B-T-3292

MODULAR, OBJECT-ORIENTED REDESIGN OF A LARGE-SCALE MONTE CARLO NEUTRON TRANSPORT PROGRAM

B. S. Moskowitz

Contract No. DE-AC11-98PN38206

BETTIS ATOMIC POWER LABORATORY WEST MIFFLIN, PENNSYLVANIA 15122-0079

Operated for the U. S. Department of Energy
by BECHTEL BETTIS INCORPORATED

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

MODULAR, OBJECT-ORIENTED REDESIGN OF A LARGE-SCALE MONTE CARLO NEUTRON TRANSPORT PROGRAM

Bradley S. Moskowitz
Bechtel Bettis Inc.
P.O. Box 79
West Mifflin, PA 15122-0079
moskowit@bettis.gov

ABSTRACT

This paper describes the modular, object-oriented redesign of a large-scale Monte Carlo neutron transport program. This effort represents a complete "white sheet of paper" rewrite of the code. In this paper, the motivation driving this project, the design objectives for the new version of the program, and the design choices and their consequences will be discussed. The design itself will also be described, including the important subsystems as well as the key classes within those subsystems.

In addition, an important aspect of the project, the strategy followed to allow the new version of the program to be integrated smoothly into a large existing external system structure, will also be discussed. This part of the project was critical to allowing the new version of the program to serve as an eventual replacement for the older version.

This paper concludes with a brief discussion of our implementation strategy and the programming decisions made along the way. While the new design is complete, the project is still in progress, in terms of the full implementation and testing of the production code. However, most of the program features have been implemented and tested and we have been encouraged by the results thus far, both in terms of correctness and performance, and in the prospects for reaching our major design objectives.

1. INTRODUCTION

Faced with the large and complex RCP legacy code[4] (first full production version circa 1978), which performs essential Monte Carlo neutron transport calculations efficiently and accurately, but which also requires an ongoing large amount of manpower for maintenance (porting to new platforms, improving performance to take advantage of scalar cache-based parallel systems, add-

ing new capabilities), we decided that in the long run, a complete "white sheet of paper" redesign and rewrite of the program would be worth the investment and the risk involved.

As described in Section 2, our motivation for redesigning and rewriting the program from scratch was to reduce the ongoing maintenance costs associated with the program, improve its portability, and to make it possible in the future to expand the capabilities of the program more quickly and efficiently than could be done with the existing program. Our design objectives were chosen based on these concerns, and we decided to pursue a modular, object-oriented design. Our full list of design objectives is discussed in Section 3.

Section 4, a large section of this paper, concerns our design decisions, how and why the program was divided into its modular subsystems, what the responsibilities of each subsystem were, and how the subsystems were further subdivided into object classes which map directly into the classes used in the implementation of the program.

Another important aspect of the new design was our strategy for integrating the new version of the program into a large existing system structure consisting of several support programs providing input data such as geometry models and cross section libraries and several post-processing programs providing the ability to combine results from multiple jobs and select particular combinations of results for further processing. This is particularly important given the large amount of existing input data for RCP and level of proficiency in terms of preparing RCP input and analyzing RCP output which its users have developed over many years. This is discussed in Section 5.

The full project is still in progress, since a fully tested production version has not yet been completed. However, the design is complete and most of the features in the design have been implemented and tested. Section 6 briefly describes the implementation which we have chosen for the new version of the program and some of our programming decisions. In particular, the majority of the program has been implemented in Fortran 90 using what other authors have described as an "object-based" approach, since Fortran does not allow true inheritance.[1,2,3] The reasons for choosing not to implement the program in a fully object-oriented manner, using C++ for example, which would have matched the object-oriented design more closely, are discussed briefly in Section 6.

Finally, this paper concludes with a discussion of observations based on experiences in working on this project.

2. MOTIVATION FOR PERFORMING A MAJOR REDESIGN

RCP was designed in the early 1970's when computer systems and software techniques were very different from today. Core memory size was limited and great efforts were made to limit the amount of memory used by programs in order to fit large problems onto the computer. Overlays were used so that various sections of the code could occupy the same memory space but be explicitly paged in and out of memory for various portions of a calculation.

The language of choice, Fortran 66 and later Fortran 77, restricted variable names to six characters and had no explicit support for dynamic allocation of memory. As a result, meaningful variable names were often unavailable. Elaborate schemes were used to manually achieve the equivalent of dynamic memory allocation. A typical strategy was (and still is today in many programs) the following: create a single large array containing as much memory as possible (often by placing it in blank common), explicitly break this array up into subsections of dynamically set (i.e. runtime) sizes, and equivalence these subsections with the array names on which actual calculations are performed. This technique allowed detailed control of memory usage by the programmer, but also required great care to ensure that no arrays overlapped. In addition, this technique was sometimes taken further, and sections of the large array were equivalenced to arrays of different native data types. This non-portable technique produced reals, integers, and sometimes even characters and logicals which shared the same memory locations. In many cases program correctness depended on particular exact sizes in memory for these various data types.

RCP made full use of all of these techniques. This was done with good reason in the 1970's when the program was first designed and written, based on the available technology at the time. The fact that the program is still heavily and productively used today is a testament to the high quality of that original design and implementation. Nonetheless, as porting of the program to a new platform has become a nearly annual process, and as user requests for new features stream in, we found it increasingly difficult and costly, in terms of time and manpower, to maintain this large and complicated program at a pace which could continue to satisfy the needs of the users. In particular, ensuring that changes affecting any of the arrays which were "dynamically allocated" sections of a larger array did not produce overwrites of other array sections became an increasingly tedious and difficult process. Problems were difficult to track down since changes causing an overwrite could produce side-effects which did not show up until much later in very different parts of the program. Another growing problem was the long and steep learning curve required for anyone new to assist with program maintenance and development.

Development of new features for the program suffered as a result, and efforts to take advantage of clear opportunities for parallelization of RCP on new scalar cache-based systems were stalled by frequent difficulties associated with the data structure described above. Given this situation, we considered the option of rewriting only the RCP data structures while retaining large parts of the original program. However, it appeared that this "conservative" approach would be just as difficult as starting completely from scratch and would also not produce as large a long-term benefit. Therefore, we decided to undertake the project of redesigning and rewriting the RCP program.

A critical element in being able to start from scratch was starting from a clear understanding of the FUNCTIONAL capabilities of the program, independent of how they were actually implemented. This depended on frequent discussions and assistance from experts and original programmers of the program. This invaluable assistance allowed us to understand the functional purpose of many of the original routines without always requiring us to explicitly examine the source code at every step (although occasionally such detailed inspection of the source code was unavoidable). Fortunately, many of the basic algorithms used by the program were well understood by our developers and we did not always need to get into the details of how they were implemented. The lesson here is that in planning such an effort, it helps tremendously to do so while the program-

mers of the original program are available to answer questions and provide guidance. Such assistance can be absolutely critical to the success of the effort.

3. DESIGN OBJECTIVES

Given the motivation discussed in the preceding section, the following design objectives were chosen, **in order of importance**, to guide the redesign of RCP. The developers strived to follow these objectives throughout the design and implementation phases of the project. A great deal of thought was put into the selection of these objectives and they were "enforced" at the developers' weekly source code review meetings in which we would challenge each other's decisions and suggest changes as needed. Following each objective is a brief description and discussion. (Note: Some of these objectives extend beyond design and include implementation issues as well).

3.1 MODULAR, OBJECT-ORIENTED STRUCTURE AND DECOMPOSITION

- The program should be organized into a set of top-level subsystems each responsible for a particular well-defined aspect of the program as a whole. Each subsystem defines a set of object classes each of which is responsible for a specific aspect of the subsystem. Classes should represent either clearly defined abstract data types or else serve as containers for closely related sets of services. Class names should be nouns.

Based on the size and scope of this project, and the large amount of data involved, we decided that this kind of modular, object-oriented decomposition provided the best way to manage the data and processing in a clear and easy to maintain manner. Numerous experts in the fields of computer science and software engineering, and experienced programmers have highly recommended such techniques and we decided to heed their advice.[5,6,7] In addition, we were encouraged by the experiences of others in writing an object-oriented Monte Carlo neutron transport program.[9]

3.2 UNDERSTANDABILITY AND READABILITY

- The program should be written in such a way that it would be relatively easy for someone with a reasonable base of knowledge about nuclear physics, Monte Carlo methods, and object-oriented programming techniques (including the developers themselves) to pick up a section of source code and understand what the program is doing. Names of classes, methods, and variables should be chosen so that someone reading the names and guessing what they represented would be as likely as possible to guess correctly. Source code should also follow a consistent style, including accurate comments where needed.

It is important to note that we placed this objective high on the list, even ahead of correctness. We felt that given the rapid rate of change (adding new features) expected for this code once the initial redesign was completed, it was essential that the design as well as the source code itself be easy to understand in order to reduce the chance that mistakes would be made when new features were added, even if the original version were entirely correct.

3.3 CORRECTNESS AND COMPLETENESS

- The program should correctly satisfy all of its requirements. All of the necessary features from the original version of RCP should be correctly implemented in the new version such that the results from the two versions are either statistically equivalent, or else in the case of differences, these differences are understood and explained by intentional changes which have been made.

Clearly this is a desired objective. It is a property of both the design and the implementation of the design. The reason that it follows the preceding objectives has already been discussed above.

3.4 ALGORITHMIC PERFORMANCE ON SCALAR, CACHE-BASED, MULTIPLE-PROCESSOR SYSTEMS

- The program should be designed and written to run efficiently on scalar cache-based systems. It should also be written so that it is able to take advantage of multiple processors. This means that the overall solution algorithm should parallelize well and that lower-level solution algorithms should be efficient ones for scalar systems. In addition, results should not change depending on the number of processors.

This design goal is concerned with algorithmic issues affecting overall performance on certain types of systems. The selection of a particular type of system is an acknowledgement that performance does matter for this sort of program. We cannot simply develop an easy to read, correct code which runs too slowly to be useful in practice. The focus on scalar, cache-based, multiple processor systems represents our observation that these types of systems now dominate the high-performance scientific computing arena, and our expectation that this trend will persist.

3.5 PORTABILITY

- The program should be designed and written to be highly portable. The programming language and parallelization library chosen should follow widely accepted standards. System-specific portions of the program should be isolated in wrapper classes and preprocessing directives should be used so that a single source code is valid on all systems. Use of 'tricky' techniques which may not work on all systems are to be avoided unless well justified and well documented.

Despite the performance targets of the preceding objective for particular types of systems, we recognize that systems can change rapidly today and the costs of porting a code to new platforms can be high. This objective represents an effort to minimize future porting costs.

3.6 FLEXIBILITY AND EXTENSIBILITY

- The program is expected to have a long lifetime involving many modifications and extensions. Therefore program features which will make it easier to modify the program in the future should be favored over features which tend to lock the program in and make future changes more difficult.

This objective is relatively self-explanatory. It represents another effort to reduce future maintenance costs, and improve the "time to market" of future enhancements where possible. This objective is towards the bottom of the list simply because it can be difficult at times to evaluate which design alternative will produce a program that will be easier to modify in the future. To a certain extent the first objective, a modular object-oriented design, is related to this goal but at a higher, more abstract level.

3.7 LOW LEVEL PERFORMANCE TUNING

- It is important that the program run efficiently. Therefore, when it is possible to make low level changes to improve performance, these should be taken advantage of, but not at the cost of the preceding design objectives.

The reason that this objective is last is a belief that low level tuning often has the effect of obfuscating the design, and that improvements in compiler technology or changes in computer hardware often render such changes as irrelevant at best and harmful at worst over the course of time. We felt that low-level tuning was best left as a last resort when needed and avoided otherwise. A distinction was made however between "low-level tuning" and algorithmic decisions made for performance reasons, which we do consider to be important, and which were therefore included as the fourth objective listed above.

4. TOP LEVEL DESIGN

In terms of its design as well as its implementation, at the highest level, the redesigned RCP program is divided up into a set of relatively independent modular subsystems. The benefits of such a decomposition have been well documented by others.[5] Each subsystem is further divided up into a set of object classes which constitute the lowest logical unit in terms of the design. The key subsystems are the following:

1. Main
2. Job Control
3. Nuclear Model
4. Space Model
5. Composition Definitions
6. Neutrons
7. Scoring

In addition, several smaller subsystems provide general purpose facilities. These include the following: Basic Modules (file handling, error handling, precision settings), System Utilities (com-

mand line access, memory leakage detection, message passing), Math Utilities (search and sort, constants, normalization), Random Numbers (generation, transformations), and Timing.

In the design, these subsystems served as logical groupings of the data and processing required in the program. In the implementation, the subsystem boundaries were clearly defined by creating a subdirectory under the main source code directory for each subsystem. All of the source code files associated with each subsystem were then placed in the corresponding subdirectory. While this may seem to be a minor point, it served to emphasize that the implementation was following the modular design and it forced the developers to explicitly assign each source code file to a specific subsystem. (Note: As will be described later, each source code file corresponded to a single object class definition.)

A description of each of the major subsystems follows.

4.1 MAIN SUBSYSTEM

The **Main** subsystem consists exclusively of the main program itself. This subsystem acts as the top level manager over all the others, controlling the flow of execution when the program runs. The overall flow of execution in this program is iterative at the highest level, following neutron generation iterations to completion. The program is parallelized by distributing neutrons to parallel processes at the next level. At the third level, within each parallel process, the program is neutron event-based, similar to the approach described by Brown and Martin.[10]

At the third, event-based, level, an event loop for each parallel process selects the next event (e.g. collision, tracking step) to be computed based on the number of neutrons waiting for each event. The selected event is then completed for all of the neutrons waiting. This loop repeats until all of the neutrons have either been absorbed, killed (by Russian Roulette), or have leaked out of the problem domain. At the second, parallel process, level, an iteration ends only when all of the parallel processes have exited their event loops. Finally, at the top level, a job completes after a selected number of iterations have been completed. All of this runtime control logic is handled within the **Main** subsystem.

4.2 JOB CONTROL SUBSYSTEM

The **Job Control** subsystem is responsible for managing the data used for controlling a job. This includes user input options such as the number of iterations to run, whether a job is a fixed source or an eigenvalue job, the neutron batch sizes to use, and the kinds and amount of output data a particular job should produce. It also includes runtime data used by the main program such as the current iteration counter, and an event counter. Placing this data in a subsystem separate from the main program allows the overall program flow logic to be separated from the mundane details of managing the control data, which is performed within this subsystem.

This subsystem does not include the data defining the spatial model, the nuclear model, or the compositions present, since these categories of data are managed by the space model, nuclear model, and composition definition subsystems themselves respectively.

4.3 NUCLEAR MODEL SUBSYSTEM

The **Nuclear Model** subsystem is responsible for the data and processing involved in handling reactions between neutrons and nuclides. This includes, for example, the energy mesh on which microscopic cross sections are tabulated, the microscopic cross section data itself, linear interpolation calculations for the cross sections at a particular energy, exit energy and angle calculations for elastic and inelastic scattering, and also basic nuclide data such as name and atomic mass.

This is a large subsystem which includes several classes, which interact in a variety of ways. There is the EnergyModel class, with a single object representing the energy model in use for a particular job, the Nuclide class, with one object for each nuclide defined for a job, and the FissForNuc and MicroForNuc classes which each define objects which are contained as components of Nuclide objects. The FissForNuc class is concerned with the fission data for a nuclide such as its nu-values and its fission spectrum. The MicroForNuc maintains all of the microscopic cross section data for a nuclide and performs single nuclide cross section calculations as well. There is also an EnergyPt class which defines objects representing particular energies within the energy model, and a MicroTotAtE class which defines transient objects containing total microscopic cross sections for all nuclides for particular EnergyPt's. There are also classes to support different kinds of scattering such as the ElasticScattAng class.

The main value of this approach is that each of these classes is responsible for a small, specific piece of the entire nuclear model. Therefore, it is relatively easy to construct and maintain each class. Objects in different classes can only interact in a limited and controlled manner - by calling methods (functions or subroutines) associated with each of the classes. Building up the pieces into an entire functioning system can be complicated, but maintaining and managing the collection is simplified by the limitations on how the pieces connect. In terms of the design of the Nuclear Model subsystem, this extensive use of classes allowed design decisions to be made at a higher level than could have been done if all the data resided in one location. The selection of the classes was made to facilitate higher-level reasoning about the nuclear model as a whole. In addition, we found that encapsulation of the data relevant to each class within the objects of that class provided large maintenance benefits since we could be confident that if some data became corrupt, it must have been caused locally within the class rather than via a side effect of an operation in another class acting on shared global data, since the program contains virtually no shared global data.

4.4 SPACE MODEL SUBSYSTEM

The **Space Model** subsystem defines the three dimensional space in which compositions (defined in the **Composition Definitions** subsystem) appear and through which neutrons travel, interacting with the nuclides in the compositions. The RCP spatial model capabilities will not be discussed in full here, since they are outside the scope of this paper, but a brief description follows in terms of the important classes:

A set of SpatialElement objects are the basic building blocks for an RCP spatial model. Each of these refers to a BaseGrid object paired with a CoordSystem object.

A BaseGrid object defines a region of three dimensional space which is divided up into cells by an overlaid mesh. The model allows for standard geometrical shapes for these base grids, such as rectangular blocks, cylinders, and spheres. These are handled in the object-oriented model as "subclasses" of the BaseGrid "superclass."

A CoordSystem object positions a BaseGrid within three dimensional space, allowing arbitrary rotation, reflection, and translation.

There are additional classes to represent locations in the space model (SpaceLoc), define global boundary surfaces such as reflecting boundaries (GlobBndrySurf), and allow cells to be further subdivided into subcells (CellDivision). There are also two "calculator" classes which do not contain any data, but which perform necessary calculations which did not logically fall within any of the other classes. These are the OverlapCalc class, which performs calculations to determine whether various geometric figures overlap, and the DistanceCalc class, which computes distances between different surfaces and also distances along rays to their intersections with surfaces.

4.5 COMPOSITION DEFINITIONS SUBSYSTEM

The **Composition Definitions** subsystem defines the material compositions which can appear within the **Spatial Model** subsystem and through which neutrons travel and interact. This is a relatively simple system. At the most basic level, a composition is nothing more than a list of nuclides and their densities. The Composition class is defined such that each composition object contains this data for a single composition.

There are two additional levels of complexity which have been added to the basic system described above. The first, composition templates, was added in order to reduce memory usage of the program. The second, materials, was added as a convenience feature.

It was recognized that in some cases many compositions share the same list of nuclides but contain different densities for these nuclides. Composition templates represent a way of taking advantage of this situation. A composition template object simply contains a list of nuclides. A composition can then refer to a particular composition template for its list of nuclides and provide only the associated densities, rather than providing an entire list of both nuclides and densities. For jobs with a large number of similar compositions, this can represent a large savings in terms of memory use at the cost of a relatively small increase in complexity. The current version of RCP contains a similar memory saving feature, but this is buried under many layers of indexing and data, which makes it difficult to maintain.

Materials, represented by the Material class, provide the ability to combine multiple compositions together. This is a convenience feature allowing groups of compositions to act as units. The Material class and the Composition class include many methods (subroutines and functions) with the same names, a form of polymorphism, so that objects in these classes can be used interchangeably at various points in the program. (Note: This is a simple example of a design pattern known as the "composite" pattern which is described in the book Design Patterns[8].)

4.6 NEUTRONS SUBSYSTEM

The **Neutrons** subsystem is responsible for managing the data and behavior of neutrons in a job. There is just a single large class defined in this system, which is the Neutron class. Each object in this class represents a single neutron history which lives from birth in fission to either leakage from the problem domain or death by absorption or Russian Roulette. In a job involving multiple parallel processes, each process manages its own set of neutron objects during an iteration from birth to death. (Note: In the implementation, these are handled as an array of neutrons for each process).

Each neutron contains the following component data: a unique ID value, a current position in the spatial model (represented by a SpaceLoc object), a current energy (represented by an EnergyPt object), a random number stream (represented by a RandomNumStream object), and several other components.

Organizing the neutron data in the way described above, as an array of neutron objects, represented an important design decision in our redesign effort. In the older version of RCP, the data associated with a particular neutron often appeared in the form of a value at a particular array index within an array of energies, x coordinates, y coordinates, z coordinates, and so forth. This was done because on vector computers, which represented the main high-end scientific computing platform for many years, such an arrangement of the data facilitated performing calculations on long vectors which could significantly improve performance on these systems. This layout will be referred to as the "vectorized layout" below.

As we redesigned RCP, we decided to take the approach of associating all the data for a neutron with a neutron object for several reasons. First, this fit into our overall object-oriented design in the most straightforward way. We expected that such an approach would produce a more readable and easier to maintain code than the vectorized layout. As discussed in Section 3, we made a decision to emphasize an object-oriented design, and readability and maintainability over performance concerns.

Nonetheless, we knew that we could not completely sacrifice performance in designing RCP and produce a program which any of our users would want to use. In testing early prototype implementations, however, we found that with a good optimizing and inlining compiler, the performance observed while taking our preferred approach was quite acceptable. We also found that on scalar cache-based systems in particular, this arrangement of the data could have positive benefits in terms of reduced cache misses since the data for a particular neutron generally appears closer together in memory than it would in the vectorized layout over a set of large arrays of component values. Hence, we decided to take the approach of defining the Neutron class as described above.

Note: In an earlier design iteration, we defined a NeutronSet class instead of a Neutron class. An object in this class contained data for a set of neutrons which was arranged as a set of arrays, similar to the vectorized layout described above. However, we found that the coding became awkward and was dominated by a large number of gather/scatter operations with many opportunities for coding errors. When we switched to the design using the single Neutron class, we found that the coding became simpler, more readable and easier to maintain, and we were satisfied with the

performance as well, which, on scalar cache-based systems, was often superior to the earlier design.

A large number of neutron methods (subroutines and functions) are supported, some of which act on single neutrons and others which act on arrays of neutrons. At the highest level, the *next_event* method is the main method called by the main program. This method takes an array of neutrons, selects the event with the most neutrons waiting and performs that event on all the waiting neutrons. Most of the other methods are called by this method and therefore can be made "private" to the neutron subsystem, meaning that they are invisible to the rest of the program. This provides additional modularity in the design.

Methods called by the *next_event* method include the following methods whose purpose should be clear from their names: *step_travel, delta_travel, collide, ext_cross,* and *remove_dead.* These methods in turn call more specialized methods in the Neutron class and also frequently call methods in the **Nuclear Model** and **Space Model** subsystems to perform the actual calculations required in order to move the neutrons through space and handle their interactions with nuclides and boundaries.

4.7 SCORING SUBSYSTEM

The **Scoring** subsystem handles the data and calculations associated with tallying and computing reaction rate results for neutrons based on the neutrons' interactions with nuclides in compositions which appear in the spatial model. This is a complicated system because it requires interaction with most of the other systems.

Reaction rate scores are stored using ScoreTable objects. Each of these objects contains a large array of scores for all nuclides, compositions, and energy ranges of interest. One ScoreTable is used for each kind of reaction rate (e.g. capture, scattering). In addition, there are two kinds of score tables, represented by two subclasses of the ScoreTable class. There are score tables which include statistical accumulations such as the squared sum of previous samples, so that uncertainties can be computed in addition to mean values. These are called StatTables. There are also score tables which only include current accumulations, so that only mean values can be computed, but less memory is used. These are called MeanTables.

At a higher level, all the score tables in a job are combined within a single object of the ScoreStorage class. This is where the overall management of the scoring data is handled and reaction rate calculations are performed.

An important design consideration was the handling of scoring data for multiple process jobs. Two approaches were taken, both of which had their strengths. These are described next.

The first approach was to maintain an independent ScoreStorage object used by each process during an iteration. At the end of each iteration, the results from all of the ScoreStorage objects were then summed over all processes and stored in a single master copy. This is a memory intensive solution because for N processes, N copies of the scores, which can be very large, must be stored. However, this has the advantage of reducing the amount of interprocess communication to a min-

imum during an iteration. On a shared-memory system such an approach can be wasteful but on a distributed memory system it can be essential.

The second approach was to maintain a single ScoreStorage object which all of the processes contribute to. In order to prevent clashes in accessing and updating this data, a dedicated process known as the ScoreServer accepted scoring input from the other processes, acting as ScoreClients, during an iteration, and at the end of an iteration the ScoreServer would provide any reaction rates needed by the clients to perform additional calculations. Buffering of scoring input data was used to reduce the number of interprocess communications, but even so, with this strategy a large amount of interprocess communication is required throughout each iteration. On the other hand there can be a significant reduction in overall memory use. This approach is inefficient on most distributed memory systems because of the communication overhead. However, on a shared memory system it can be very effective. In fact on systems with optimized message passing libraries which effectively convert interprocess communications into shared memory accesses we found little performance difference between this approach and the first, but a large savings in terms of memory use.

The new version of RCP supports both of the approaches listed above, with a simple user input choice selecting the approach used for a particular job. The modular design of the program facilitated our ability to support both approaches in the same code and make the choice a runtime user option.

5. INTEGRATION STRATEGY

In order for the new version of RCP to be a useful replacement for the older version, an integration strategy was necessary. The older version expects its nuclear library data, spatial model input data, and control data to be received in a particular format. A large body of existing data is available in these formats for RCP, and preprocessor programs related to RCP are available for producing and manipulating such data. Similarly, RCP produces output files in particular formats which postprocessor programs are able to read and process. This posed a design problem for the new version of RCP which is described next, along with our solution.

A design problem for the new version of RCP was that the existing input and output file formats for the older version of RCP were closely related to the internal data structures of the program. Since the new version of the program contains completely redesigned internal data structures, it would have been difficult and awkward to retain the prior formats. However, replacing the file formats would introduce obvious integration problems into the larger systems in which RCP was used.

In keeping with our plan to completely rewrite RCP, we decided also to completely reorganize the input and output file formats. We felt that our design goals of modularity, readability and maintainability supported this decision. However, this left us with major integration concerns. These were addressed on the input side by designing and implementing a separate translator program which had a single task: accept RCP input files in their original formats and produce input files in the new formats as its output. Placing this functionality in a program separate from RCP served to

maintain the consistent modular design of RCP while moving the complications associated with the conversion process into a single isolated program.

A similar approach was taken for output files. We created new output data formats for RCP and then also created new versions of the RCP postprocessor programs. These new versions could accept the new RCP results as input and produce output results which are either identical or extremely similar to the equivalent outputs from the original RCP postprocessor programs. This should allow the RCP users to work with either the new or old version of RCP without making major adjustments to their overall process.

A trade-off cost associated with this strategy is that it will require maintaining additional programs for as long as conversion to and from the older data formats is necessary.

6. IMPLEMENTATION AND PROGRAMMING DECISIONS

It can be difficult at times to separate design from implementation since the two often overlap in practice. As noted in Section 3, several of the RCP design objectives dealt with implementation as well as design issues. In this section, we will go beyond the main focus of this paper, which is the RCP design, and briefly discuss some of the major implementation decisions.

One choice was to make the implementation "look" like the design in the following ways: each class in the design corresponded directly with a single source file which implemented the class (see below for additional details), and all of the new RCP source files were placed in subdirectories corresponding directly with the subsystems defined in the design and described in Section 4. This forced the developers to explicitly assign each source file to a specific subsystem in the design. It also forced them to assign each source file either to a specific class in the design or else to recognize the class as an "implementation" class which did not have a corresponding class in the design.

A second choice was to follow a strict set of coding style guidelines. Part of the reason for this was to ensure that all of the source code followed the same "object-based" style, as discussed next. Another reason was to improve the readability and maintainability of the code, based on the second design objective (Section 3.2). Weekly review meetings were held so that the developers could check each other's work and make sure that style guidelines were being followed. In addition, design questions were often discussed as well.

A third implementation choice was to rely on widely accepted standards for the implementation. This was done to improve portability, the fifth design objective (Section 3.5). Thus, parallelization was handled via the standard Message Passing Interface (MPI), standard C preprocessor (CPP) directives were used for inserting precompilation directives in the code, and as described below, the standard Fortran 90 programming language was chosen as the main implementation language.

Another important choice was our decision to follow a phased implementation strategy. Rather than attempting to implement all the functionality at once in the new version of RCP, we instead

implemented the program in several phases consisting of subsets of the full functionality. Each phase was tested before moving forward to the next phase.

Finally, the most significant implementation choice was to program the new version of RCP in Fortran 90 instead of C++. This was a difficult decision. The design is fully object-oriented, which lends itself logically to a C++ implementation. However a combination of several factors described below caused us to choose a Fortran 90 implementation. Fortran 90 does not fully support object-oriented programming, primarily because there is no mechanism for implementing true inheritance relationships between classes. However, Fortran does support user-defined types, scope restrictions on variables through the use of modules, and overloaded function naming.[11] Several authors have described a form of programming in Fortran 90 known as "object-based" which makes aggressive use of the new Fortran 90 features to produce programs which are similar to fully object-oriented C++ programs, but without true inheritance.[1,2,3] This is the style we adopted. As mentioned earlier a key component to making this work was for the developers to follow a strict set of style guidelines.

Our reasons for the decision to use Fortran 90 rather than C++ included the following: a) As mentioned earlier, we had a strong desire to use widely accepted standards for the implementation. At the time development work started, the C++ standard had not yet been approved and thus we had concerns about the lack of portability between C++ compilers on different systems. b) We feared that a completely object-oriented implementation in C++ would suffer from major performance problems, while a Fortran 90 implementation would represent more of a reasonable compromise between performance and our other key objectives.

Our decision to program using an "object-based" approach with Fortran 90 has worked out well in terms of the performance we have observed, but it has caused some difficulties in implementing the design due to missing features of the language such as inheritance, templates, and built-in support for constructors and destructors. If we had to make the same decision again today, either C++ or Fortran 90 would be a reasonable choice, but we would probably choose C++ because of its wider set of language features.

7. PRELIMINARY EVALUATION OF OUR SUCCESS

Ultimately the true measure of the success of this redesign project will be how well the new version of the program is able to meet the needs of its users, both in terms of features and performance, and how much cost is associated with maintaining the program and adding new features. While we expect that the modular object-oriented design will lower maintenance costs compared with the old version of RCP and that the turnaround time for adding new features to the program will be reduced, we will not know whether this proves to be true until time has passed and we are able to observe how well the design holds up as it is placed under stress by use, modification, and extension.

We can make some preliminary evaluations however. First of all, the amount of time and effort required thus far to complete the design and the implementation have been close to our original expectations. Second, the correctness of the code has been confirmed by detailed statistical com-

parison of benchmark results for the new version of the program against the older version. Third, performance testing of the new version of the program has thus far indicated that its single processor performance is about 50% faster than the older version of the program for the largest jobs tested. In addition, linear, or in some cases super-linear, speedups have been observed in tests of the new version of the program on up to 20 parallel processors.

CONCLUSIONS

This paper presents a road-map description of the effort to compete a modular object-oriented redesign of the RCP Monte Carlo neutron transport code. In the discussion, the motivation for this project was explained, major design objectives were listed, the top level design was described, integration and implementation decisions were discussed, and finally a preliminary evaluation of our success was made.

Our adherence to a carefully chosen set of design objectives, as listed in Section 3, has been a key factor in guiding the project. In addition, frequent assistance from programmers of the older version of RCP in explaining the functional characteristics of the program have saved us an enormous amount of effort which would have otherwise been required in detailed examination of the source code of the program.

In terms of implementation, weekly source code review meetings have been valuable in terms of ensuring that the design was faithfully implemented and establishing consistent coding styles among the developers. Our decision to program using an "object-based" approach with Fortran 90 has worked out well, we believe, in terms of the performance we have observed, but it has caused some difficulties in implementing the design due to missing features of the language such as inheritance, templates, and built-in constructors and destructors.

Our advice to others who are faced with a legacy code which costs a large amount of manpower and time to maintain and which still commands a large user base, is to consider a similar project, especially if the developers of the original program are available to help in explaining the functional properties of the code.

ACKNOWLEDGEMENTS

This work was supported by the U.S. Department of Energy under Contract No. DE-AC11-98PN38206.

Thanks to the rest of the team which has worked on the design and implementation of the new version of RCP: W. R. Nichols, L. J. Tyburski, and M. L. Zerkle.

Thanks to L. A. Ondis II, whose support and frequent advice as one of the original developers of the RCP program have been invaluable.

REFERENCES

[1] M. G. Gray and R. M. Roberts, "Object-Based Programming in Fortran 90," Computers in Physics 11, 355-361 (1997).

[2] L. Machiels and M. O. Deville, "Fortran 90: An Entry to Object-Oriented Programming for the Solution of Partial Differential Equations," ACM Trans. Math. Software 23, 32-49 (1997).

[3] C. D. Norton, V. K. Decyk, B. K. Szymanski, "On Parallel Object Oriented Programming in Fortran 90," ACM SIGAPP Applied Computing Review 4 27-31 (1996).

[4] N. R. Candelore, R. C. Gast, L. A. Ondis II, "RCP01 - A Monte Carlo Program for Solving Neutron and Photon Transport Problems in Three-Dimensional Geometry with Detailed Energy Description," WAPD-TM-1267 (1978).

[5] S. McConnell, "Code Complete," Microsoft Press, Redmond, WA (1993).

[6] B. Meyer, "Object-Oriented Software Construction," 2nd Ed., Prentice Hall PTR, Upper Saddle River, NJ (1997).

[7] C. S. Horstmann, "Practical Object-Oriented Development in C++ and Java," Wiley Computer Publishing, New York, NY (1997).

[8] E. Gamma, R. Helm, R. Johnson, J. Valissides, "Design Patterns - Elements of Reusable Object-Oriented Software," Addison-Wesley, Reading, MA (1995).

[9] S. R. Lee, J. C. Cummings, S. D. Nolen, "Building a Transport Code using POOMA and Object-Oriented Methods," X Division Research Note, LANL, XTM-RN(U)96-003 (1996).

[10] F. B. Brown. W. R. Martin, "Monte Carlo Methods for Radiation Transport Analysis on Vector Computers," Progress in Nuclear Energy 14 269-299 (1985).

[11] J. W. Kerrigan, "Migrating to Fortran 90," O-Reilly & Associates, Sebastopol, CA (1993).