CW-123300-CONF-005
UNRESTRICTED

The Development of Model Generators for Specific Reactors

J.C. Chow
Chalk River Nuclear Laboratories, Atomic Energy of Canada, Limited

Abstract

Authoring reactor models is a routine task for practitioners in nuclear engineering for reactor design, safety analysis, and code validation. The conventional approach is to use a text-editor to either manually manipulate an existing model or to assemble a new model by copying and pasting or direct typing. This approach is error-prone and substantial effort is required for verification. Alternatively, models can be generated programmatically for a specific system via a centralized data source and with rigid algorithms to generate models consistently and efficiently. This approach is demonstrated here for model generators for MCNP and KENO for the ZED-2 reactor.

1. Introduction

The advent of computing technology in the last few decades has allowed for the simulation of reactor systems with unprecedented fidelity and outcome that increase the predictive capability of simulations of reactor properties considerably. Reactor models for many computer simulation codes, such as MCNP [1] and SCALE [2], are often realized as plain text files containing hundreds or thousands of lines of data that represent the key characteristics such as material compositions and geometry of the reactor. Authoring reactor models is a routine task for practitioners in nuclear engineering for reactor design, safety analysis, and code validation. Conventional methods include writing the model from scratch using a generic text editor via direct typing, or copying/pasting from other text files and spreadsheets; modifying an existing model for a different configuration; or using extensive Excel® macros to generate the models. It has been the author's observation that the first two methods are rather tedious, models generated by these methods are often error-prone, and extensive effort on verification is required to warrant the fidelity and validity of the models. The method using Excel macros is probably more preferable among the three, but this method is often tailored for personalized use and with cryptic documentations, if any, which renders it difficult to share among workers. Other methods have also been implemented with scripting languages but the solutions are often in a piecemeal fashion and difficult to use. In an attempt to address these issues, the author has developed a suite of Windows® applications capable of generating full-core MCNP5 and KENO-V.a models of the ZED-2 reactor (Zero Energy Deuterium) [3][1], and the CANDU-6 reactor.

The two model generators for MCNP and KENO for the ZED-2 reactor are used in this article to demonstrate the approach the author advocates as an alternative to the conventional approach of authoring large reactor models with text editors and/or spreadsheets. Inconsistencies among models are avoided by generating the models through a common data source. Graphical user interfaces, with data validation capability, are provided to facilitate entry of all data required for generating a model. Efficiency is enhanced by automatically generating data cards, consistency with the syntactical

[1] A research reactor located at the Chalk River Laboratories, ON, dedicated predominately to the study of CANDU®-type nuclear fuels.

requirement of the codes, based on data entered by the user and other specific data files provided along with the applications.

Other applications are available that facilitate authoring of models of generic nuclear systems for MCNP and KENO, such as `VISED` [4] for MCNP input data entry and geometry visualization, and `Geewiz`, which is included in the SCALE [2] package serving similar purposes as `VISED`. However, it should be noted that applications such as `VISED` and `Geewiz` were designed to facilitate modeling of generic systems of any complexity allowable by the simulation tools. On the contrary, the suite of applications developed by the author has been designed to target specific systems such as ZED-2 and CANDU-6. In this respect, the usefulness of the applications targeting specific systems might seem to be limited compared to the generic applications. However, the process of defining a problem domain that addresses generic problems automatically excludes any specific problems relevant to specific systems. Incidentally, it is exactly the lack of functionalities in the generic tools that address the specific problems that has motivated the author to develop the specific applications. Furthermore, since the problem domains that the specific applications address are well-defined, the applications are highly customized and developed in-house, the codes so developed are much more manageable than any tools acquired through external vendors. In fact, since the set of codes for the specific applications have been developed with a fully object-oriented programming (OOP) approach with emphases on maintainability and extensibility, they can be updated frequently according to changing requirements, and highly extensible to solve other problems of a similar nature within the organization. In this respect, the short term investment for specific projects to develop such applications might seem high, but the benefits from the investment will prove to be worthy to the organization in the long term.

2. Software Development

The development of the two model generators for MCNP and KENO for the ZED-2 reactor follows the "Iterative and Incremental"[2] approach which is more suitable for this kind of applications than the "Waterfall"[3] approach, since the requirements are expected to evolve in time. The model generators are implemented with the modern programming language C# 3.0 [5] along with the `.NET Framework Library` version 3.5 (SP1) to take advantage of the rich graphical user interface (GUI) features provided by the library. The integrated development environment (IDE) `Microsoft Visual Studio 2008` was employed for coding the applications. The software development process is elaborated in the following subsections.

2.1 Modelling Requirements

Initial requirements for the model generators were compiled by the author based on his experience in authoring the models manually, with additional requirements collected through literature research and discussions with colleagues. The requirements are dictated by the physical configuration of ZED-2, a research reactor dedicated to the study of CANDU-type fuel bundles and related reactor physics phenomena. A schematic of ZED-2 is shown in Figure 1. The periphery of the reactor consists of a graphite wall enclosing an aluminum calandria tank of ~3.4 meters both in diameter and height.

[2] In the "Iterative and Incremental" approach, systems are developed through repeated cycles (iterative) and in small manageable portions (incremental), allowing software developers to take advantage of experience learned during development of earlier components of the system, and to expand on functionalities as requirements evolve.
[3] In the "Waterfall" model, the development process is divided into phases that are executed sequentially. Requirements are captured up-front and revisiting or revising of prior phase is strongly discouraged.

Neutron shielding materials at the top of the reactor also constitute part of the periphery. Fuel lattices are formed by suspending fuel channels, which contain up to five CANDU-type fuel bundles, from steel beams located at the top of the reactor. ZED-2 is categorized as a thermal reactor which utilizes heavy water as a moderator to slow down the neutrons to optimize fission of the fuel. Heavy water is pumped into the calandria tank through dump lines at the bottom. Criticality is achieved by controlling the volume of heavy water in the calandria tank. Elaborated details of the ZED-2 reactor can be found in [6].

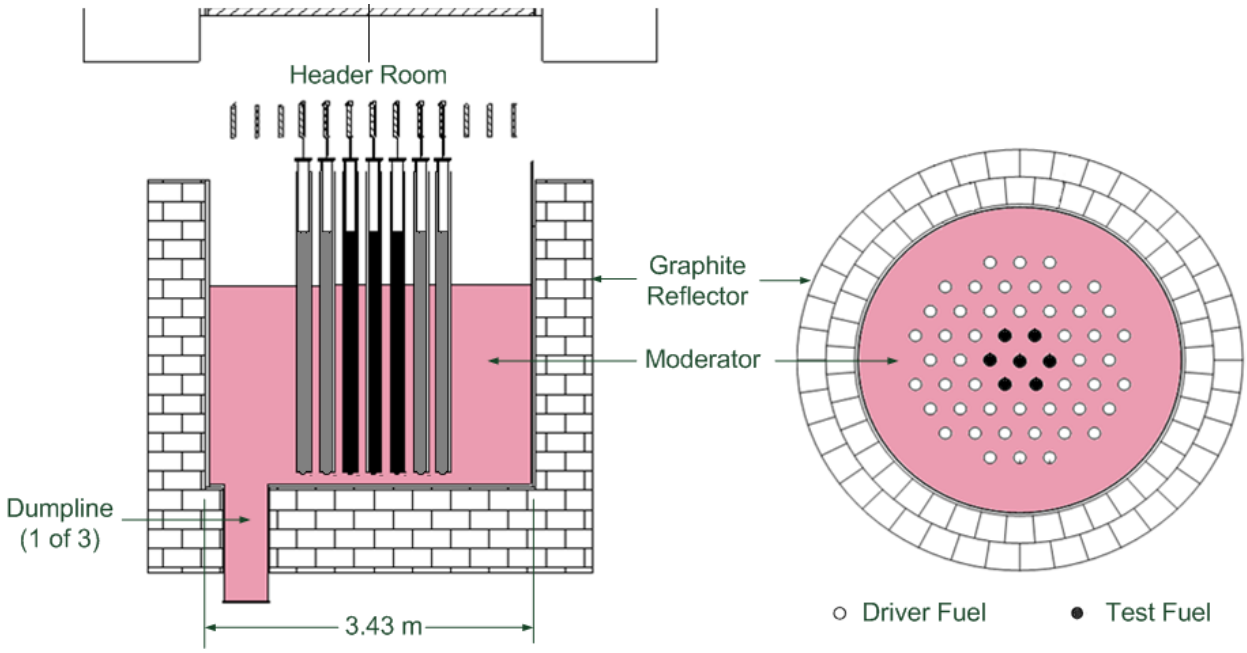

Figure 1 Schematic of the ZED-2 Reactor.

The model of a physical system is defined with two aspects: geometry and material, which are often handled separately and subsequently cross-referenced in the input data of a model. In essence, a physical system is simply a distribution of materials in space. Geometrically, each experiment in ZED-2 is defined by a specific fuel lattice, such as the one shown in Figure 1, which contains different types of fuel channels, which in turn, might contain different types of fuel bundles. The cross-section of a typical CANDU-type fuel bundle contained in a fuel channel, consisting of a pressure tube and a calandria tube, is shown in Figure 2. The fuel lattice is either square or triangular, and varies in pitch between ~20 to ~40 cm.

According to the above description, a general requirement is that the generator must be able to handle the geometry and material compositions of different types of fuel bundles, fuel channels, and lattice shapes. Specifically, ZED-2 is dedicated to the study of CANDU-type fuel bundles, which usually contain a center fuel pin and up to three concentric rings of pins as depicted in Figure 2. In fact, only this type of bundle geometry, which covers all the fuel bundles used in ZED-2 to date, was targeted in the initial development of the model generators.

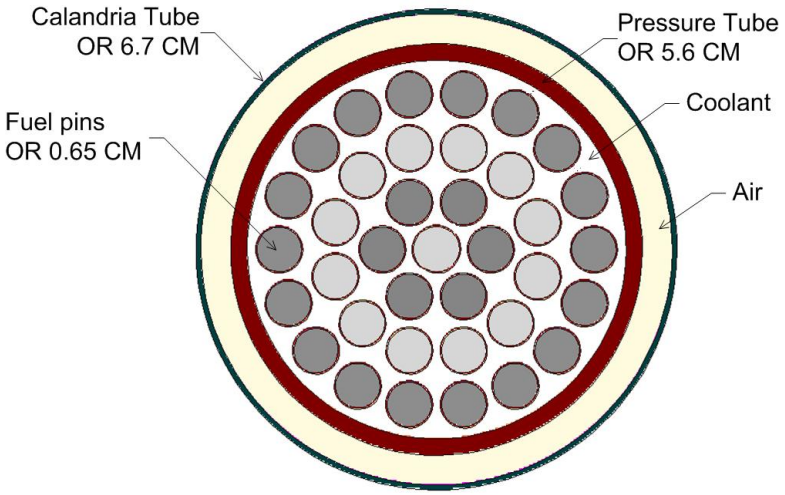

Figure 2 Cross-section of a CANDU-type Fuel Bundle inside a Fuel Channel.

2.2 Application Design and Implementation

From the object-oriented programming (OOP) perspective, it is common practice to describe the structure of an application with the aid of a class diagram using the Universal Modelling Language (UML) [8], which consists of symbols and connectors (arrows) that represent the structure of a software system and the relations among its components. A class in OOP is a construct with two distinct aspects: properties (data) and methods (functions); and methods are implemented around the properties to solve the problem. This is different from the procedural programming approach in which data and methods are often treated separately. In the present context, the basic data that represent the variable parameters of the physical system are to be supplied by the user, and the methods are implemented, often taking the data structure into consideration, according to the objective of the application. By design, the process for the model generator(s) to generate a full-core model can be described as follows:

i) Data that represent the static component of the system, i.e., the periphery, are pre-compiled in a static data file, which might also reference other static data files.

ii) The physical system is broken down into components. For each component, the user utilizes a graphical user interface specifically designed to capture the variable parameters of the component and saves the data to persistent storage with a predefined format. Other parameters such as the properties of coolant(s) and moderator are captured similarly.

iii) The entire set of data file names pertaining to a specific reactor configuration are compiled into a single full-core configuration file and the user specifies this file when invoking the method to generate the full-core model.

The philosophy behind the above design is that data that represent the entire physical system are decomposed into small and manageable parts, the data pertaining to each part are provided by the user with the aid of GUIs, and the parts can be reused and shared among similar configurations.

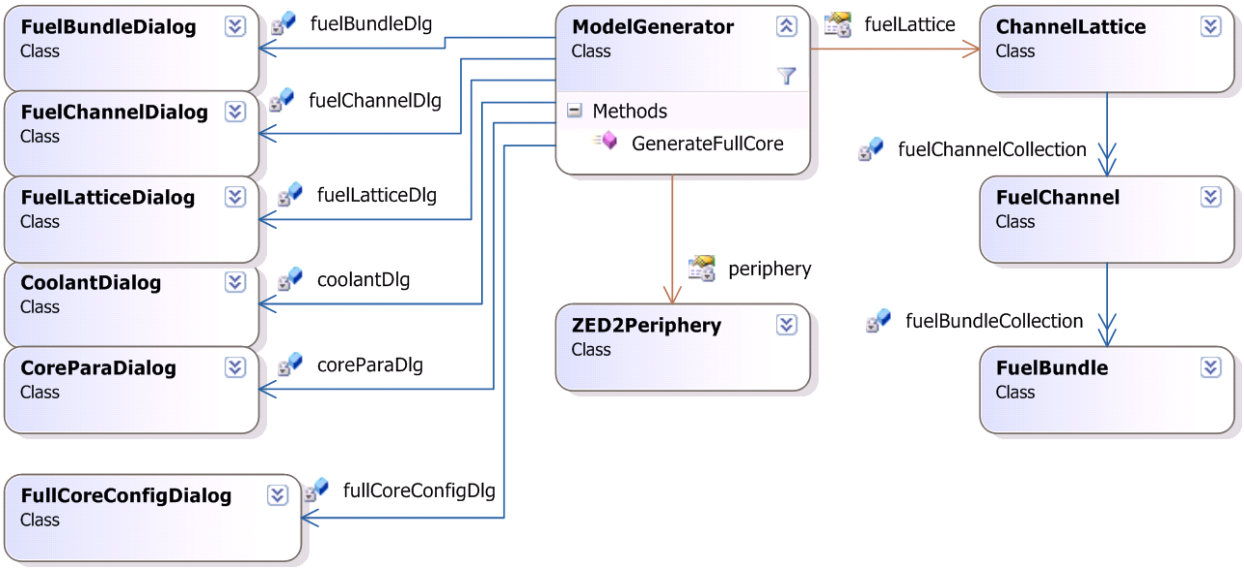

Figure 3 UML Class Diagram of a Model Generator.

A class diagram of the model generator(s) is shown in Figure 3, which will be referenced frequently in the following discussions. In Figure 3, the `ModelGenerator` class implements the main window of the application from which the user can invoke other functions through visual objects, such as buttons and pull-down menu items familiar to Windows users, or shortcut keys. The other classes listed in Figure 3 can be divided into two categories:

a) Classes that represent physical components of the system:
i) `FuelBundle`, ii) `FuelChannel`, iii) `FuelLattice`, and iv) `ZED2Periphery`;

b) Classes that implement graphical user interfaces for data entry by the user:
i) `CoreParaDialog`, ii) `FuelBundleDialog`, iii) `FuelchannelDialog`,
iv) `CoolantDialog`, v) `FuelLatticeDialog`, and vi) `FullCoreConfigDialog`.

Each of the four classes in the first category represents a physical component of the system, which is a natural design that connects the physical system with the software. Those in the second category are graphical user interface (GUIs) classes that implement visual objects, such as textboxes and data tables, which allow the user to enter data through the keyboard or mouse. The `Prerequisites` pull-down menu in the main window is shown in Figure 4; each of the menu items in the pull-down menu is mapped to one of the six GUI classes. The functions of the two categories of classes will be elaborated in the following subsections.

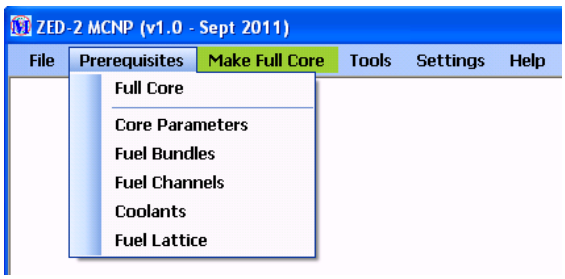

Figure 4 The Main Window of the Model Generator showing a Pull-down Menu

2.2.1 The Fuel Lattice and Periphery

The hierarchical structure of a fuel lattice in the ZED-2 reactor coincides ideally with a pattern in OOP in which an object contains a collection of another type of object; in parallel with a fuel lattice containing a collection of fuel channels, which in turn, contain a collection of fuel bundles. Accordingly, three generic classes[4], *viz*., `FuelBundle`, `FuelChannel`, and `FuelLattice`, have been identified for development, as depicted in the right hand column of Figure 3 whereby a double-headed-arrow line represents an object containing a collection of the objects it points to. Thus, the problem of modelling a fuel lattice is decomposed into three hierarchical components, each of which is associated with a physical component of the system. The advantage of taking this approach is that a large problem is broken down into smaller ones and each is tackled separately and relatively independently; which makes code development more focused, less error-prone, and easier to debug and maintain.

As mentioned previously, a class contains two aspects: properties and methods. In fact, each of the three classes in the right hand column of Figure 3 defines a set of geometric and material properties that specify the dimensions of the object and the materials that comprise it, and methods are implemented according to these properties to generate text outputs in accordance with the syntactical structure of the specific type of model. By design, the specific properties of the object are read from a data file in XML (Extended Markup Language) format [7] when each object is instantiated. The methods to create and maintain these data files are discussed in the Section 2.2.2.

The periphery of the ZED-2 reactor, which consists of the calandria tank, the graphite wall, and neutron shielding at the top of the reactor, is identical for all experiments, and therefore, it is modelled independently of the fuel lattice. The class `ZED2Periphery` shown in the middle of Figure 3 has been designed for this purpose. It has been implemented as a singleton class, i.e., only a single instance of the class exists within the application. Otherwise, its implementation is very similar to that of the other three classes that comprise the fuel lattice, except that it always reads the same XML file (provided along with the applications) at instantiation.

2.2.2 Graphical User Interfaces and XML Files

The user must provide data to the model generator to specify the configuration of a fuel lattice. Graphical user interfaces (GUIs), depicted as `dialogs` in the left hand column of Figure 3, have been designed to facilitate data entry. In fact, each of the classes `FuelBundle`, `FuelChannel`, and `FuelLattice`, is associated with a GUI class and a data structure appropriate for the physical object. For example, the GUI for inputting the geometric data of a fuel bundle, implemented in the `FuelBundleDialog` class, is shown in Figure 5. It has been demonstrated that the data fields shown in Figure 5 are adequate for capturing geometric parameters of all the common fuel bundles used in experiments in ZED-2. This dialog also implements a button [`Materials`], which invokes another dialog for inputting the material data. Once data entry is complete, the [`Save`] button can be used to save the data, which are captured from the GUI and formatted into XML[5], to persistent storage with a

[4] Strictly speaking within the OOP paradigm, a class is a construct that is used to create instances of itself, referred to as objects, i.e., an object is an instance of a class.

[5] The XML (Extended Markup Language) format, which is by itself text-based, is preferred against a plain text file. XML has been advertised as both human- and machine-readable. A well-formed XML file is self-describing; and since it is a standard, it also facilitates data communication across computer platforms.

unique file name provided by the user. In principle, an XML data file can be authored manually, or an existing XML data file can be modified, using a text editor without resort to using the GUIs. However, using the GUIs has the added advantage of data validation. For example, methods have been implemented in the GUI classes such that if a numeric value is mistyped with letters or if a numeric value is outside of a predefined limit, a pop-up message will prompt the user to correct the error(s) before the data can be saved.

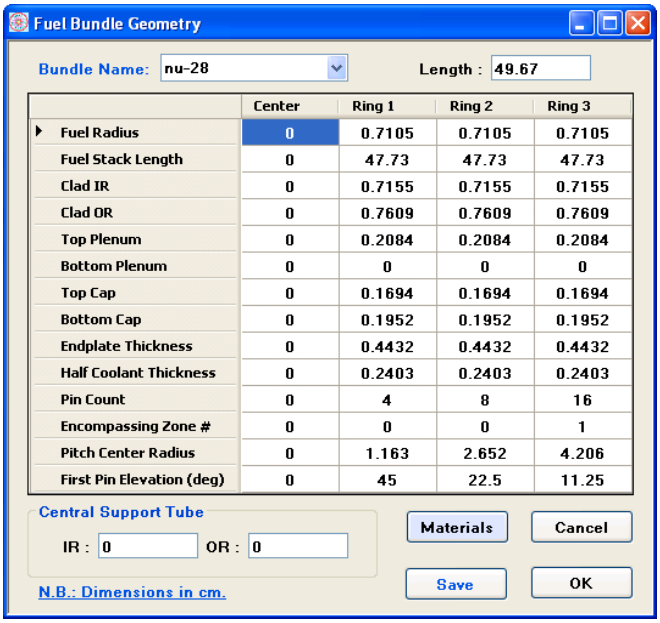

Figure 5 GUI for Input of Fuel Bundle Parameters.

GUIs relevant to the other two physical component classes, `FuelChannelDialog` and `FuelLatticeDialog`, have been implemented in a similar manner, except for the different GUIs designed for capturing the relevant data.

As mentioned previously, the reactor periphery is handled separately from the other three physical component classes. Since the periphery is static, user input via GUI is not necessary. An XML file provided along with the applications and with a schema understood and read by the `ZED2Periphery` class at instantiation, is used to specify the geometric and material properties of the periphery components.

Two other GUI classes listed in Figure 3 are relevant to capturing of crucial data to completely specify the configuration of a ZED-2 experiment. Inside a fuel channel in ZED-2, the fuel bundles are passively cooled by a coolant inside the channel through convection. The properties of the coolant (type, density, and temperature, etc), usually heavy water or air, are captured by invoking an instance of the `CoolantDialog` class, and the data are saved as an XML file, similar to the other GUI classes, for subsequent reference by a fuel channel object. As mentioned previously, ZED-2 is a thermal reactor that utilizes heavy water as moderator to slow down the neutrons. The properties of the moderator (purity, temperature, and density) are captured by invoking an instance of the `CoreParaDialog` class. Other parameters such as the height of the moderator, the concentration of poisons[6] in the moderator, and the size of the lattice pitch, are also captured through this GUI, and the data are again saved in an XML file, for subsequent use to generate a full-core model.

[6] In reactor physics, a poison is a substance with a large neutron absorption cross-section.

2.2.3 Generate Full-core Model

The hierarchical structure of the application, mirrored by the physical structure of a fuel lattice, dictates that components at the bottom of the hierarchy must first be specified before those higher in the hierarchy can be assembled. Logically, one or more fuel bundles must have been defined before a fuel channel can be assembled. Similarly, at least one fuel channel must have been defined before a fuel lattice can be assembled. Once a set of fuel bundle or channel data files have been generated, they can be verified by another independent analyst. The verified fuel bundle and channel data files can then be used in production mode to assemble fuel lattices appropriate to the specific configurations of the experiments.

Besides the three types of data associated with the physical component classes, two other sets of data must be provided: the coolant properties and core/moderator parameters. Understandingly, XML data files associated with each of the abovementioned five types of data must have been created, either by the user using the GUIs as discussed in the last section or from other sources, before a full-core model can be generated.

Once all the data files relevant to a specific experimental configuration have been generated, a full-core configuration can be assembled via the GUI implemented by the `FullCoreConfigDialog` class. An example of a full-core configuration is shown in Figure 6. To assemble a full-core configuration, the user chooses the relevant configuration files through the pull-down menus that show the available configurations. Once all the required configuration files have been specified, the user can point to the [`Generate Model`] button shown in Figure 6 to generate the full-core model, manifest as text which will be displayed in the main window of the application.

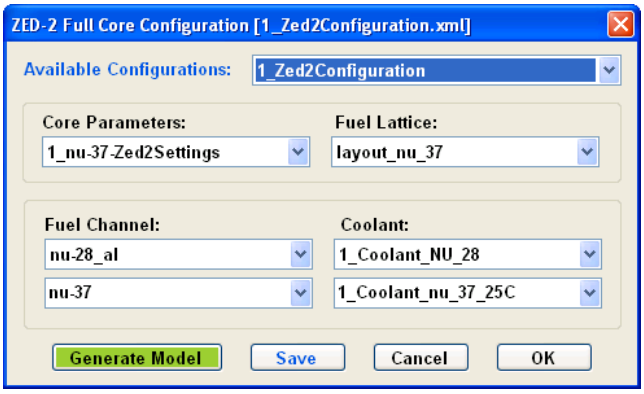

Figure 6 GUI for Assembling Data to Generate a Full-Core Model

2.2.4 Brief Discussion on Polymorphism

The discussions so far have not specified the type of model the applications generate. In fact, the non-necessity of specifying a type is considered a useful feature known as polymorphism in OOP, which allows values of different data types to be handled using a uniform interface. The present applications target two types of models: MCNP and KENO-V.a, which require different syntactic structures. All the features of the applications not specific to any type of model (the raw geometric and material data and the GUIs that facilitate data entry) are handled by the classes described in Figure 3. In order to generate text data appropriate to the specific type of model, specialized classes, known as *derived classes* or *subclasses* in OOP, are constructed from their respective base classes. Figure 7 shows the UML class

diagram of the derived classes for the `FuelBundle` class. In essence, the derived classes inherit all the `public` and `protected`[7] properties and methods of the base class. In a derived class, an inherited method can be overridden (more commonly known as *function overloading* in OOP), i.e., a method with the same signature (name, and arguments, if any) but different codes compared to that in the base class, or new methods can be added to expand the functionality of the derived class. In the present context, all the properties and data access methods of a `FuelBundle` are inherited and shared between the derived classes `MCNPFuelBundle` and `KENOFuelBundle`, and the only difference between the two is in the method implemented to generate the text in accordance with the syntactic structures of the specific model, MCNP or KENO. Specialized classes for the `FuelChannel`, `FuelLattice`, and `Zed2Periphery` classes are derived and implemented similarly. In fact, two model generators, `Zed2MCNP` and `ZED2KENO`, have been developed in parallel to target the two types of model. These two applications share all the common classes listed in Figure 3, and derived classes such as `MCNPFuelBundle` and `KENOFuelBundle` are implemented to suit the different syntactical structures of the models.

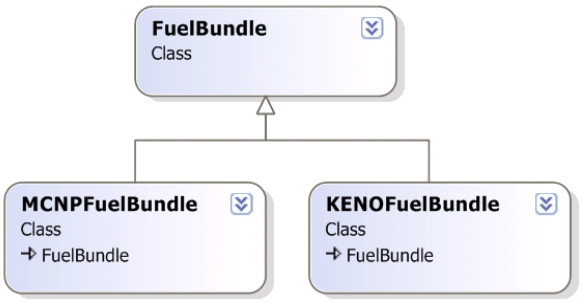

Figure 7 UML Class Diagram − Derived Classes of the `FuelBundle` Class

2.2.5 Advantages of the Approach

From the perspective of authoring a complex reactor model, the advantages of using the above approach over using plain text editors and/or spreadsheets should now be obvious. The task of manipulating a large amount of data, dominated by complex and strict syntactic structures, has been converted to acting on small sets of data through the use of GUIs with data validation capability, which makes data entry straightforward and less error-prone. The user is only required to provide the parametric values (dimensions and material compositions) that specify the physical system, which is decomposed into manageable components that can be handled, and the syntactic structures of the models are handled completely by the code. Inconsistencies among models are avoided since they are generated from the same data source, i.e., the set of predefined XML data files created with the aid of the GUIs or otherwise. By taking advantage of polymorphism, two very similar problems are solved simultaneously with exactly the same pattern and interfaces, except for the specialized part that generates the text for the specific model, which must be handled separately.

Another benefit of the approach is that it promotes a standard algorithm to model ZED-2 experiments. As a result of issues such as discrepancies between design and as-built values and availability of as-built data, and more so on the tolerance (uncertainties) inherent in the specifications of any physical system, any model of a relatively complex system such as ZED-2 is at best a good approximation of

[7] In the C# implementation of OOP, "`public`" and "`protected`" are keywords that allow derived classes to access members (properties and methods) of the base class while "`private`" members are not accessible to derived classes.

reality. Furthermore, models of the same system authored by different analysts might be rather different depending on the personal experience, preference, and engineering judgement of the individual analyst while interpreting the raw physical data. Since the applications enforce the use of a single data source (the set of XML data files that specifies the system), the models generated are guaranteed to be consistent. It also promotes collaboration since the standard algorithm should be defined collectively with inputs from all relevant workers.

Another capability of the generator, only relevant to MCNP models, is the automation of the process of generating the so-called "pseudo-materials" [9] for a particular temperature at which data are not available in the nuclear data library. The pseudo-materials are obtained by interpolating between two temperature nodes available in the data library. This is a very tedious task if done manually but the process has been automated in `Zed2MCNP` and it is completely transparent to the user. Note that since this capability is not available in `Zed2KENO`, one should take the temperature effect into consideration when results from the two types of models are compared, although the impact is expected to be small.

3. Examples on Using the Model Generators

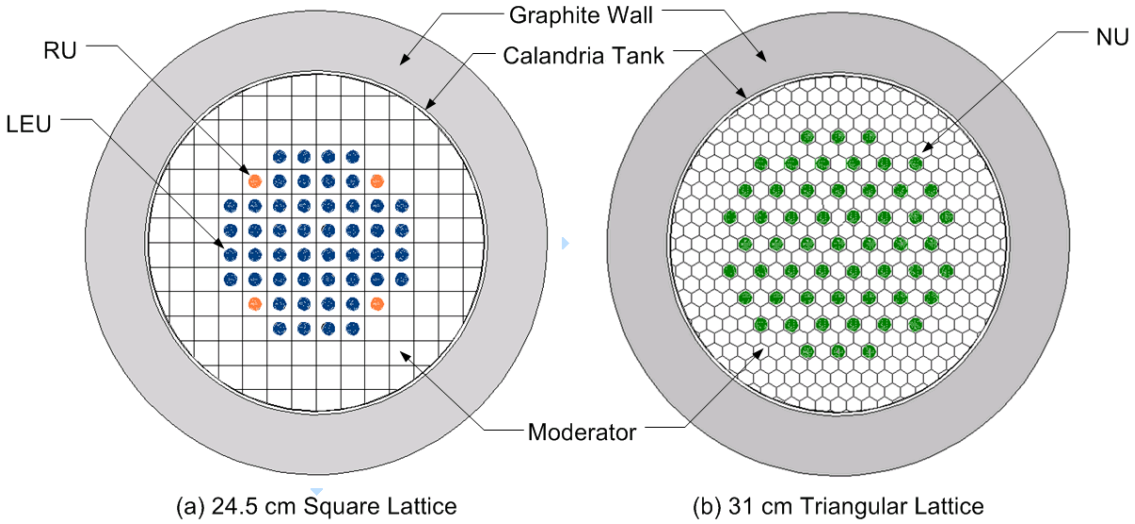

Figure 8 Experiments with LEU/RU Fuel and NU Fuel in ZED-2

Two examples are provided here to demonstrate the practical capability of the model generators. The cross-sectional views of the configurations of two ZED-2 experiments are shown in Figure 8. The core lattice in Figure 8(a) consists of Light Enriched Uranium (LEU) and Recovered Uranium (RU) fuel bundles arranged in a square lattice at a pitch of 24.5 cm. The objective of the experiment was actually a study of the effect of gadolinium in the moderator [10]. In the present demonstration, comparisons are made on the results of k_{eff} values of the reference case (without gadolinium) computed for the models generated by `Zed2MCNP` and `Zed2KENO`, targeting nuclear data libraries based on ENDF/B-VII.0 [11]. The procedure for obtaining the models with the generators is listed below:

i) Compile the physical data for the LEU/RU fuel bundles and the fuel channels: dimensions and material types from design and/or as-built drawings; material compositions from standard material data sheets, fuel fabrication reports, and mass spectroscopy reports.
ii) Compile experimental data of the reference case (moderator height, moderator and coolant temperatures and purity, etc) from the experimental report.

iii) Use the GUIs described in Section 2.2.2 to input the above data, save the corresponding XML data files, and compile them into a full-core configuration file.

iv) Run `Zed2MCNP` and `Zed2KENO` to generate the models based on the above full-core configuration file.

The first three steps in the above procedure are not specific to the model type. Once all the physical data have been collected (in the form of XML data files), the full-core MCNP and KENO models can be obtained by running the respective generators. The k_{eff} values obtained with running the MCNP and KENO simulation programs are listed in Table 1. Since the two models were based on exactly the same set of physical data, the resulting k_{eff} values are expected to be very similar. Indeed, as shown in Table 1, the difference is only 0.76 mk, which can be attributable to minor differences between computational algorithms, temperature treatments, and the nuclear data libraries used by the two codes.

Table 1 Results of k_{eff} values for the Square Lattice Experiment

MCNP k_{eff}	KENO k_{eff}	Diff. [mk]
0.99782(3)*	0.99706(3)*	0.76(4)

* Statistical Uncertainty (±0.00003).

The core lattice in Figure 8(b) consists entirely of 28-element Natural Uranium (NU) fuel bundles arranged in a triangular lattice at a pitch of 31 cm. This experiment has been evaluated and determined to be acceptable as benchmark data describing a critical configuration [6]. The nuclear data library based on ENDF/B-VI.8 [12] was used in the evaluation. Following a procedure similar to the square-lattice experiment with the model generators, four MCNP models were generated for the benchmark experiment targeting both the ENDF/B-VI.8 library and the more recent ENDF/B-VII.0 library, and with heavy water and air as coolant. Two KENO models were also generated targeting the ENDF/B-VII.0 library. The k_{eff} values obtained with running the MCNP and KENO simulation programs for the six cases are listed in Table 2, except for the first column, which has been quoted from Reference [6]. A comparison of the k_{eff} values of the MCNP models targeting the ENDF/B-VI.8 library (Columns 2 and 3 of Table 2) shows a difference of 1.55 mk between the D_2O-cooled cases and 1.04 mk for the air-cooled cases. Noting that temperature interpolation was not used in the benchmark evaluation but is implemented in `Zed2MCNP`, and minor differences also exist between some of the material compositions, the differences in the k_{eff} values are expected. Incidentally, the k_{eff} values obtained with `Zed2MCNP` are more consistent (wthin 0.3 mk) than those in the benchmark evaluation [6] when compared with those listed in Ref. [13], which listed comparisons of k_{eff} results between the ENDF/B-VI.8 and ENDF/B-VII libraries. Comparisons of k_{eff} values between KENO and MCNP models obtained with the generators targeting the ENDF/B-VII.0 library (Columns 5 and 6 of Table 2) show that the k_{eff} value of all the cases are within ~0.2 mk of unity, consistent with the results in Ref. [13].

Table 2 Results of k_{eff} values for the Triangular Lattice (Benchmark) Experiment

Coolant	ENDF/B-VI.8 MCNP k_{eff} [6]*	†ENDF/B-VI.8 MCNP k_{eff}	Diff. [mk]	†ENDF/B-VII.0 MCNP k_{eff}	ENDF/B-VII.0 KENO k_{eff}	Diff. [mk]
D_2O	0.99318(7)	0.99473(3)	1.55(8)	0.99989(3)	1.00006(3)	0.17(4)
Air	0.99361(8)	0.99465(3)	1.04(9)	0.99982(3)	1.00004(3)	0.22(4)

N.B.: Values in parentheses are statistical uncertainties in the last digit. *Results quoted from the benchmark evaluation [6].

†These values have been obtained with 400M active neutron histories.

4. Conclusion

This article introduces an approach in generating computer models for simulations of reactor systems in which models are generated programmatically as compared to the conventional approach of using text-editors and/or spreadsheets. The model generators have been implemented with a fully object-oriented programming approach that promotes maintainability and extensibility. Graphical user interfaces are provided with data validation capabilities which make data entry straightforward and less error-prone. Since this approach enforces the use of a single data source, models generated are guaranteed to be consistent. It also promotes collaboration since the standard algorithm to generate the models should be defined collectively with inputs from all relevant workers. Two examples are provided to demonstrate the capability of the model generators. The author is of the view that this kind of approach can be applied to any simulations of any physical systems with any complexity using any codes.

5. References

[1] X-5 Monte Carlo Team; "MCNP – A General Monte Carlo N-Particle Transport Code, Version 5", LA-UR-03-1987, Los Alamos National Laboratory (2003).

[2] Oak Ridge National Laboratory, "SCALE: A Modular Code System for Performing Standardized Computer Analyses for Licensing Evaluation", ORNL/TM-2005/39, Version 6, 2009.

[3] J.E. Atfield, ZED-2 User Facility Proposal Package, ZED2-123110-REPT-001, AECL Report, http://www.aecl.ca/Programs/Nuclear_Innovation_Networks_Program.htm.

[4] A.L. Schwarz , R.A. Schwarz, and L.L. Carter, "MCNP/MCNPX Visual Editor Computer Code Manual for Vised Version 22S", Visual Editor Consultants (2008).

[5] "C# Language Specification Version 3.0", Copyright ® Microsoft Corporation 1999-2007.

[6] J. E. Atfield, *28-Element Natural UO_2 Fuel Assemblies in ZED-2 (ZED2-HWR-EXP-001)*, in International Handbook of Evaluated Reactor Physics Benchmark Experiments (IRPhEP), OECD – NEA/NSC/DOC(2006)1, CD ROM March 2011 Edition.

[7] Extensible Markup Language (XML) 1.0, 5th Ed., The World Wide Web Consortium (W3C), http://www.w3.org/TR/REC-xml/, (2008).

[8] M. Fowler, "*UML Distilled*", 3rd ed., Addison-Wesley, ISBN 0-321-19368-7.

[9] J.L. Conlin, W. Ji, J.C. Lee, and W.R. Martin, "Pseudo Material Construct for Coupled Neutronic-Thermal-Hydraulic analysis of VHTGR", *Trans. Am. Nucl. Soc*., **92**, 225-227 (2005).

[10] J.C. Chow, F.P. Adams, D. Roubstov, R.D. Singh, and M.B. Zeller, "Nuclear Data and the Effect of Gadolinium in the Moderator", submitted to *AECL Nuclear Review*, Inaugural Edition, Atomic Energy of Canada Ltd., (2012).

[11] M. B. Chadwick, *et al.*, "ENDF/B-VII.0: Next Generation Evaluated Nuclear Data Library for Nuclear Science and Technology", Nuclear Data Sheets, Vol. **107**, pp. 2931-3060, (2006).

[12] R. C. Little and R. E. MacFarlane, "ENDF/B-VI Neutron Library for MCNP with Probability Tables," LA-UR-98-5718, Los Alamos National Laboratory (1998).

[13] D. Altiparmakov, "*ENDF/B-VII.0 Versus ENDF/B-VI.8 in CANDU® Calculations*", in Proceedings of PHYSOR 2010 – Advances in Reactor Physics to Power the Nuclear Renaissance, Pittsburgh, Pennsylvania, USA, May 9-14, 2010, on CD-ROM, American Nuclear Society, LaGrange Park, IL (2010).