CODE GENERATION OF RHIC ACCELERATOR DEVICE OBJECTS[*]

R.H. Olsen, L. Hoff, T. Clifford, RHIC Project, Brookhaven National Laboratory, Upton, NY, 11973-5000,USA

Abstract

A RHIC Accelerator Device Object is an abstraction which provides a software view of a collection of collider control points known as parameters. A grammar has been defined which allows these parameters, along with code describing methods for acquiring and modifying them, to be specified efficiently in compact definition files. These definition files are processed to produce C++ source code. This source code is compiled to produce an object file which can be loaded into a front end computer. Each loaded object serves as an Accelerator Device Object class definition. The collider will be controlled by applications which set and get the parameters in instances of these classes using a suite of interface routines. Significant features of the grammar are described with details about the generated C++ code.

I. INTRODUCTION

The accelerator controls architecture for the Relativistic Heavy Ion Collider (RHIC) being built at Brookhaven National Laboratory (BNL) uses the standard model with two conceptually distinct layers [1]: UNIX workstations provide the platform for the console-level computers (CLCs). These are networked with the geographically distributed VME systems running real-time operating systems that serve as front-end computers (FECs)[2]. The FEC supports an object-oriented paradigm using C++ object classes called accelerator device objects (ADOs). These ADOs provide a software abstraction of the underlying collider hardware.

Each ADO class comprises control points represented as parameters, and a standard set of operations with a consistent interface by which applications can examine or manipulate those parameters and the underlying hardware. The fact that the way the parameters are represented, and the interface to them, are highly consistent across ADOs allows us to gain considerable leverage through the use of code generation: a few lines of code specifying a parameter's characteristics can be used to generate the many lines of code used to implement that parameter and its interface within the larger context of a C++ class.

A. Adogen and .rad files

The program used to generate the C++ source code is called *adogen*. It is itself built, in part, using two widely available code generation utilities. Lex and yacc[3] are used to generate the lexical analysis and parsing code and the remainder of the program is coded in C++. Adogen gets its input from files which, by convention, have a ".rad" (for <u>R</u>HIC <u>A</u>DO <u>D</u>efinition) filename extension. The .rad files are text-only files which are easily administered using any of the standard source code control packages available - we have been using RCS at RHIC.

B. ADO Hierarchies

ADOs are conceptually divided into Concrete and Abstract types. A Concrete ADO can be accessed from the console level computers through libraries of routines which are used to examine or manipulate its constituent parameters. The Abstract ADOs are not directly accessible themselves, rather, they serve as a convenient way of packaging parameters. Different ADOs can share parameter collections by referencing the same abstract class as a parent. This allows for hierarchies of ADOs.

Hierarchies are implemented in abstract ADOs when a derived ADO class inherits parameters from its parent, possibly adding new parameters itself. Abstract ADOs define the data, and Concrete ADOs inherit this data from the Abstract parent(s) and can augment the methods that access the data in order to handle the specifics of the underlying hardware. Figure 1 shows an example ADO hierarchy. The base *ADO* class comprises parameters common to all ADOs, for example the Name and a Description. The *ProfileMonitor* is an abstract class which inherits those base parameters and adds NHScanLines and NVScanLines (Number of Horizontal/Vertical Scan Lines). The final abstract class *TwoDProfileMonitor* adds Horizontal and Vertical projections. In this example *Flag* is the concrete ADO. Applications can access any of the parameters which the Flag ADO inherits from its parents through an instance of the Flag ADO class, whose methods define what those accesses actually do.

II. ADOGEN GRAMMAR

One of the primary goals of adogen is to allow the code for ADOs and ADO hierarchies to be created and modified quickly by specifying values for some minimum number of variables. The adogen grammar keeps the simple things simple. For instance,

[*]Work performed under the auspices of the U.S. Department of Energy

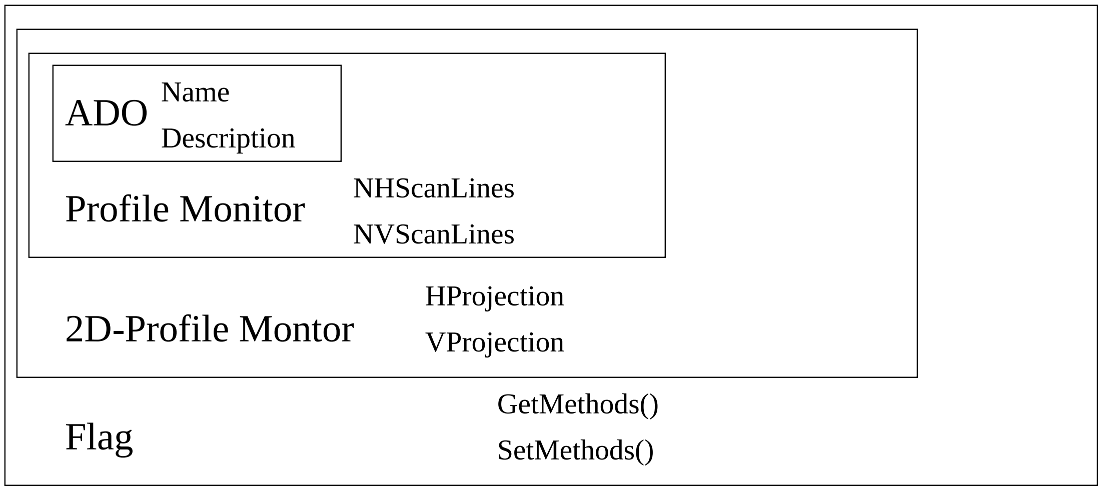

Figure. 1. Flag ADO hierarchy.

all of a parameter's important characteristics are specified in one place in the .rad file. A parameter's data type only has to be specified once and wherever the code that adogen generates is dependent on that type, consistency is guaranteed. Change that type specification and the generated code will be updated accordingly. Another goal of adogen is to eliminate the need to edit the generated code. (If there is no editing of the C++ code subsequent to its generation then the more compact rad files can become the source that is maintained in version control.) To this end the grammar for ADO definition is fairly flexible, and offers a rich set of keywords. Descriptions of some of these follow:

A. Keywords

- CONCRETE - the name of the concrete class.
- ABSTRACT - the name of the concrete class's abstract parent.
- PARENT - If an abstract class's parent is not the base ADO class, the parent name must be specified. (For hierarchies.)
- PARAMETER - the name of an individual parameter, and a COUNT if it is an array. In addition to this a complete parameter specification includes:

CATEGORY - Each parameter is assigned a category which determines a set of support properties for the parameter. All of the categories have support properties such as a description, and a format string. Continuous Settings add more information such as high and low limits. Discrete Settings add Legal Values, etc. The recognized categories are: BASIC, CONT_SETTING, CONT_MEAS, DISC_SETTING, DISC_MEAS, CONFIG_DATA, USAGE_DATA and DIAG_DATA.

TYPE - a data type. The recognized types are: CharType, UCharType, ShortType, UShortType, LongType, ULongType, FloatType, DoubleType, StringType, StructType.

READWRITE - specifies whether the parameter is writable or just readable (R, W).

DESC - text describing the parameter is optional, but console level utilities know how to access this support property and it can be useful as "help" text.

SETCODE - any specialized set code for the parameter is specified using this keyword. While ADO infrastructure code provides default set and get codes for each parameter which accesses memory locations, any special driver or hardware level requirements for a particular parameter can be met by providing more set code.

GETCODE - any specialized code which may be necessary for retrieval of information from the hardware.

- MEMBERCODE - This code becomes member code in the ADO's C++ class. Different set or get codes can share the same member code.
- INITCODE - This code is executed when the ADO instance is first created.
- THRUCODE - This code is passed through "as-is".
- EVENTCODE - This code is made accessible to an Event Management subsystem. Machine events such as interrupts and timers can be "connected" to this code, which has access to the ADO's parameters and particulars about the underlying hardware, so that it is executed when the event occurs.

B. Syntax

The syntax for adogen resembles "C++" code. The ADO designers at RHIC are familiar with C++ and quickly became comfortable with the requirements of adogen. The same development environment can be used for either C++ or adogen grammar. Since line number information from the .rad file is included in the generated code, compiler error messages can be processed and debuggers can reference code in the .rad file rather than in the generated code.

Here is an example rad file:

```
//!- device.rad -----------------
// This example rad file shows some
adogen // input that can be used to
build ADOs.
ABSTRACT = two d profile;
CONCRETE = flag;
ARGS = int gain, char * string2;
VERSION = $Revision: 0.0 $;
DESCRIPTION = A simple, sample
ADO;
// Member data (ADO globals)
MEMBERDATA = long int _last;
PARAMETER gain {
CATEGORY = CONT_SETTING;
TYPE = ShortType;
READWRITE = W;
DESC = allows an electronic gain to be set;
SETCODE = {
// set the gain of our device
const int writeVal = write(_fd, (char *) setGain, strlen(setGain));
if ( writeVal != strlen(setGain) ) return ADO_HW_FAILURE; // hardware error
return OK;
} // ENDCODE
}
PARAMETER array size {
CATEGORY = BASIC;
TYPE = ShortType;
READWRITE = W;
SETCODE = {
// set the size of our array
variableLengthArray temp(p,
_arraySize); _arrayParameter =
temp;
} // ENDCODE
PARAMETER arrayParameter [ ] {
// Parameter uses a variable length
array CATEGORY =
CONT_MEAS;
TYPE = ShortType;
READWRITE = R;
DESC = variable length array values;
}
THRUCODE = {
 extern "C" void updateNow(char * codes) {
// Tell the device to update
const int writeVal = write(_fd, (char *) codes, strlen(codes));
if ( writeVal != strlen(codes) ) return ADO_HW_FAILURE; // hardware error
return;
 }
} // ENDCODE
EVENTCODE  updateStuff= {
// update some stuff
updateNow(‘‘0xff00’’);
} // ENDCODE
```

```
ENDDEF;
// end device.rad ----------------
```

The rather simple ADO described in the above .rad file has three parameters. It also has memberdata, thrucode, and some eventcode.

The *ARGS* keyword allows the ADO designer to specify arguments to the ADO constructor; values for these arguments can be listed in configuration files which will be used when a particular ADO instance is loaded. The *VERSION* and *DESCRIPTION* can be useful in CLC utility programs. The *arrayParameter* shows how a variable length array is specified by using empty square brackets; a fixed length array's length is specified by putting a number within those brackets. The *EVENTCODE* provides a routine called "updateStuff" where the Event subsystem can access the updateNow() *THRUCODE.*

III. ADOGEN OUTPUT

When the above file is used as input, adogen generates files for both the Abstract and Concrete classes. These files contain all the source code, header information, and Makefile details necessary to compile an ADO class which can be loaded into a Front End Computer. Simply invoking the UNIX make utility with the concrete class's make file gets everything compiled and linked. Then instances of this class can be dynamically added and removed from the system, and the parameters can be set and obtained from applications running at the console level. The interface to an ADO and its parameters is well known, so even generic applications, which are not privy to the specifics of the hardware involved, can be used to view and adjust values.

IV. CONCLUSIONS

Code generation allows the designer of an Accelerator Device Object class to leverage a concise description of the ADO's parameters and the particulars of the hardware interface(s) involved into a full-featured ADO with all of the attendant infrastructure elements and procedures fully implemented. This is much more efficient and less error-prone than either individually coding each class from scratch or using a "boiler-plate" method where one has to fill in the blanks with parameter information that must be kept consistent through several different representations. Among the future enhancements being considered is the possibility of using the .rad files as sources for database information about the ADOs in RHIC. In the same way that Adogen is used to generate C++ code from these files, another program can be used to generate SQL code for database access and updates.

One advantage of having the .rad files be the "authority" on what is the current version of an ADO is that they are text-only files. They can be manipulated and examined using standard UNIX tools. This allows, for example, using RCS for version control.

Adogen is currently being used at RHIC to define, develop and maintain more than twenty different ADO Classes. The median size of one of these rad files is about 200 lines; these 200 lines generate more than 1700 lines of C++ code and makefile information. RHIC benefits from the savings in programer time and effort and from the improved concordance between different ADO class implementations.

References

[1] L.T. Hoff and J.F.Skelly, Accelerator Devices at Persistent Software Objects, Nucl. Instr. and Meth. in Phys. Res. A 352 (1994), 185-188
[2] D.S. Barton, Controls for RHIC, a Progress Report, Nucl. Instr. and Meth. in Phys. Res. A 352 (1994), 6-12
[3] J.R. Levine, T.Mason and D.Brown, lex & yacc, (O'Reilly & Associates, Sebastopol, CA 1992).