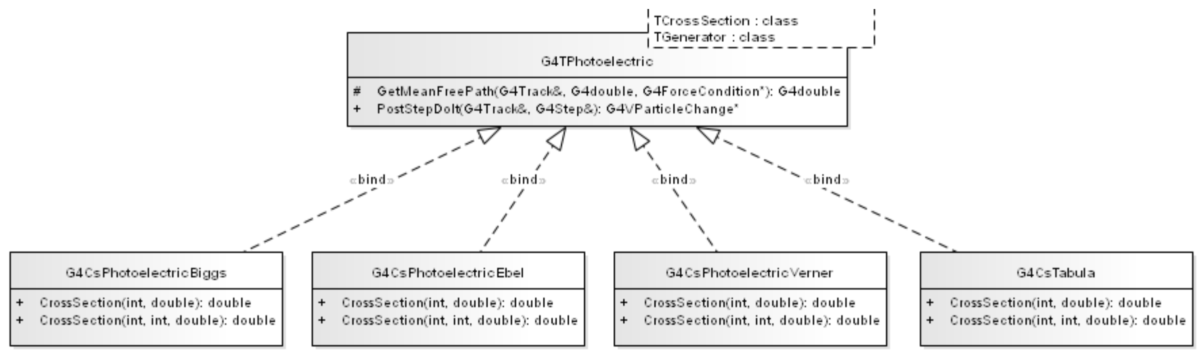
TCrossSection : class
TGenerator : class
G4TPhotoelectric
# GetMeanFreePath(G4Track&, G4double, G4ForceCondition*): G4double
+ PostStepDoIt(G4Track&, G4Step&): G4VParticleChange*
«bind»
«bind»
«bind»
«bind»
G4CsPhotoelectricBiggs
+ CrossSection(int, double): double
+ CrossSection(int, int, double): double
G4CsPhotoelectricEbel
+ CrossSection(int, double): double
+ CrossSection(int, int, double): double
G4CsPhotoelectricVerner
+ CrossSection(int, double): double
+ CrossSection(int, int, double): double
G4CsTabula
+ CrossSection(int, double): double
+ CrossSection(int, int, double): double

Figure 4. Overview of a testable design of photoelectric cross sections: cross section models are policy classes with minimal dependencies, which can be tested and validated (i.e. compared with experimental cross section data) in simple unit tests.

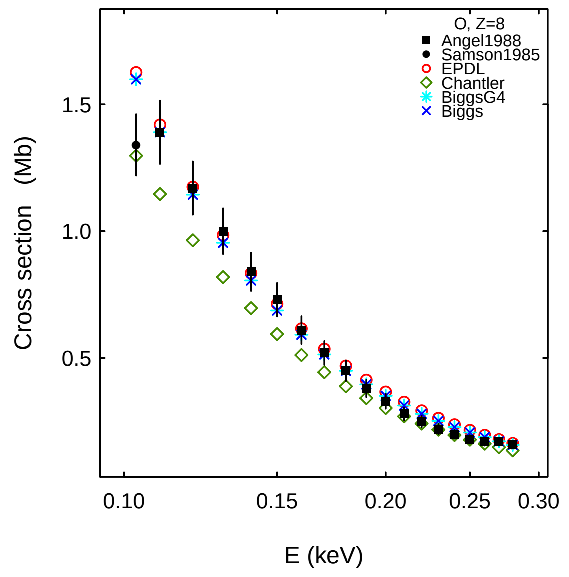

Figure 5. Comparison between total photoelectric cross section models (coloured empty symbols) and experimental data (black solid symbols) for oxygen.

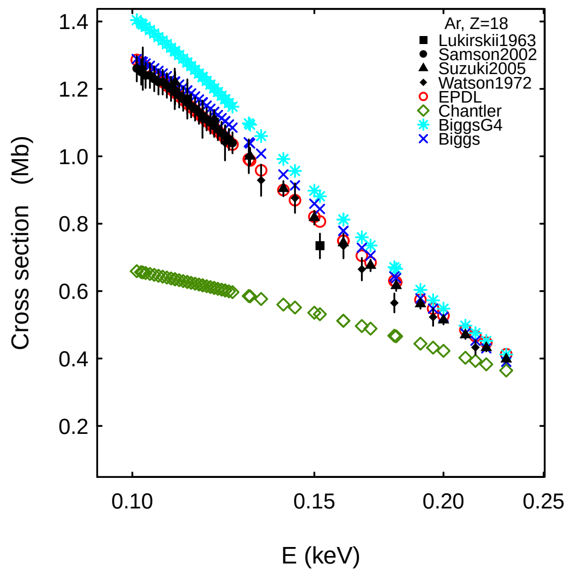

Figure 6. Comparison between total photoelectric cross section models (coloured empty symbols) and experimental data (black solid symbols) for argon.

4. Epistemic uncertainties embedded in the software

Improvements to the software design along the lines discussed in the previous sections enable introducing physics calculations in unit tests. Although this is an essential step towards their validation, the peculiarity of physics software requires in-depth knowledge of some implementation features as well.

A typical case is, for instance, the presence of “magic numbers” in the code, which may derive from fits to reference data, or parameters that may have been adjusted in a software calibration process to optimize the compatibility with some experimental benchmarks. Refactoring techniques, or more generally improvements to the software design, can make such software testable in terms of accessing its functionality in a testing framework; nevertheless, lack of knowledge of its implementation details would expose it to epistemological mistakes.

For instance, if the testers are not aware of the grounds of a physics models or of the details of its implementation, it could happen that data used to determine parameters embedded in the software implementation are reused in the software validation process: this would be an epistemological flaw, as validation requires comparison with independent data.

This example serves as a reminder that sound epistemology must always complement software technology in the process of testing physics software.

5. Conclusion

The ability to test physics software at a fine level of detail is an essential requirement to build confidence in the results of its calculations. This requirement should be taken into account at the time when the software is designed, and should be maintained in the course of the evolution of the software. A practical guideline in this context is that tests should be developed along with the design of the code: if a test is hard to write, that means that one has to devise a different design, which is testable.

Nevertheless, the experience with major software systems in use in high energy physics shows that the testability of physics software is often neglected at design time. Making it testable at a later stage implies breaking dependencies. Established techniques such as refactoring and methods "à la Feathers" [4] exist, which allow the improvement of legacy software in view of making it testable. Nevertheless, the process of making existing software testable usually requires extensive effort: caring about physics testability at an early stage is definitively preferable.

The peculiarity of physics software requires that the technology for making software testable is supported by sound epistemology.

An exploratory research project is in progress to enable the testability of Geant4 basic physics modelling features by improving the software design. In-depth problem domain analysis and sharp domain decomposition support the design of minimalistic physics entities, mostly in terms of policy classes, which can be easily evaluated through unit tests. This software design enables fast, quantitative validation of existing Geant4 physics models as well as of new models that are not yet available in Geant4. Concepts, methods and techniques developed for this project are applicable also to other contexts of physics software systems.

Acknowledgments

The authors acknowledge the support from their institutes and funding agencies in the course of the research documented in this paper. The authors thank Steffen Hauf and Markus Kuster (European XFEL GmbH) for valuable discussions.

The UML diagrams enclosed in this paper have been produced by means of an academic license of Enterprise Architect graciously made available by Sparx Systems.

Some reviewer's corrections have not been introduced in the revised version to preserve correctness (e.g. "de facto" is a correct Latin expression, unlike the recommended "de-facto").

References

[1] Agostinelli S *et al* 2003 *Nucl. Instrum. Meth. A* **506** 250
[2] Allison J *et al* 2006 *IEEE Trans. Nucl. Sci.* **53** 270
[3] Fowler M 1999 *Refactoring* (Boston: Addison-Wesley Professional)
[4] Feathers M C 2004 *Working Effectively with Legacy Code* (Upper Saddle River: Prentice Hall)
[5] IEEE Computer Society 2012 *IEEE Standard for System and Software Verification and Validation* (New York: IEEE)
[6] ISO/IEC/IEEE 2008 *Standard for Systems and Software Engineering - Software Life Cycle Processes* (Geneva: ISO)
[7] Lehman M M 1980 *Proc. IEEE* **68** 1060
[8] Alexandrescu A 2001 *Modern C++ Design: Generic Programming and Design Patterns Applied* (Boston: Addison-Wesley Professional)