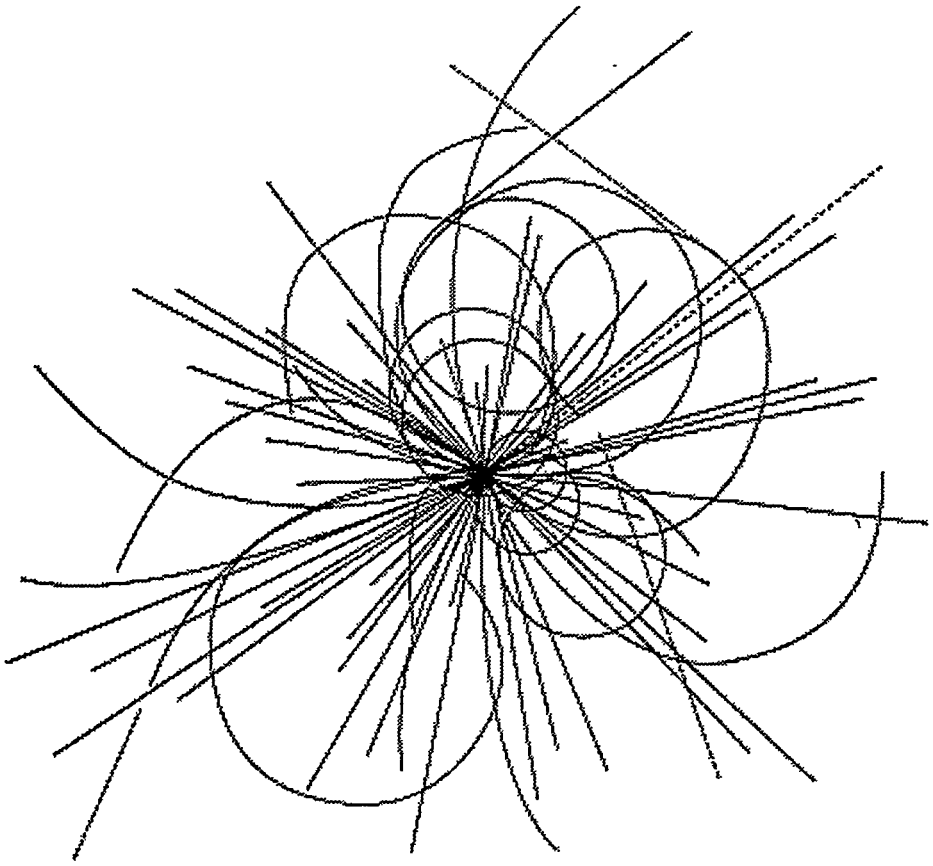
SSCL-675

SSCL-675
June 1994
Distribution Category: 414

N. Malitsky
A. Reshetov
G. Bourianoff

PAC++: Object-Oriented Platform for Accelerator Codes



Superconducting Super Collider Laboratory

SSC Laboratory

PRINT SERVICES
RUN/ASSEMBLY SHEET

THIS FORM MUST BE ATTACHED TO ANY REQUISITION FOR COPYING OR PRINTING SERVICES THAT HAS 10 OR MORE ORIGINALS. TTQ32700TTTTTOC

DATE SUBMITTED: 7/19/94

REQUESTER: N. MALITSKY EXT 3805

PROJECT TITLE: PAC++ OBJECT-ORIENTED PLATFORM FOR ACCELERATOR CODES

DOCUMENT NUMBER: SSCL-675

FRONT	BACK	FRONT	BACK	FRONT	BACK	FRONT	BACK
FRONT COVER	DISCLAIMER						
TITLE	BLANK						
1	2						
3	4						
5	6						
7	8						
9	10						
11	12						
13	BLANK						
15	BLANK						
17	18						
19	BLANK						
21	22						
23	24						
25	BLANK						
27	28						
29	30						
31	32						
33	34						
35	BLANK						
37	BLANK						
39	BLANK						
BACK COVER							

QA CHECKER/OPERATOR

DATE

COMMENTS: MAKE 30 COPIES FOR AUTHOR (TO TAKE WITH HIM TO BNL AND CORNELL UNIVERSITY)

SSCL F 0284
2/93 LTS

SSCL-675

PAC++: Object-Oriented Platform for Accelerator Codes

N. Malitsky, A. Reshetov, and G. Bourianoff

Superconducting Super Collider Laboratory*
2550 Beckleymeade Avenue
Dallas, Texas 75237

June 1994

* Operated by the Universities Research Association, Inc., for the U.S. Department of Energy under Contract No. DE-AC35-89ER40486.

1.0 INTRODUCTION

Software packages in accelerator physics have relatively long life cycles. They had been developed and used for a wide range of accelerators in the past as well as for the current projects. For example, the basic algorithms written in the first accelerator program TRANSPORT[1] are actual for design of most magnet systems. Most of these packages had been implemented on Fortran. But this language is rather inconvenient as a basic language for large integrated projects that possibly could include real-time data acquisition, data base access, graphic user interface modules (GUI), and other features.

Some later accelerator programs had been based on object-oriented tools (primarily, C++ language). These range from systems for advanced theoretical studies[10,12] to control system software.[13] For the new generations of accelerators it would be desirable to have an integrated platform in which all simulation and control tasks will be considered with one point of view.

In this report the basic principles of an object-oriented platform for accelerator research software (PAC++) are suggested and analyzed. Primary objectives of this work are to enable efficient self-explaining realization of the accelerator concepts and to provide an integrated environment for the updating and the developing of the code. C++ language increases portability and clarity of Fortran-based programs and also provides some new features in comparison with the traditional approaches:

- All accelerator formalisms are considered from a common position. Transition from one paradigm to another is performed with one assignment operator **=(Pac& P)**. PAC++ itself could be considered as a shell for some old accelerator algorithms as well as a tool for developing new ones.
- Overloaded arithmetic and assignment operators provide the user with a powerful method of accelerator description that is an integral part of C++ source code. Each operation has an intuitive physical sense. Addition enables one to construct an element from some simple bricks while multiplication is a concatenation of elements and lines.
- PAC++ is implemented as an open hierarchy of classes that enables one to derive new paradigms while preserving compatibility with the old ones. This makes possible to link PAC++ applications with the lattice databases and uses them in the control systems. Users could also create their own libraries including in them specific accelerator lattices and algorithms.
- "Time function" concept enables one to simulate changing accelerator features and to receive a more valid understanding of the real physical processes.

2.0 INPUT LANGUAGE

Traditionally programs for the lattice design in accelerator physics had been written in the some standard input language[3] that was then interpreted by the package. This approach in particular stemmed from the desire to write basic algorithms as simple and portable as possible, and from the inability of Fortran to provide suitable language constructions. The powerful mechanisms of object-oriented programming (such as polymorphism, inheritance *etc.*) now allow one to write accelerator algorithms directly in C++ language without damaging program clarity.

This report considers the object-oriented description of accelerator structure. Basically we suggest overloading arithmetic and assignment operators and describing the lattice element as a member of some linear space $\mathcal{E}$. In Appendixes C and D the simplified implementation of the Low Energy Booster,[19] based on the MAD input language and PAC++ syntax respectively, is presented.

2.1 Linear Space $\mathcal{E}$

In PAC++ lattice elements are considered as instances of C++ class **Element** and declared as :

Element id("Name of the element");

where **id** is any valid C++ identifier and "Name of the element" is some symbolic name of this element. We do not utilize specific keywords (**Rbend, Quadrupole, Kicker** *etc.*) for the type of element but rather describe each element with definite (systematic) parameters as a member of a linear space $\mathcal{E}_s$, which contains a subset of instances of C++ class **Element** (Appendix A). The basis class **ElementParameter** defines objects with a single nonzero MAD parameter (L, ANGLE *etc.*) and forms a basis of this linear space. Then an arbitrary element may be presented as their superposition. For example, the basic magnet for the LEB will be defined as:

hb= 1.940***L**+2 $*$ $PI/96$***ANGLE**;

where **L** and **ANGLE** are instances of class **ElementParameter** and element **hb** is an object whose length is equal to 1.940 m and bend angle is $2 * PI/96$ rad. A total list of **ElementParameters** and their relations with MAD keywords is presented in Appendix B. The module structure of PAC++ allows one to easily modify this set by adding new definitions for old objects (*e.g.*, **M** for **L**, **RAD** for **ANGLE** *etc.*) or include new parameters. This approach enables one to describe an element without splitting it into some auxiliary parts and also forms the lattice in direct accordance with the sequence of real elements. We developed ideas, first suggested for the code TEAPOT,[4] and included "nonstandard"

TEAPOT parameter **IR** that defines internal representation of the element. Certainly, the rules and algorithms of this transformation should be determined in the chosen code (Subsection 4.1).

To represent the root mean square (rms) of random error for specific elements, we define C++ class **Error**. Class **Error** is derived from the class **ElementParameter**, inherits its arithmetic operators, and determines similar linear space $\mathcal{E}_r^*$ (Appendix A). For example:

$$\begin{aligned} \mathbf{Error\ glRmsShift} &= 0.0004 * (\mathbf{DX} + \mathbf{DY}); \\ &\cdots \\ \mathbf{Error\ hbRmsMlt} &= 5.82\text{e}{-}05 * \mathbf{K}[1] + 2.56\text{e}{-}05 * \mathbf{KT}[1] + \\ &= 5.70\text{e}{-}04 * \mathbf{K}[2] + 2.69\text{e}{-}04 * \mathbf{KT}[2]; \end{aligned}$$

PAC++ enhances the standard MAD input language and includes the additional **Error** variable and corresponding arithmetic operators in class **Element**. It allows one to consider instances of this class as members of some extended linear space $\mathcal{E}$ that is a direct sum of linear spaces $\mathcal{E}_s$ and $\mathcal{E}_r$:

$$\begin{aligned} &\cdots \\ \mathbf{hb} &+= \mathbf{hbSysMlt} + \mathbf{hbRmsMlt} + \mathbf{glRmsShift}; \\ &\cdots \end{aligned}$$

Access functions **hb.sysValue**(K[1]) and **hb.rmsValue**(K[1]) return the systematic and rms values of the MAD parameter K1. To print out all information about arbitrary element **hb** the standard C++ I/O operator may be used:

cout $\ll$ **hb**;

One of the additional features of PAC++ is the inclusion of arbitrary time functions in the description of element to simulate the real physical processes (magnetic field ramp, power supply ripple *etc.*):

Element rfcavity = 0.024***VOLT*sin**;

It enables one to design the object-oriented implementation of ESME[7] and develop some special operators of different accelerator codes (*e.g.*, *Adiabatic variations* in DIMAD[5]). We did not use it for the Low Energy Booster (Appendixes C and D) because TEAPOT assumes that rf voltage has to be constant and matches different rf parameters in accordance with its own algorithm.

2.2 Line

In accordance with the MAD terminology, lines in PAC++ are defined as a sequence of elements and other lines that form the accelerator structure. We consider them as instances of C++ class **Line** and their concatenation as multiplication of **Element** and **Line** variables. For example, superperiod of the LEB will be defined as (Appendix C):

```
Line arc, strSection;
Line superPeriod = arc*strSection;
```

where **superPeriod** is a instance of class **Line**. **superPeriod** includes pointers of the factors **arc** and **strSection** and all their modifications that may be made before or after this concatenation. It could be useful for lattice optimization or for inclusion of special sections (injection, extraction *etc.*) without changes in other parts of an accelerator structure:

```
Line strSection01, strSection02;
            ...
// Variant 1
strSection  =  strSection01;
            ...                  // Study of physical parameters
                                 // of superPeriod
// Variant 2
strSection  =  strSection02;
            ...                  // Study of physical parameters
                                 // of superPeriod
```

As a consequence of the multiplicative operations, the repetition of elements and lines is considered as their raising to the power and described by the functions **power (Element& e, int n)** and **power(Line& l, int n)**. To reverse order in the sequences of elements and lines, the function **reflect(Line& l)** is used. This function satisfies the following rule: *If* **e1** *is an instance of class* **Element** *and* **l2** *is an instance of class* **Line**, *then* **reflect** *(***e1****l2) is equal to* **reflect** *(l2)**e1.

The optimal accelerator structures and their sections may be allocated in the different header files and used together to study common physical processes:

```
main()
{
#include "LINAC_LEB.icc"
#include "LEB.icc"
```

```
...
Line injection = linac_leb*leb;
...
}
```

3.0 BASIC DEFINITIONS AND CLASSES

As any physical object, accelerator and its elements are characterized by some set of input and output parameters: the linear transfer matrix and periodic twiss functions,[14] the second order aberration coefficients,[15] one-turn maps,[17] position of closed orbit and particle coordinates referred to the ideal orbit, element coordinates in the global Cartesian system and all element parameters, described in the previous section. They are the important ingredients of all accelerator algorithms and may be considered as a set of C++ classes that form the platform for object-oriented implementation of modern accelerator programs (Appendix F).

3.1 Particle or Position

In PAC++ the position of the particle is determined by the standard canonical variables[3] and considered as C++ class **Particle**.† The subscripting operator [ ]($int\, i$) returns the reference to i-th variable and enables to use the instance of class **Particle** as usual vector:

```
double x, px, y, py, dt, dE;
...
Particle p;
p[1] = x;     // Horizontal position x,[m]
p[2] = px;    // Horizontal canonical momentum, divided
              // by the reference momentum: p_x/p_0,[1]
p[3] = y;     // Vertical position y,[m]
p[4] = py;    // Vertical canonical momentum, divided
              // by the reference momentum: p_y/p_0,[1]
p[5] = dt;    // Velocity of light times the negative time difference with
              // respect to the reference particle: -cΔt,[m]
p[6] = dE;    // Energy difference, divided by the reference momentum
              // times the velocity of light: δ = ΔE/p_0 c,[1]
```

† To describe the beam deviation in monitors or the position of closed orbit we use the alternative name **Position**.

If we will consider the accelerator as some object **A** of class **Accelerator**, the one-turn tracking may be defined as multiplication of this object **A** and phase space vector **x**:

```
y = A * x ;
```

where **x** and **y** are the instances of class **Particle** and contain the particle coordinates before and after one turn, respectively. For multiparticle tracking the arrays of **Particles** could be used:

```
Particle**  x;
            ...
for(int i = 1; i <= numberParticles; i++)
   for(int j = 0; j < numberTurns; j++)
      x[i][j+1] = A*x[i][j];
```

The common beam parameters, such as energy, particle mass and charge, are located in C++ class **Beam**. Class **Particle** is derived from this class and shares all its static data members. The default particle type is proton, and the energy is equal to infinite value. To change them the corresponding variables **ENERGY**, **MASS** and **CHARGE** may be used:

```
ENERGY = 1.538;                    // Beam energy, [GeV]
```

This set of parameters could be developed to include in our model some additional physical effects (*e.g.*, number of particles for simulation of space charge).

3.2 Rmatrix

The position $\vec{x}$ of the particle after some magnet transport system may be expressed by means of Taylor expansion as:

$$x_i = \sum_{j=1,6} R_{ij} * x_{j0} + \sum_{j,k=1,6} T_{ijk} * x_{j0} * x_{k0} + \text{higher order}, \qquad (1)$$

where R_{ij} and T_{ijk} are first and second order Taylor coefficients and vector $\vec{x_0}$ is the vector of initial particle coordinates. The transfer matrix R (or the Courant-Snyder matrix[14]) is the fundamental object of accelerator optics. It allows one to represent the transport system and its elements in simple and convenient form that is described by standard mathematical formalism.

We implement the Courant-Snyder matrix R as C++ class **Rmatrix**. Since its coefficients are determined by the physical parameters of the element or magnet system, data

members of class **Rmatrix** may be initialized by the corresponding objects **Element E1** or **Line L2**:

Rmatrix R1 = E1;
Rmatrix R2 = L2;

From the other side, for the investigation of dominant effects in complicated accelerator systems (*e.g.*, interaction region (IR) in colliders) it is very useful to replace the long regular section by the single linear matrix and determine the object of class **Line** by the suitable **Rmatrix** variable **R**:

Line IR;
Line regularSection = R;
Line Collider = IR*regularSection;

Class **Rmatrix** overloads assignment and multiplicative $(*, /)$ operators. For example, the transfer matrix **R** of the magnet system that consists of two elements **E1** and **E2** may be determined as:

Rmatrix R1 = E1;
Rmatrix R2 = E2;
Rmatrix R = R2*R1;

and a linear transformation of phase space vector $\vec{x_0}$ as:

x = R*x0;

where **x** and **x0** are the instances of class **Particle** and contain the particle coordinates before and after this system, respectively.

The subscripting operator [ ](*int i*) returns the reference to **Particle**, which represents i-th row of corresponding transfer matrix, and enables use of the instances of class **Rmatrix** as a usual matrix.

We did not consider the second-order matrix because of the huge size of this material. The theory of aberration coefficients is exhaustively presented in several papers[15,16] and may be implemented simultaneously with the corresponding accelerator programs.[1,2,5,8]

3.3 Twiss

In linear approximation the horizontal and vertical motion of the particle in a circular machine usually is considered independently and described by the Twiss periodic

functions.[14] For example, the horizontal position of a particle with initial coordinates $\vec{x}(0)$ after n turns at i-th element may be determined as:

$$x(i) = \sqrt{\frac{\beta_x(i)}{\beta_x(0)}}\left[\cos\Big(2\pi\Big(\mu_x(i)+n\nu_x\Big)\Big)+\alpha_x(0)\sin\Big(2\pi\Big(\mu_x(i)+n\nu_x\Big)\Big)\right]*\Big(x(0)-D_x(0)\delta\Big)$$
$$+\sqrt{\beta_x(i)\beta_x(0)}\sin\Big(2\pi\Big(\mu_x(i)+n\nu_x\Big)\Big)\Big(x'(0)-D'_x(0)\delta\Big)+D_x(i)\delta, \qquad (2)$$

where ν_x is the horizontal tune and $\beta_x(i), \alpha_x(i), \mu_x(i), D_x(i), D'_x(i)$ are Twiss parameters at the i-th element.

We consider this pair of Twiss parameters as data members of C++ class **Twiss**. Their value in the injection point may be defined from the periodic condition of the linear motion and then transformed for the i-th element. In PAC++ syntax it may be expressed by assignment and multiplicative operators:[†]

Twiss TWinj = Rtotal;
Twiss TWi = Ri*TWinj;

where **Rtotal** and **Ri** are the instances of class **Rmatrix** and contain the coefficients of the transfer matrixes of one turn and the region between injection point and i-th element. Similar expressions may be written for the calculation of linear approximation of closed orbit:

Position closedOrbitinj = Rtotal;
Position closedOrbiti = Ri*closedOrbitinj;

From the other side, if we will include the momentum compaction $\alpha = \frac{\Delta l/l}{\delta}$ in data members of class **Twiss**, its objects may be used for initialization of corresponding transfer matrixes:

Rmatrix Ri = Twi;

The similar class **Couple** could be considered for coupled betatron motion described in paper.[18]

3.4 Pac

In the various accelerator codes there are some common functions and parameters that we represent as class **Pac**. Then new accelerator programs may be implemented as its derived classes (Appendix F). One modifies only those access and virtual functions of

[†] To determine Twiss parameter μ we included a phase advance in the data members of class **Rmatrix**.

basis class **Pac** that are different. It enables one to reduce code size and concentrate all efforts on design of the specific algorithms.

The instances of this classes could get initial values by using ordinary C ++ assignment operators. Operator **=(Line& L)** defines only one pointer **Line* line**. To identify each element in the accelerator structure we include the additional pointers **Element** delta** and **Element** error** and introduce second assignment operator **=(Pac& P)**. It allows one to use the instances of class **Pac** as some bridge between different accelerator codes:

```
Line    leb;
Pac     lebPac = leb;
...
// Design and Optimization of leb
Mad*    lebMad; lebMad = new Mad(lebPac);
...
delete  lebMad;
// Tracking
Teapot* lebTeapot; lebTeapot = new Teapot(lebPac);
...
```

where classes **Mad** and **Teapot** are the C++ implementations of programs MAD[3] and TEAPOT.[4]

The random errors for some elements **hb, qf1h and qf2h** may be distributed by means of the overloaded function **setError**:

```
Pac lebPac;
...
// To initialize by external rms errors Er1 and Er2
lebPac.setError(seed, rmsEngine, Er1, hb);
lebPac.setError(seed, rmsEngine, Er2, qf1h*qf2h);
...
// To initialize by "own" rms errors
lebPac.setError(seed, rmsEngine, hb);
lebPac.setError(seed, rmsEngine, qf1h*qf2h);
```

where **seed** is **int** argument for the random number generator **rmsEngine**. To use **setError** in the derived accelerator programs we should modify only one protected virtual func-

tion **ownSetError(int n)** that connects own data members with **PAC** parameters of n-th element.

Because of memory problems instances of class **Pac** contain the second-order matrix only for the whole accelerator. It is returned by access function **tmatrix()**.‡ The corresponding parameters for i-th element may be calculated by the similar function **tmatrix(int i)**.

To extract some specific set of elements from the accelerator structure, we introduce additional C++ class **Set**. It is derived from class **Pac** (Appendix E) and initialized by the public function **set**:

```
Pac lebPac = leb;
Set bpmH = lebPac.set(pm);
```

where **pm** is the instance of class **Element**. The function **set** may have different arguments, but it is defined by single virtual functions **ownSet(int* numbers, int n)**, where **numbers** is an array of original element numbers.

Besides the inherited data members, class **Set** includes for each element second-order matrix coefficients, closed orbit and beam position. It allows simplified access to the data and uses these objects in different optimization algorithms.

4.0 SEQUELS

In this report some common principles for object-oriented implementation of accelerator algorithms were described. Our consideration was limited by the direct (element-by-element) particle tracking and extraction of a one-turn Taylor map. As an example we present the initialization and tracking functions of two widespread programs TEAPOT[4] and Zmap.[11]

4.1 Teapot

The program TEAPOT has been developed as a symplectic integrator that produces exact particle tracking in some approximate lattice. In accordance with PAC++ rules this program is implemented as C++ class **Teapot**. It is derived from class **Pac** and inherits all its public and protected data members and functions. Program TEAPOT introduces its own internal representation of lattice elements as sequences of drifts and thin multipoles. These parameters may be considered as data members of auxiliary class **TeapotElement**, which instances are included in class **Teapot**.§ To initialize them, the corresponding **Pac** virtual functions and assignment operators were modified. For

‡ Now we use only the linear part of this object (Section 3.2).

§ Different accelerator programs (such as MAD, DIMAD) could use instances of class **Line** to make similar element representation.

example, the operator =(Line& l) includes additional TEAPOT function **survey**, which determines the coordinates of multipole planes:

```
Teapot::operator=(Line& l)
{
    Pac::initialize(l);
    survey (l);
}
```

To optimize the tracking procedure, TEAPOT considers the sequence of several drift regions as one total section. It breaks the original order of elements and complicates the use of element parameters between different accelerator codes. The propagation of the particle through a drift region may be described by a few mathematical operators and will take an insignificant amount of CPU time. To check it we saved the original accelerator structure in the C++ version and used the example of LEB lattice presented in Appendixes D and E.

4.2 Zmap

A Taylor map $\mathcal{U}(\vec{z})$ in accelerator physics may be considered as the m-dimensional vector of power series $\vec{U}(\vec{z})$ and expressed as:[9]

$$\vec{U}(\vec{z}) = \sum_{k=0}^{\Omega} \vec{u}(\vec{k})\vec{z}^{\vec{k}}, \tag{3}$$

where

$$\vec{z}^{\vec{k}} \equiv z_1^{k_1} z_2^{k_2} \ldots z_n^{k_n},$$

$$k = \sum_{i=1}^{n} k_i, \text{for } 0 \leq k_i \leq \Omega.$$

$$\vec{U}(\vec{z})^T = [U_1(\vec{z}), U_2(\vec{z}), \ldots, U_m(\vec{z})].$$

There are several accelerator packages[6,10] that perform differential algebra through the operations of expanded power series and allow one to obtain one-turn Taylor maps. In this report the object-oriented implementation of program Zmap[11] is considered. Zmap translates the Teapot tracking algorithm into TPS operations through the calling statements of the corresponding Fortran subroutines of differential algebra library ZLIB.[9] In the new object-oriented version ZLIB++,[12] TPS and Taylor maps are implemented as C++ classes **ZSeries** and **ZMap**. They may be naturally linked with PAC++ classes and represent program Zmap as usual assignment operator =(**Teapot**& T):

Teapot lebTeapot = leb;
Zmap lebZmap = lebTeapot;

where class **Zmap** is derived from class **ZMap**. This operator determines one-turn transformation of some pseudo-particle whose coordinates are instances of class **ZSeries**. ZLIB++ overloaded additive (+ and −) and multiplicative (∗ and /) operators for these objects and introduced special rules for their usage. It allows one to simplify the Zmap program and directly copy the corresponding Teapot mathematical expressions.[4] As an example, in Appendix F we present **Zmap** function **mltKick** that determines the transformation of **ZSeries** variables in a thin multipole.

5.0 APPLICATION ISOLATION CODE

In order to provide easy read/write operations for the PAC++ constructions, a specific interface module between PAC++ and the Application Isolation Code (AIC) library has been designed. It would be especially fruitful for development of the integrated control systems that have to include distributed database access combined with simulation facilities.[13]

A typical module in the Simulation Facility consists of the three processes running simultaneously:

1. display program to support interaction with an operator;
2. application itself which realizes specific algorithms in accordance with operator requests;
3. simulator module to emulate a real accelerator.

These processes are interactive only through the external database and so when the operator activates some command, the display program places the request into the database and then the application reads the command and activates the simulator, also through the database request. This scheme was chosen to insure the maximum portability because for some database management systems (*e.g.*, EPICS) display programs exist only as independent tasks and cannot be directly incorporated into the source code of the application. Besides, this approach enables one to change realization of one process while not affecting two others.

The application uses C++ Application Isolation Code (AIC) library[13] to insure portable access to the database. AIC is realized as a set of C++ base classes that provide basic control data types and some operations for this data. Details of the database call interface are hidden from the High Level Application Code (HLAC).

AIC could utilize different underling database management systems. In particular, it could simply use a share memory model. Changing of the underling model would require only relinking of the application with another library. The AIC is to provide data concentration and isolation functions. This allows accelerator physicists to concentrate on the physics and avoid the complications associated with data attributes, error conditions *etc.*

AIC is intended to provide the following features:

- a simple representation of control data in a structured manner;
- independence of high level application from database and hardware related features;
- portability of control code with respect to different operating systems as well as different database packages;
- uniform approach to data retrieved from various sources (EPICS database, text file(s), share memory).

To utilize access to the external database, PAC++ application must declare instance of the access handler:

AIC database;

This will enable one to read/write PAC++ constructions to the database by C++ I/O operators:

database $\ll$ **bpm;**

... *//Some calculations*

database $\gg$ **corr;**

where **bpm** and **corr** are instances of the class **Set** (Section 3.4).

All specific features of the used database are hidden from the HLAC code and located in the AIC library that is linked to produce the executable file.

ACKNOWLEDGEMENTS

We would like to thank Prof. Richard Talman and Dr. Yiton Yan for providing us with the codes TEAPOT and Zmap.

REFERENCES

1. Karl L. Brown., D.C. Carey, Ch. Iselin, and F. Rothacker, "TRANSPORT - A Computer Program for Designing Charged Particle Beam Transport Systems," CERN 73-16, revised as CERN 80-4, CERN, 1980.
2. Yu.P. Severgin, Preprint G-0270, NIIEFA, Leningrad, 1976.
3. H. Grote and F.C. Iselin, "The MAD Program (Methodical Accelerator Design) Version 8.1, User's Reference Manual," CERN/SL/90-13 (AP).
4. L. Schachinger and R. Talman, "Teapot: A Thin-Element Accelerator Program for Optics and Tracking," *Particle Accelerators*, **22**, 35(1987).
5. R.V. Servranckx, *et al.*, "User's Guide to the Program DIMAD," SLAC Report 285 UC-28, May 1985.
6. M. Berz, H.C. Hofmann, and H. Wollnik, "COZY 5.0, the fifth order code for corpuscular optical systems," *Nuclear Instruments and Methods*, A258:402, 1987.
7. S. Stahl and J. MacLachlan, "User's Guide to ESME v.7.1," Fermilab internal note TM-1650 (Dec. 90).
8. M.G. Nagaenko, "A program package for beam dynamics study," *In Proc. of the 2nd European particle accelerator conference*, Nice, 1990.
9. Y. Yan and Chiung-Ying Yan, "ZLIB - A Numerical Library for Differential Algebra," SSC Laboratory Report SSCL-300, 1990.
10. L. Michelotti, "MXYZPPLK: a C++ version of differential algebra," Fermi National Accelerator Laboratory Report FN-535, 1990.
11. Y. Yan, "Zmap - A Differential Algebraic High-Order Map Extraction Program for Teapot Using ZLIB," SSC Laboratory Report SSCL-299, 1990.
12. N. Malitsky, A. Reshetov and Y. Yan, "ZLIB++: Object-Oriented Numerical Library for Differential Algebra," SSC Laboratory Report SSCL-659, 1994.
13. G. Bourianoff, A. Reshetov and N. Malitsky, "Object-Oriented Approach for the Design of the Simulation Facility of the SSC," SSC Laboratory Report SSCL-677, 1994.
14. E.D. Courant and H.S. Snyder, "Theory of the Alternating Gradient Synchrotron," *Ann.Phys.* 3, 1–48, 1958.
15. Karl L. Brown. "A First-and Second-Order Matrix Theory for the Design of Beam Transport System and Charged Particle Spectrometers," SLAC 75, Revision 3, SLAC, 1972.

16. M.G. Nagaenko, Yu.P.Severgin and I.A.Shukeilo, "Study of nonlinear beam circular motion in accelerators based on aberration theory," *In Proc. of the XIIth International Conference on High-Energy Accelerators*, Batavia, 1983.

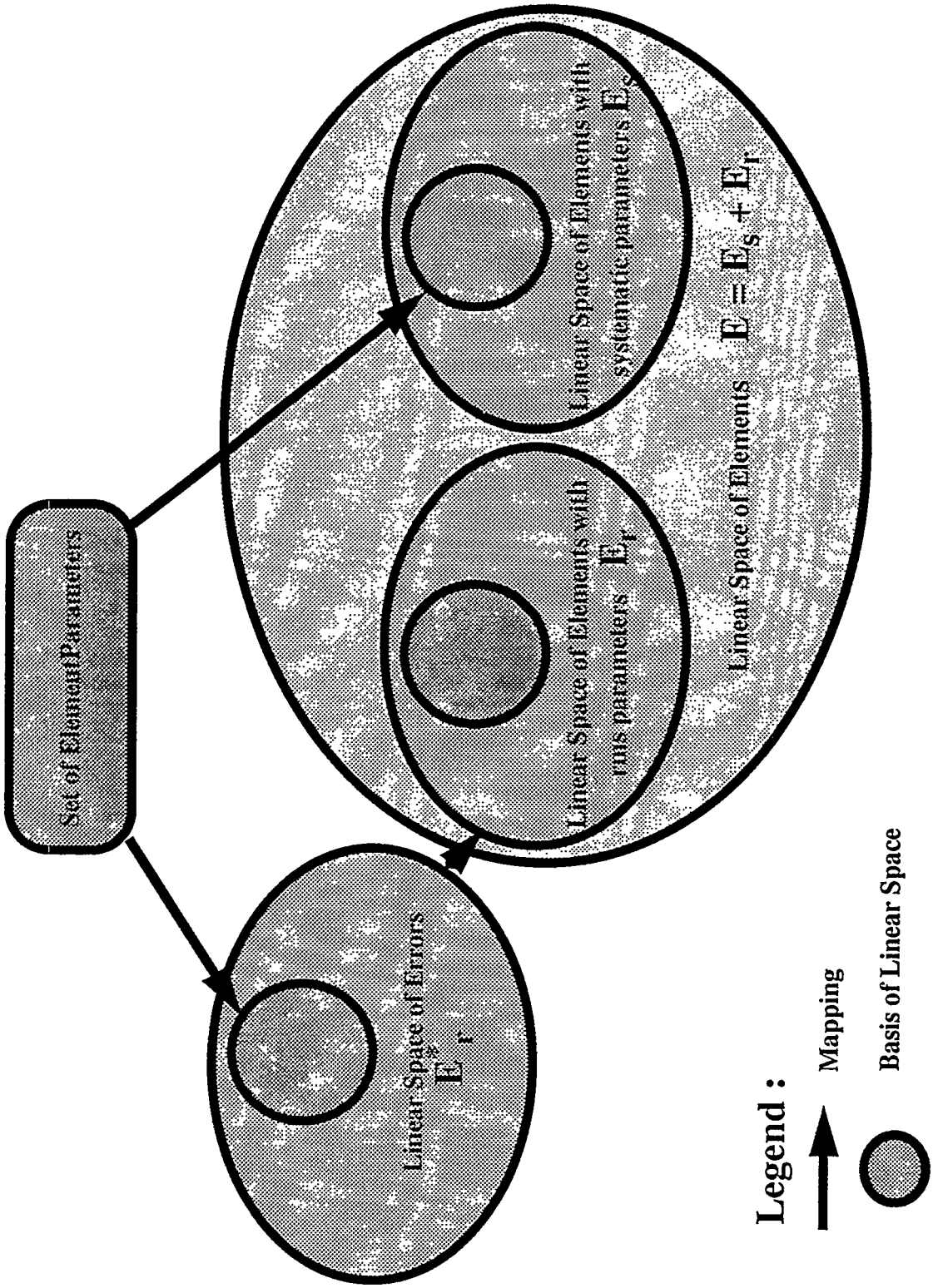
17. A.J. Dragt, "Lectures on nonlinear orbit dynamics," *1981 Femilab Summer School.* AIP Conference Proceedings Vol. 87, 1982.

18. R. Talman, "A Universal Algorithm for Accelerator Correction," AIP Conference Proceedings Vol. 255, 1991.

19. U. Wienands, *et al.*, "The H-γ_t Lattice of the SSC Low Energy Booster," *Conference Record of the XVth International Conference on High Energy Accelerators*, Hamburg, 1992.

APPENDIX A

Linear Space of Elements



APPENDIX B

List of the MAD Keywords and Element Parameter Variables

MAD			PAC++		
Parameter keyword	Quantity	Unit	Object of ElementParameter	Quantity	Value
L	length	m	L	length	1 m
			M	-"-	1 m
–	–	–	IR	splitting	1 split
ANGLE	bend angle	rad	ANGLE	bend angle	1 rad
			RAD	-"-	1 rad
E1	entrance edge angle	rad	E1	entrance edge angle	1 rad
E2	exit edge angle	rad	E2	exit edge angle	1 rad
FINT	fringe field integral	–	–	–	
–	–	–	FINT1	entrance f.integral	1
–	–	–	FINT2	exit f.integral	1
H1	entrance pole face curvature	m^{-1}	H1	entrance pole face curvature	1 m^{-1}
H2	exit pole face curvature	m^{-1}	H1	exit pole face curvature	1 m^{-1}

MAD			PAC++		
Parameter keyword	Quantity	Unit	Object of **ElementParameter**	Quantity	Value
HKICK	horizontal kick angle	rad	HKICK	horizontal kick angle	1 rad
VKICK	vertical kick angle	rad	VKICK	vertical kick angle	1 rad
KICK	kick angle in both planes	rad	–	–	
Kn	strength of normal multipole $K_n = \frac{1}{B\rho}\frac{\partial^n B_y}{\partial x^n}$	$m^{-(n+1)}$	K[n]	strength of normal multipole $k(n) = \frac{1}{n!B\rho}\frac{\partial^n B_y}{\partial x^n}$	1 $m^{-(n+1)}$
Kn,TILT	strength of skewed multipole $K_n^{(tilt)} = \frac{1}{B\rho}\frac{\partial^n B_x}{\partial x^n}$	$m^{-(n+1)}$	KT[n]	strength of skewed multipole $kt(n) = \frac{1}{n!B\rho}\frac{\partial^n B_x}{\partial x^n}$	1 $m^{-(n+1)}$
KnL	integrated strength of normal thin multipole	m^{-n}	KL[n]	integrated strength of normal multipole	1 m^{-n}
KnL,Tn	integrated strength of skewed thin multipole	m^{-n}	KLT[n]	integrated strength of skewed multipole	1 m^{-n}

MAD			PAC++		
Parameter keyword	Quantity	Unit	Object of **ElementParameter**	Quantity	Value
			K[n]	integrated strength of normal thin multipole	1 m^{-n}
			KT[n]	integrated strength of skewed thin multipole	1 m^{-n}
Tn	tilt angle for n-order multipole components	rad	T[n]	tilt angle for n-order multipole components	1 rad
KS	solenoid strength $KS = \frac{B_0}{B\rho}$	rad/m	KS	solenoid strength $ks = \frac{B_0}{B\rho}$	1 rad/m
E	electric field strength	$\frac{MV}{m}$	E	electric field strength	1 $\frac{MV}{m}$
VOLT	peak rf voltage	MV	VOLT[n]	peak rf voltage	1 MV
			MV[n]	-"-	1 MV
LAG	phase lag of rf cavity (* 2π)	–	LAG[n]	phase lag of rf cavity (* 2π)	1

MAD			PAC++		
Parameter keyword	Quantity	Unit	Object of **ElementParameter**	Quantity	Value
HARMON	harmonic number of rf cavity	–	HARMON[n]	harmonic number of rf cavity	1
			FREQ[n]	-"-	1
XSIZE	horizontal half-aperture	m	XSIZE	horizontal half-size of rectangle	1 m
YSIZE	vertical half-aperture	m	YSIZE	vertical half-size of rectangle	1 m
–	–	–	XAXIS	horizontal half-axis of ellipse	1 m
–	–	–	YAXIS	vertical half-axis of ellipse	1 m
DX	misalignment in the x-direction	m	DX	shift in the x-direction	1 m
DY	misalignment in the y-direction	m	DY	shift in the y-direction	1 m
DS	misalignment in the s-direction	m	DS	shift in the s-direction	1 m

MAD			PAC++		
Parameter keyword	Quantity	Unit	Object of **ElementParameter**	Quantity	Value
DPHI	rotation around the x-axis	rad	DPHI	rotation around the x-axis	1 rad
DTHETA	rotation around the y-axis	rad	DTHETA	rotation around the y-axis	1 rad
DPSI	rotation around the s-axis	rad	DPSI	rotation around the s-axis	1 rad
TILT	-"-	rad	TILT	-"-	1 rad

APPENDIX C
LEB Lattice. MAD Version

```
title
Teapot::New triangular LEB lattice updated on OCT. 28, 1992

!*****************************************************************************
! Elements
!*****************************************************************************

!************************************
! Dipole Magnet
!************************************

para, twopi= 6.2831853071796
para, nbend=96
para, abend=twopi/nbend
para, lbend=1.940
para, radb=lbend/abend
para, kb=sin(abend/2.)/cos(abend/2.)/radb

hb    :   sbend,     l=lbend, ANGLE=abend
mb    :   multipole, k1l=-kb

db1   :   drift, l=0.1115
db2   :   drift, l=0.1485

hbL   :   line = (db2, mb, hb, mb, db1, db1, mb, hb, mb, db2)

!************************************
! Main Quadrupoles
!************************************

kq=0.3722

qf1h  :  quadrupole, l=0.7564/2, k1=+kq, type=ir
qf2h  :  quadrupole, l=0.5919/2, k1=+kq, type=ir
qd1h  :  quadrupole, l=0.5983/2, k1=-kq, type=ir
qd2h  :  quadrupole, l=0.6127/2, k1=-kq, type=ir
qfs1h :  quadrupole, l=0.6568/2, k1=+kq, type=ir
qfs2h :  quadrupole, l=0.5552/2, k1=+kq, type=ir
qds1h :  quadrupole, l=0.6980/2, k1=-kq, type=ir
qds2h :  quadrupole, l=0.6858/2, k1=-kq, type=ir

!************************************
! Norm Trim Quads
!************************************

qtf    : quadrupole, l=0.294, k1=+0.589E-02
qtd1   : quadrupole, l=0.294, k1=-0.116E-01
qtd2   : quadrupole, l=0.294, k1=-0.157E-01
qtfs1  : quadrupole, l=0.294, k1=+0.425E-01
qtfs2  : quadrupole, l=0.294, k1=+0.573E-01
qtds1  : quadrupole, l=0.294, k1=-0.663E-02
```

```
qtds2  : quadrupole, l=0.294, k1=-0.253E-01
         .
dqt    : drift, l=0.048

qtfL   : line = (dqt, qtf,   dqt)
qtd1L  : line = (dqt, qtd1,  dqt)
qtd2L  : line = (dqt, qtd2,  dqt)
qtfs1L : line = (dqt, qtfs1, dqt)
qtfs2L : line = (dqt, qtfs2, dqt)
qtds1L : line = (dqt, qtds1, dqt)
qtds2L : line = (dqt, qtds2, dqt)

!***********************************
! Norm Sextupoles
!***********************************

sextf     : sextupole, l=0.3, k2=+1.07E+00
sextd1    : sextupole, l=0.3, k2=-1.94E+00
sextd2    : sextupole, l=0.3, k2=-2.08E+00

dsext     : drift,     l=0.05

sextfL    : line = (dsext, sextf,  dsext)
sextd1L   : line = (dsext, sextd1, dsext)
sextd2L   : line = (dsext, sextd2, dsext)

!***********************************
! Position Monitors
!***********************************

bpm    : monitor, l=0.15

dbpm   : drift,   l=0.02

bpmL   : line = (dbpm, bpm, dbpm)

!***********************************
! Correctors
!***********************************

ch  :  hkick,      l=0.15
cv  :  vkick,      l=0.15

dc  :  drift,      l=0.06

chL : line = (dc, ch, dc)
cvL : line = (dc, cv, dc)

!***********************************
! Drifts
!***********************************

dq1     : drift, l=0.1
```

```
darc0   : drift, l=0.4915
darc1   : drift, l=0.4974
darc2   : drift, l=0.2091
darc3   : drift, l=0.3591
darc4   : drift, l=0.2134
darc5   : drift, l=2.6814
darc6   : drift, l=0.1900
darc7   : drift, l=2.9514
darc8   : drift, l=0.2274
darc9   : drift, l=0.1900
darc10  : drift, l=0.4915

dstr0   : drift, l = 6.4231
dstr1   : drift, l = 2.7976
dstr2   : drift, l = 7.4234
dstr3   : drift, l = 0.09
dstr4   : drift, l = 7.1534

!*************************************************************************
! Lines
!*************************************************************************

!***********************************
! Arc
!***********************************

fstDBFBD: line = &
  (qds2h,(bpmL,qtds2L,darc0),hbL,(darc2,qtfL,bpmL) ,qf2h, &
   qf2h, (dq1,chL,darc3)     ,hbL,(darc4,qtd1L,bpmL),qd1h)

DBFBD1  : line = &
   (qd2h,(bpmL,qtd2L,darc1),hbL,(darc2,qtfL,bpmL) ,qf2h, &
    qf2h,(dq1,chL,darc3)    ,hbL,(darc4,qtd1L,bpmL),qd1h)

DOFOD   : line = &
   (qd1h,(dq1,sextd1L,cvL,darc5,chL,sextfL ,bpmL), qf1h, &
    qf1h,(darc6,sextfL,darc7,cvL,sextd2L,dq1)     , qd1h)

DBFBD2  : line = &
   (qd1h,(bpmL,qtd1L,darc4),hbL,(darc3,chL,dq1)        ,qf2h, &
    qf2h,(bpmL,qtfL,darc2) ,hbL,(darc8,cvL,qtd2L,darc9),qd2h)

lstDBFBD: line = &
   (qd1h,(bpmL,qtd1L,darc4),hbL,(darc3,chL,dq1)      ,qf2h, &
    qf2h,(bpmL,qtfL,darc2) ,hbL,(darc10,qtds2L,bpmL),qds2h)



arcf      : line = (fstDBFBD,DOFOD, DBFBD2)
arcs      : line = (DBFBD1  ,DOFOD, DBFBD2)
arcl      : line = (DBFBD1  ,DOFOD, lstDBFBD)
arc       : line = (arcf, 2*arcs, arcl)
```

```
!***********************************
! Straight Section
!***********************************

DOFOD1  : line = &
   (qds2h,(dq1,cvL,dstr0,qtfs1L,bpmL),qfs1h, &
    qfs1h,(dq1,chL,dstr1,qtds1L,bpmL),qds1h)

DOFOD2  : line = &
   (qds1h ,(dq1,cvL,dstr2,qtfs2L,bpmL)          ,qfs2h, &
    qfs2h ,(dq1,dstr3,qtfs2L,chL,dstr4,cvL,dq1),qds1h)

str     : line = (DOFOD1, DOFOD2, -(DOFOD1))

period  : line = (arc, str)

!*************************************************************************
! LEB
!*************************************************************************

leb     : line=(3*period)

!*************************************************************************
```

APPENDIX D

LEB Lattice. PAC++ Version

```
// File               : LEB.icc
// Description        : This file contains the structure of LEB
// Created            : May 15, 1994
// Authors            : George Bourianoff  (gib@mesquite.ssc.gov)
//                      Nikolay Malitsky   (malitsky@ivory.ssc.gov)
//                      Alexander Reshetov (reshetov@vernon.ssc.gov)
//
//
// (C) Copyright
// SSC Laboratory
// 2550 Beckleymeade Ave.
// Dallas, TX, 75237

// Global Parameter

   ENERGY = 1.538;

   Error glRmsShift = 0.0004*(DX + DY);

// Main Dipole Magnet

   Element hb("Main Dipole");

   int    hbNumber = 96;
   double hbAngle  = 2*PI/hbNumber;

   Element hbSysMlt  = -4.41e-04*K[2] +
                        2.32e+00*K[4] -
                        4.09e+03*K[6] +
                        5.90e+06*K[8];

   Error   hbRmsMlt  =  5.82e-05*K[1] + 2.56e-05*KT[1] +
                        5.70e-04*K[2] + 2.69e-04*KT[2] +
                        8.96e-03*K[3] + 5.06e-03*KT[3] +
                        7.04e-02*K[4] + 5.63e-02*KT[4] +
                        3.78e+00*K[5] + 1.34e+00*KT[5] +
                        3.58e+01*K[6] + 2.05e+01*KT[6];

   hb  = 1.940*M + hbAngle*(RAD + E1/2. + E2/2.);
   hb += hbSysMlt + hbRmsMlt + glRmsShift;

   Element db1("DB1"); db1 = 0.1115*M;
   Element db2("DB2"); db2 = 0.1485*M;

   Line hbL = db2*hb*db1*db1*hb*db2;

// Main Quadrupoles

   double qK =  0.3722;
```

```
   Error   qRmsMlt  =   7.28e-05*K[1] + 1.00e-18*KT[1] +
                        3.36e-04*K[2] + 1.82e-04*KT[2] +
                        1.09e-02*K[3] + 7.28e-03*KT[3] +
                        2.80e-02*K[4] + 3.64e-02*KT[4] +
                        7.28e-01*K[5] + 1.09e+00*KT[5];

   Element qtmp  =  qRmsMlt + glRmsShift + 3*IR;
   Element qftmp =  qK*K[1] + qtmp;
   Element qdtmp = -qK*K[1] + qtmp;

   Element qf1h("QF1H");   qf1h  = 0.7564/2*M + qftmp;
   Element qf2h("QF2H");   qf2h  = 0.5919/2*M + qftmp;
   Element qd1h("QD1H");   qd1h  = 0.5983/2*M + qdtmp;
   Element qd2h("QD2H");   qd2h  = 0.6127/2*M + qdtmp;

   Element qfs1h("QFS1H"); qfs1h = 0.6568/2*M + qftmp;
   Element qfs2h("QFS2H"); qfs2h = 0.5552/2*M + qftmp;
   Element qds1h("QDS1H"); qds1h = 0.6980/2*M + qdtmp;
   Element qds2h("QDS2H"); qds2h = 0.6858/2*M + qdtmp;


// Norm Trim Quads

   Element qt[7] = {"QTF", "QTD1", "QTD2", "QTFS1", "QTFS2", "QTDS1", "QTDS2"};
   Line    qtL[7];

   static double qtLength = 0.294;

   qt[0] =  0.589e-02*K[1];
   qt[1] = -0.116e-01*K[1];
   qt[2] = -0.157e-01*K[1];
   qt[3] =  0.425e-01*K[1];
   qt[4] =  0.573e-01*K[1];
   qt[5] = -0.663e-02*K[1];
   qt[6] = -0.253e-01*K[1];

   Element dqt("DQT"); dqt = 0.048*M;

   for(int iqt=0; iqt < 7; iqt++)
   {
      qt[iqt]  += qtLength*M;
      qtL[iqt]  = dqt*qt[iqt]*dqt;
   }

// Norm Sextupoles

   Element sext[3] = {"SEXTF", "SEXTD1", "SEXTD2"};
   Line    sextL[3];

   static double sextLength = 0.3;

   sext[0] =  0.535*K[2];
   sext[1] = -0.970*K[2];
   sext[2] = -1.040*K[2];
```

```
   Element dsext("DSEXT"); dsext = 0.05*M;

   for(int isext=0; isext < 3; isext++)
   {
     sext[isext]  += sextLength*M;
     sextL[isext]  = dsext*sext[isext]*dsext;
   }

// Beam Position Monitors

   Element bpm("BPM");   bpm   = 0.15*M;
   Element dbpm("DBPM"); dbpm  = 0.02*M;
   Line    bpmL;         bpmL  = dbpm*bpm*dbpm;

// Correctors

   Element ch("CorH");  ch  = 0.15*M;
   Element cv("CorV");  cv  = 0.15*M;
   Element dc("DC");    dc  = 0.06*M;

   Line    chL = dc*ch*dc;
   Line    cvL = dc*cv*dc;

// Drifts

   Element dq1("DQ1"); dq1 = 0.1*M;

   Element darc[11] = { 0.4915*M, 0.4974*M, 0.2091*M, 0.3591*M, 0.2134*M,
                        2.6814*M, 0.1900*M, 2.9514*M, 0.2274*M, 0.1900*M,
                        0.4915*M};

   Element dstr[5]  = { 6.4231*M, 2.7976*M, 7.4234*M, 0.0900*M, 7.1534*M};

// Lines

// Arc

   Line fstDBFBD = qds2h*(bpmL*qtL[6]*darc[0])*hbL*(darc[2]*qtL[0]*bpmL)*qf2h*
                   qf2h *(dq1*chL*darc[3])     *hbL*(darc[4]*qtL[1]*bpmL)*qd1h;

   Line DBFBD1   =  qd2h*(bpmL*qtL[2]*darc[1])*hbL*(darc[2]*qtL[0]*bpmL)*qf2h*
                    qf2h*(dq1*chL*darc[3])    *hbL*(darc[4]*qtL[1]*bpmL)*qd1h;

   Line DOFOD    =  qd1h*(dq1*sextL[1]*cvL*darc[5]*chL*sextL[0]*bpmL)*qf1h*
                    qf1h*(darc[6]*sextL[0]*darc[7]*cvL*sextL[2]*dq1) *qd1h;

   Line DBFBD2   =  qd1h*(bpmL*qtL[1]*darc[4])*hbL*(darc[3]*chL*dq1) *qf2h*
          qf2h*(bpmL*qtL[0]*darc[2])*hbL*(darc[8]*cvL*qtL[2]*darc[9])*qd2h;

   Line lstDBFBD =  qd1h*(bpmL*qtL[1]*darc[4])*hbL*(darc[3]*chL*dq1)     *qf2h*
                   qf2h*(bpmL*qtL[0]*darc[2])*hbL*(darc[10]*qtL[6]*bpmL)*qds2h;

   Line arcf     = fstDBFBD*DOFOD*DBFBD2;
```

```
   Line arcs      = DBFBD1*DOFOD*DBFBD2;
   Line arcl      = DBFBD1*DOFOD*lstDBFBD;

   Line arc       = arcf*arcs*arcs*arcl;

// Straight Section

   Line DOFOD1       =  qds2h*(dq1*cvL*dstr[0]*qtL[3]*bpmL)*qfs1h*
                        qfs1h*(dq1*chL*dstr[1]*qtL[5]*bpmL)*qds1h;

   Line DOFOD2       =  qds1h*(dq1*cvL*dstr[2]*qtL[4]*bpmL)  .        *qfs2h*
                        qfs2h*(dq1*dstr[3]*qtL[4]*chL*dstr[4]*cvL*dq1)*qds1h;

   Line strSection   = DOFOD1*DOFOD2*reflect(DOFOD1);

// SuperPeriod

   Line superPeriod = arc*strSection;

// LEB

   Line leb("C++ version");
   leb = power(superPeriod, 3);
```

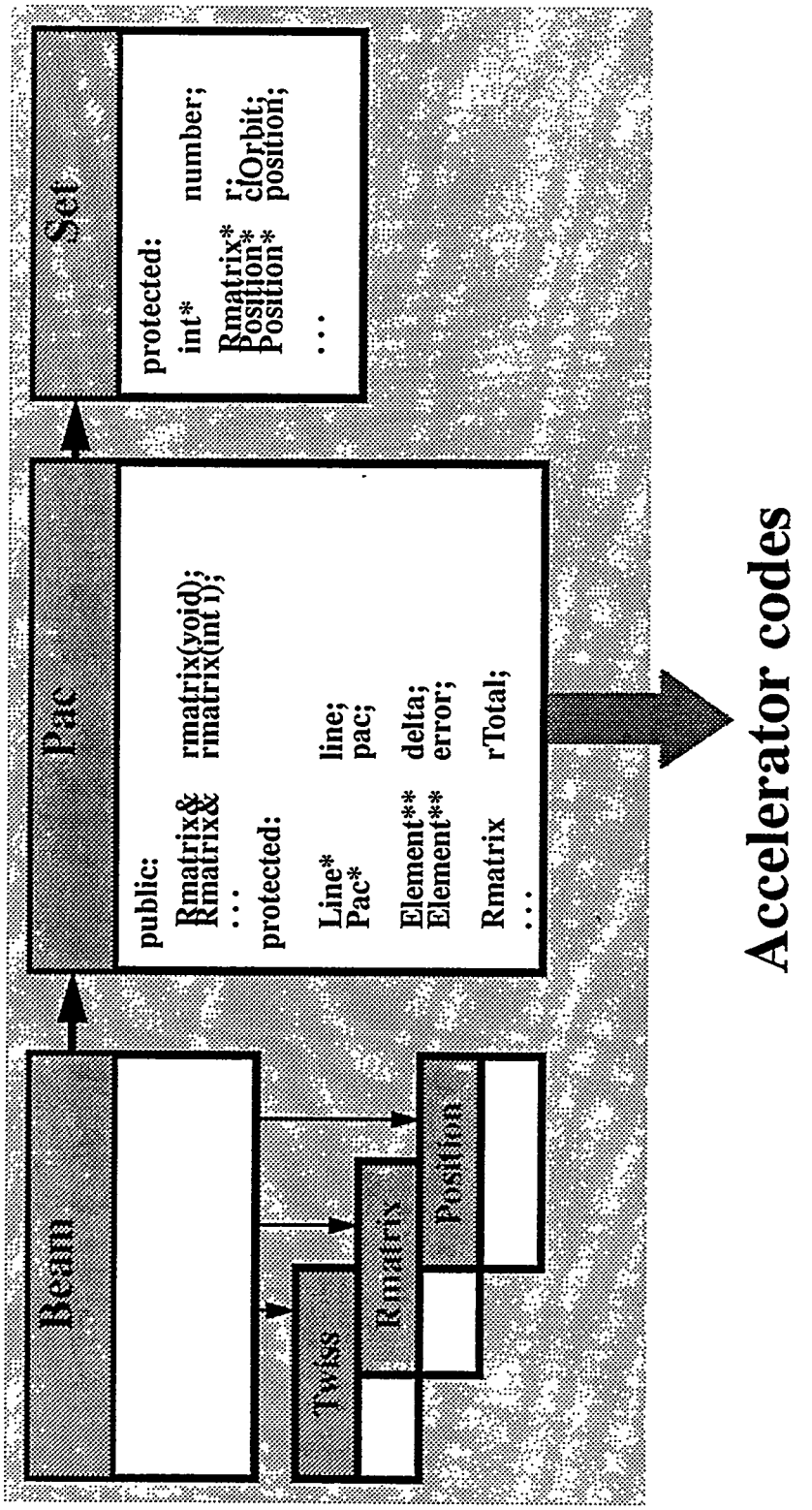
APPENDIX E

LEB Lattice. Example of PAC++ Programming

```
// File                : LEB.cc
// Description         : This file contains the example of the direct
//                       tracking and one-turn map extraction for LEB.
// Created             : May 15, 1994
// Authors             : George Bourianoff  (gib@mesquite.ssc.gov)
//                       Nikolay Malitsky   (malitsky@ivory.ssc.gov)
//                       Alexander Reshetov (reshetov@vernon.ssc.gov)
//
// (C) Copyright
// SSC Laboratory
// 2550 Beckleymeade Ave.
// Dallas, TX, 75237

#include <time.h>
#include "Teapot/Teapot.hh"
#include "Zmap/Zmap.hh"

double staticGenerator(int& s) { return(1.);}

main()
{

#include "LEB.icc"

// Assignment

   Teapot lebTeapot("LEB: Teapot implementation");
   lebTeapot = leb;

// Set Error

   int seed = 1;
   lebTeapot.setError(seed, staticGenerator, hb);
   lebTeapot.setError(seed, staticGenerator, qf1h*qf2h*qd1h*qd2h);
   lebTeapot.setError(seed, staticGenerator, qfs1h*qfs2h*qds1h*qds2h);

// Tracking

    Particle p;
    for(int i=1; i <= 6; i++) p[i] = 1.0e-3;

    for(i=1; i <= 100; i++)  p = lebTeapot*p;

// Extract one-turn Taylor Map

    ZLIB_ORDER = 5;
    Zmap lebZmap = lebTeapot;
    cout << lebZmap;

}
```

APPENDIX F

Chart of PAC++ Classes



APPENDIX G

Zmap. Function mltKick

```
// File               : Zmap.cc
// Description        : This file contains the object-oriented
//                      implementation of program Zmap.
// Created            : May 1, 1994
// Authors            : Nikolay Malitsky (malitsky@ivory.ssc.gov)
//                      Alexander Reshetov (reshetov@vernon.ssc.gov)
//
//
// (C) Copyright
// SSC Laboratory
// 2550 Beckleymeade Ave.
// Dallas, TX, 75237

...

void Zmap::mltKick(TeapotElement& teapot, ZSeries* P, ZSeries* tmp)
{
  tmp[1] = P[1] - teapot.delx;                    //  xdif
  tmp[3] = P[3] - teapot.dely;                    //  ydif

  tmp[2] = 0.0;                                   //  bytw
  tmp[4] = 0.0;                                   //  bxtw
  for(int i=teapot.mltOrder; i >= 0 ; i--)
  {
    tmp[0]  = tmp[1]*tmp[2];
    tmp[0] -= tmp[3]*tmp[4] - teapot.btw[i];
    tmp[4]  = tmp[1]*tmp[4];
    tmp[4] += tmp[3]*tmp[2] + teapot.atw[i];
    tmp[2]  = tmp[0];
  }
  tmp[2] += tmp[1]*teapot.btw01;
  tmp[4] += tmp[3]*teapot.atw01;

  P[2] -= tmp[2];                                 // px/p0
  P[4] += tmp[4];                                 // py/p0

  tmp[0]  = 1.;
  if(Zmap_DIM >= 6)  tmp[0] = tmp[6]*tmp[6];

  tmp[0] -= P[2]*P[2];
  tmp[0] -= P[4]*P[4];
  tmp[0]  = sqrt(tmp[0]);                         // ps/p0

  tmp[0]  = 1./tmp[0];
  tmp[2]  = P[2]*tmp[0];                          // vx/vs
  tmp[4]  = P[4]*tmp[0];                          // vy/vs

  return;
}
```