A Proposed Flat Yet Hierarchical Accelerator Lattice Object Model

N. Malitsky and R. Talman, Laboratory of Nuclear
Studies Cornell University Ithaca NY, 14853

and

F. Dell, S. Peggs, F. Pilat, T. Satogata, S. Tepikian, D. Trbojevic, G. Trahern, and
J. Wei, RHIC Project Brookhaven National Laboratory Upton, Long Island, NY

and

L. Schachinger, Berkeley, CA, 94708

ABSTRACT

A computer generated, standard machine file (*SMF*) is proposed. Its purpose is to facilitate communication of "flat" lattice descriptions between different users and processes, especially for model-based control. "Essential" parameters, possibly time dependent, of the physical elements in the lattice are recorded. Though fully-instantiated, a hierarchical lattice view is retained. "Objects" in the model are closely associated with physical elements, not computer constructs. Though object oriented, class member functions are concerned with data storage and retrieval, not with beam or particle evolution or with theoretical lattice functions. This assures the segregation of data needed for *all* control and modeling programs, from algorithms specific to *particular* programs. For the same reason, logical data specific to particular programs is segregated from physical data. Different representations of the same model are optimized for different purposes: efficiency, rapid interchange, editability, commercial database access, *etc.*. Lattice sectors shared by two rings can be described.

1. Motivation and Informal Considerations

For communicating between different accelerator modeling or control processes it is essential to have a lattice model[†] in which every physical element in the accelerator has its own independent identity, name, and parameters. This can be called a "flat" or "fully-instantiated" view of the accelerator. On the other hand, the "theoretical", or "nominal" or "design" accelerator, typically exhibiting much greater symmetry than this, is most usefully viewed with a hierarchical organization.

The lattice model proposed here preserves both of these views. Predicated on the assumption that the accelerator is usefully analysed and operated as if it were almost ideal, we consider it important that the extra structures required for an instantiated lattice description be as inconspicuous as possible. Since the definition of what constitutes "essential" data is not particularly controversial, this report is more concerned with establishing terminology for describing the model than with the model itself. A computer file containing the data for this model will be called a standard machine file *SMF*, even though it may take the form of computer language structures, especially when internal to a computer program. This model is considerably more limited in scope and less ambitious than object models that encompass beam dynamics; *e.g.* Michelotti[‡] and Iselin[§].

To enhance the ability to communicate between processes, it is important to be able to segregate data that is common to most modeling codes (because they describe physical characteristics inherent to the elements) from data that are peculiar to particular codes (perhaps because they control processing algorithms.) To some extent this can be handled simply by maintaining but ignoring irrelevant data, but the structures advocated here attempt to make this segregation "natural".

Fully hierarchical descriptions have typically been expressed using the *Standard Input Format* (*SIF*[1],commonly also known as *Mad Input Language*, in one of its states of

† A "lattice model", as the term is used in this report, is a collection of numerical values of all parameters, possibly time varying, of the physical elements (not including circulating particles) essential to the operation of the accelerator.

‡ L. Michelotti, Towards C++ Object Libraries for Accelerator Physics, in AIP 292, Proceedings of Workshop on Stability of Particle Motion in Storage Rings, Brookhaven, 1995.

§ F. Iselin, The Design of MAD Version 9.0, Preliminary Draft, July 28, 1995 evolution starting from *Transport*) or by many variant, high level, languages. Since fully-instantiated formalisms have evolved independently in each lattice program and control system, they have not been standardized to the same degree. The intention here is to provide a mechanism for "remembering" the original design hierarchy, while at the same time supporting the individualization of some or all elements. This is consistent with a principle according to which data should not be discarded needlessly, especially if doing so clouds the original design structure of the lattice—this is what happens when small symmetry breaking alterations formally break design symmetries. The goal then is to have a single data set that retains both a highly symmetric view attractive to designers and the fully-instantiated view necessary for control. (A tiny preview of terminology to support these dual views: an individual element will have a *generic* name appropiate to its ideal, hierarchical existence and a *lattice* name[‡] appropriate to its flat, individual existence. If the modeling program is part of a "model-based" control system, the *generic* name has presumably been selected by an accelerator designer and the *lattice* name has been chosen by a member of (or committee of) Project Management.)

The proposed ability to "remember" design configurations is very limited; for example, changing the sequential order of physical elements or the global geometry is not allowed, and (other than a modest degree of "gangability") no form of "intermediate" organization other than partial expansion of the hierarchy is recognized.

To enforce the feature that "instantiated deviations be as inconspicuous as possible" our policy requires that only *deviations* from nominal values be recorded in fully-instantiated form. Before leaping to the conclusion that this constraint is capricious and arbitrary one should admit that it reflects operational practice in many cases. Especially during commissioning, one commonly has insufficient information to define meaningful deviation from design values. On the other hand, it is probably natural for hardware readout by a control system to recover total values. In this case, with the original design hierarchy being always known, deviations can be recovered as the instantiated total value minus the generic value. Furthermore, if there are deviations large enough to perturb operations severely they should probably be incorporated into the "ideal" lattice description—*e.g*, if the field integrals of all lattice dipoles are, say, 1% too great, that fact can be incorporated

[‡] The term "site wide name" or SiteWideName is also used for what we call the *Lattname*.

in the design so that there will then be no systematic deviation of that parameter. In our model all undefined deviations default to zero. This policy is consistent with a realistic approach in which deviations from the ideal are only included as they gradually become known, or conjectured, or are under investigation.

To recapitulate: maintaining nominal values in the generic description and only deviations from these values in the instantiated description—we have magnet strengths typically in mind—is an appropriate conceptual contraint, which can, in any case, be removed if required. Rather than "starting from a clean slate", in accelerator operation one is usually pressing forth on small salients from pre-established conditions into the as-yet unexplored.

There are many other examples of input to accelerator modeling programs taking the form of deviations; for example "per magnet" measured field expansion coefficients and measured survey locations. It is also natural to express the element strength changes recommended by the correction algorithms of a modeling program in the form of deviations from nominal (though nominal is commonly zero in this case.) Another type of deviation is Monte Carlo generated, internal to a modeling program. Finally, an even more ambitious generalization that is well represented by a *deviation* is a "ramped" or "dynamic" lattice parameter, such as hysteretic field strength, extraction octupole, bumper strength, RF frequency, *etc.*

It is intended, even for a single lattice, that there be more than one *representation* of the model being discussed. (We will use the term "equivalent representations" to refer to differently organized datasets that describe the same lattice.) As well as being required to be "equivalent" each valid representations will provide some of the following features:

- computer generated;
- human editable;
- computationally efficient;
- rapidly interchangeable—hence probably binary;
- capable of being output from one process and "piped" as input to another process. A process that accepts such a file as input and generates a revised file (normally of the same format) as output is called a "filter".[§]

[§] This functionality is copied from the *writefile/readfile* feature of *Teapot*. The filter functionality has been developed for the particular ideosyncratic needs of the *RHIC* project at *BNL*. The "thick" file, newly

- Sophisticated "book-keeping" in the face of multiple lattice variants—this probably implies manageability by a commercial database management system. (Naturally, in this case, the data for the model described here will constitute only a small fraction of the data being managed.)

The ideal lattice representation would be optimal for all of these, but it is assumed that several representations, each one optimized for a different purpose, will be used in practice. With disciplined adoption of the model, generation of translators from one representation to another should not be difficult.

What are the intended benefits of the object-oriented approach advocated here? Quite apart from any particular implementation, disciplined analysis of the problem in order to minimize the complexity of the data structures is presumably a good thing. The concept of representing the flat, instantiated, lattice strictly in the form of deviations from a generic ideal lattice is one product (which everyone may not agree with) of this discipline. Furthermore, the explicit spelling out of data structures in any particular implementation makes the data structures more easily accessible by other programs, especially if they are written in the same language. This should facilitate incorporating the powerful features of modern methodologies and programming languages. An "accesss library" capable of taking advantage of this feature and able to describe an arbitrary new element (such as an internal target, a beam-beam effect, or a real-time effect) has been described[3];it permits coding to be done without understanding or disturbing previous coding. It can also be used together with other mathematical or physical libraries (*e.g.* differential algebra.) Another purpose is to make the accelerator description accessible to modern operating systems, integrated environments (such as COMMON POINT[4]), real-time control systems,

introduced in this report, will replace the thin `fort.7` "machine file" accepted by those filters. An SDS[2] format will also be supported. (The most important modification is that element splitting into thinner elements, presumably for purposes of sympectic integration, though anticipated, is not performed at this level. Subsequently intended splitting of a thick element into $N+1$ thin elements is specified by new "element splitting" attribute `N`.) The purpose of most of the language features is to permit heterogeneous filters to process the same lattice files. It should not, however, be inferred that the language is intended to be in any way specific to *Teapot*; quite the contrary. The purpose of references to *Teapot* is to lend concreteness. The sort of flow of information envisaged is shown in Fig. 1 which applies to the RHIC accelerator. Though the shaded box is emphasized in this report, it is primarily intended as an example implementation of the object model.

distributed systems (such as CORBA[5]), databases (object-oriented or relational) and user interfaces.

A sensible strategy for reading this report might be to skip now to Appendix A.1 which contains a *SIF* description of a "toy" lattice and Appendix A.2 which contains the same lattice (with a few extra instantiations) transcribed into the *Ascii* representation of the model—explained in Section 3. The same lattice in its C++ representation—explained in Section 4—is given in Appendix A.3. To insure that legitimate C++ language is being used it is expressed as a valid C++ header file. But this is not intended to imply that compiling the lattice description into the code is either necessary or a good idea. Taking advantage of one of the tenets of the *Object Model Technique, (OMT)*[6] religion, "a good model can be understood and criticized by application experts who are not programmers",[6] the explanation of the C++ implementation is brief and cryptic. The next section gives the rationale, rules, and terminology of the object model.

2. Overview of Features

The important features of the proposed lattice model are listed informally here. In this report some organizational features are drawn from Rumbaugh *et al.*[6]. The first step in the Object Modeling Technique is supposed to be the generation of an Analysis Model which specifies what the system must do. The broad outline of this has already been given. Here we give more detail, defining terms, and describing examples by an *Ascii* "shorthand" (resembling *SIF*) intended to facilitate human (as contrasted with computer) intelligibility. Except for occasional preview examples, implementation mechanisms will be described later. Though the lattice description is to be called a "Standard Machine File" (*SMF*), this is not intended to imply that it is implemented as a true file of any particular computer language—"data set" would be more accurate.

The more important (and most likely to be ambiguous) terms are placed in quotation marks to indicate that they are being defined implicitly by their use in the accompanying sentences (or that their meaning is admittedly vague.) Technical language elements‡ are

‡ By and large, abbreviations are avoided. The only exceptions are the words *generic, element, attribute, parameter, lattice* which are abbreviated to *gen, elem, attrib, param, latt*, but only when they are prefixed as adjectives to other words.

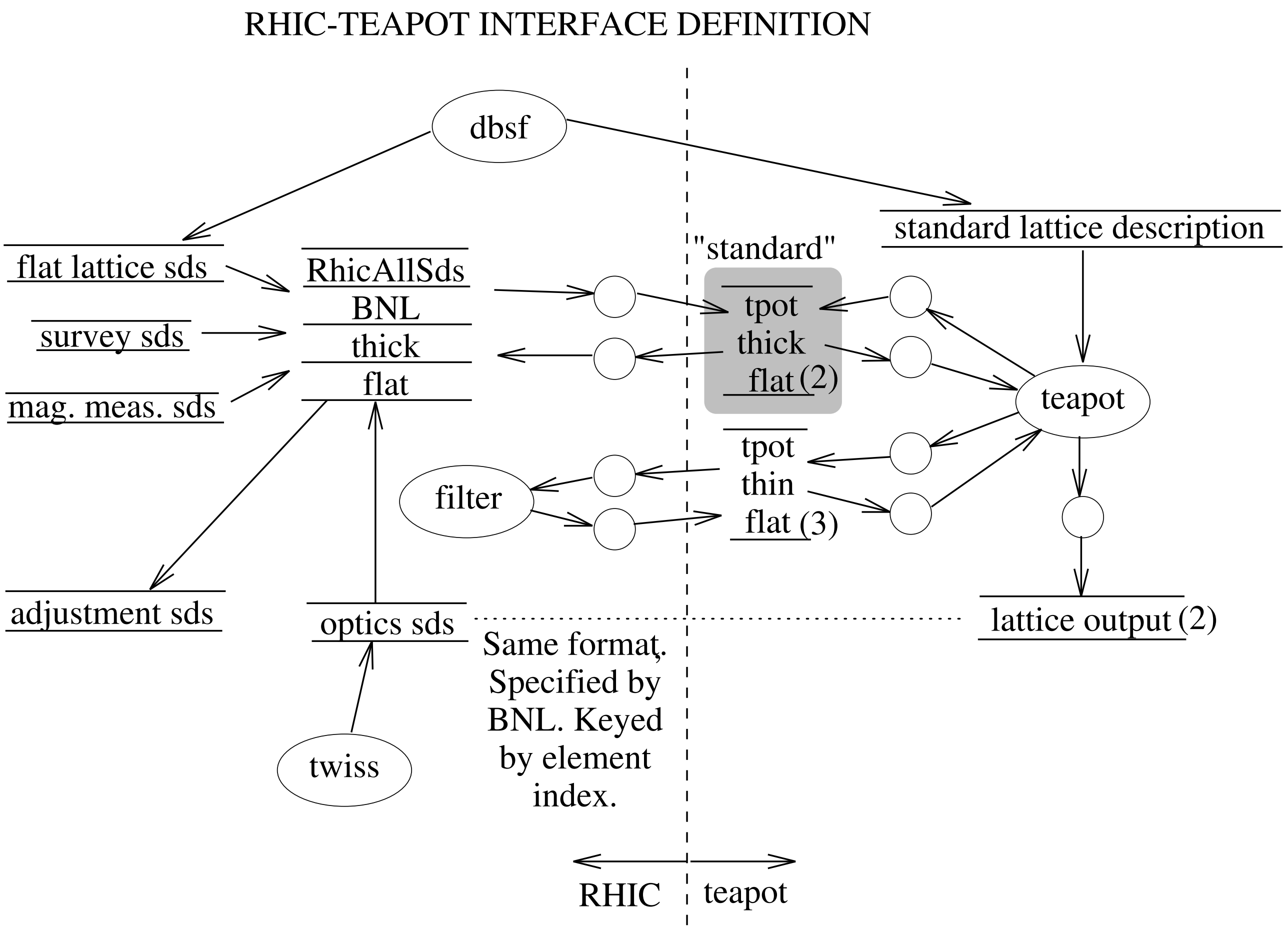


Figure 1: Flow of information back and forth between a modeling program and an accelerator control system. The *SMF* described in this report is shown as the shaded "tpot thick flat" data set. The number (2) in parenthesis indicates two formats. They are the *Ascii* representation and a C++ header representation (or an essentially similar self-descriptive standard (*SDS*) format). Though not shown, filter processing of this file is also possible.

indicated in *italic print*; especially important are those starting with a capital letter since they are names of classes in the C++ representation described in Section 4. Material in `typewriter type` represents actual entries in an Ascii *SMF*.

The following terms (with examples drawn from the *Ascii* representation) will be used:

- *Parameter*'s (typical names `ld, kq1,`...)

- *ElemAttribute*'s (such as the arc length `L` through the element) can be strength, position, *etc.*—physical properties an accelerator element can have †
- *GenElement*'s, (typical names `$quadhf`, `$sext1`,...) *GenElement* definitions include the assignment of *element type* (such as `QUAD`) and *ElemAttribute*'s
- *Line*'s (typical names before "line expansion" `^fullcell,^ring,` ..., same names residual after expansion `+fullcell, +ring,` ...)
- *LattElement*'s, in one-to-one correspondance with "physical" elements in the lattice, have the generic names and nominal values of *GenElement*'s and (optionally) "per element" attribute deviations and *LattElement* names (such as `_qf311`, `_fred`,...)

The proposed data set has various forms of organization. One is into "model levels", indicated by Roman numerals and containing the following categories of definitions:

I. *Parameter* names are introduced and assigned numerical values.

II. *GenElement*'s are defined and their attributes are assigned (eventually§) numeric values.

III. *Line*'s are defined. Unlike elements, for which two "kinds" are recognized, there is only one kind of line. Like *GenElement*'s, *Line*'s are generic. A *Line* can contain another *Line* nested within it, and so on. *Line* expansion is discussed below.

IV. *LattElement*'s are listed (eventually‡) sequentially, augmented as appropriate by all nonvanishing deviations from inherited generic values. Though primarily intended as a flat description it retains line names to permit hierarchical reconstruction.

(IV'.) This level is internal to *Teapot*; it is, strictly speaking, not part of the lattice model being discussed. It is included here to facilitate discussion.¶ Appendix

† It is also possible for an *ElemAttribute* to have logical character such as vacuum chamber `SHAPE` which might be `ellipse`, `rectangle`, *etc.* In the C++ implementation of Section 4, strings (`ellipse` for example) can be macro-replaced by numbers so that a single attribute type (numerical) is sufficient.

§ "Eventually" means "after complete processing". In this case, an attribute may be assigned a *parameter* that will eventually be replaced by a number.

‡ See previous footnote.

¶ In the (internal, but externally accessible) *Teapot* description some "physical" elements have been artificially subdivided for symplectic particle tracking purposes. This data is organized in almost strictly

B.1 shows how an extra level such as this fits into the model. It is labeled (IV') so that the following level can be called V.

V. Definition of ganged families. Interpretation of "logical" element attributes (such as are described by the "type" feature of the *SIF*).

Only level IV and (IV') are truly flat. It would be possible to relax the requirement that the data be listed sequentially in the order I, II, III, IV, V. On the other hand, since the data sets are machine generated, such flexibility seems unnecessary. So level I data comes first and is completed, then II, and so on.

Some general features of the model are listed next, roughly in order of importance, with most important first, and with some repetition of points already covered.

- Each *LattElement* inherits a *GenElement* name from level II, implicitly acquires a sequential index, and is (optionally) assigned a fully-instantiated name appropriate for flat description. *GenElement* names and *ElemAttribute*'s[§] assigned in level II are inherited by *LattElement*'s in level IV where instantiated deviations are also (optionally) assigned.[‡]
- The lattice description includes a "flat[†]" list of elements in one-to-one, sequential correspondance with "thick" elements of the actual lattice. The term "thick" is synonymous with "physical" which is the level of differentiation at which an element is individually positioned and powered, for example in a control system. Because of the sequential nature of the enumeration, each element has a unique sequential integer index (lattice beginning is 0) which must obviously be recoverable from the lattice description. This index can serve as a key for correlating with other files containing data such as lattice functions, power supply currents, surveyed positions, *etc.*

---

flat form, even to the extent that all element record formats are identical. It is available in various forms: formatted (editable or "slow") *Ascii*, unformatted (fast) *Ascii*, and *SDS* (self-descriptive standard) data sets.

§ In greater detail, there are two distinct types of attributes: numerical attributes and string or "logical" attributes (such as `SHAPE=ellipse`). The term *ElemAttribute* subsumes both meanings.

‡ Another (somewhat less essential) new feature being introduced is that each attribute is also assigned an r.m.s. uncertainty. For most parameters this feature is superfluous since no such uncertainties are available. As usual in this language, this defaults to the parameter's known perfectly. Note that this uncertainty, being generic, is defined in level II.

† To the extent that "flat" refers to a data file, we define it to mean one record per element, sequentially listed, but do not require each record to have the same format.

- The term "flat" also conveys the meaning that any element attribute can be assigned to any element. To support this feature any undefined element attribute is zero by default.
- The model is "complete"; all data required for reconstruction and subsequent analysis of the lattice is contained, and this must remain so when the file is passed through a filter. Any element attribute (for example a strength) can be assigned an instantiated deviation value whether or not the strength was introduced generically in the original design.
- A hierarchical lattice tree is based on lines as in the *SIF*. This provides a mechanism for recovering the original hierarchical organization of the lattice even after the symmetry has been broken by individualization of some of its element names or attributes. This ability relies on the capability of defining and naming lines as well as on the representation of attribute values as design values plus deviations.
- All language elements of the model should be "extendable". Hence the list of reserved *GenElement*'s types (such as `QUAD`), and the lists of reserved numerical *ElemAttribute*'s (such as `ANGLE`) and logical *ElemAttribute*'s (such as `SHAPE`) can be extended in *ad hoc* fashion. Since such extensions tend to defeat effective communication, it is important to have an agreed-to vacabulary of recommended types and attributes initially—the reserved attributes defined in the *SIF* will constitute the majority of "reserved" names. But this list can be augmented later without important restriction and to emphasize the point no lists of reserved names are given in this report. Restoring the loss of consistency between written and read files implied by introducing new elements or attributes is the responsibility of the user(s). On read-in, unrecognized elements and their values, if any, are ignored. This is a mechanism allowing the same data file to be used by diffferent programs; information specific to one program can be ignored by another.
- In the *SIF*, two elements whose strengths are given by the same symbolic parameters are implicitly "ganged" together. The same mechanism applies here. A moderate further degree of "gangability" is also supported. It permits the

declaration of individual elements as belonging to a family for which one specified parameter is allowed to be varied, with the changes constrained to be the same for all family members.† To support unequal weighting such as "equal but opposite" element change functionality, a numerical weight (defaulting to unity) can accompany the family designation. (Incidentally, no arithmetic operations are allowed in parameter definitions.)

- Free format. Only non-zero entries need be made, and in any order, except a numerical value must follow its corresponding symbolic name. There may be other *ad hoc* syntactical rules governing the sequencing of *ElemAttributes* and their values within a *GenElement* or *LattElement* definition.
- The model supports different formats optimized for different purposes. These include sequential-*Ascii* and C++ header file format. The ability to locate and edit parameters by hand in at least one format is essential, but ease of editing is not given a high priority (since complicated alterations should be performed some other way).

3. Implementation Mechanism for the *Ascii* Representation

Hierarchical organization can be recovered from the "flat" sequential level IV *Ascii* listing by appropriate processing of the *Line*'s. The fact that *Line*'s may be nested in level III is the only complication.

The simplest (and least interesting) possibility would have no *Line*'s defined in level III. In that case the sequence of elements listed in level IV would be explicitly flat. The next simplest (and not quite empty of significance since it might model an ideal accelerator) has no deviations introduced in level IV. In that case, *Line* names appearing in level IV can be "macro-expanded" by replacing the *Line* name by its "macro" definition available

† Correction schemes often employ "families" or "gangs" of elements (for example correlated beam adjusters) that are established independent of the original design hierarchy of the lattice. Such newly imposed families can "increase the symmetry" by constraining to be equal certain strengths that would have been independent because they belonged to elements that were generically independent in the original design. On the other hand, "symmetry reduction" results if a newly defined family distinguishes (for example by inclusion or exclusion) elements that were generically equivalent in the original design. The "increased symmetry" case can be described while respecting the original hierarchy. In the "reduced symmetry" case the original hierarchy can be respected as regards the ideal lattice but must be violated at the level of deviations.

from level III. Actually "replace" in the previous sentence is not quite right—the original *Line* name remains but, marked as having been expanded, it serves as memory of the ideal hierarchy but is invisible to the flat view. At this point we expand the terminology slightly by saying that the *Line* name is *alive* while it remains unexpanded, and *dead* thereafter. The simplest valid complete example of the level IV *Ascii* description of an ideal accelerator would, for example, be `^idealRing`, a *Line* defined in level III.

Practical cases are intermediate between the two extremes described in the previous paragraph. Suppose, again starting with the simplest possibility, that the ring being described were the `^idealRing` introduced above but with a single parameter changed from the ideal. In that case the same macro expansion occurs until the *LattElement* whose attribute needs changing becomes exposed. At that point macro expansion ceases and the parameter deviation is entered. This involves (logical) concatenation of the generic and instantiated records. Rather than requiring a parameter change, suppose the *LattElement* needs to be named—same process, expand until exposed and add the name. When more deviations or names are required the macro expansion process continues as necessary. On read-in full expansion presumably takes place.

It is pretty clear, for practical lattices, with realistic elements, in any actual accelerator, that at least the "upper", and possibly all, *Line*'s will have been expanded in the working lattice file. Typically then, the file will be at least hundred's of times longer than the sample file in Section A.2.

At any stage in the expansion process the list will consist of *LattElement*'s and sub-lists of *LattElement*'s and unexpanded *Line*'s contained in curly braces, { and }. Since this looks much like the block definition of the computer language *C*, it seems that "pretty-printing" level IV will recover the hierarchical structure visually. On the other hand, full expansion, with suppression of the curly braces and "dead" names, yields a flat listing.

The workability of this scheme can perhaps best be contemplated by viewing the example of Sections A.1 and A.2. A somewhat klunky feature of the scheme, even with *Line*'s fully expanded, is that some elements are *GenElement*'s and some are *LattElement*'s. (This may mirror operational practice in an actual accelerator where some elements may never acquire *LattElement* names.) To cure this formal defect our policy is to make a dummy entry _ for elements without instantiated names in the place a *LattElement* name would normally appear. A fully expanded list then has a *LattElement* name for every *element.* This has the further advantage that even the shortest possible *LattElement* record, [_], is not empty. In a fully expanded list sequential *element indices* can be obtained simply by counting occurences of [· · ·] pairs.

As implied by the miscellaneous examples given so far, the "language" elements are differentiated by special prefix characters, for example as in *Perl*, but with some syntactical features reminiscent of *TeX*. A complete data set is nothing more than a sequential stream of tokens. The *TeX* conventions for expanding an input line into a sequence of tokens can be copied directly. This includes treating end-of-line characters as equivalent to spaces (except that comments, started by special character %, terminate at the next end-of-line) and regarding multiple spaces as equivalent to single spaces.[‡]

The concept of "category code" can also be copied from *TeX* with a few of the codes to be interpreted much as in "plain" *TeX* and the remainder adapted to the present task. The suggested category code definitions (with examples in parenthesis) are:

0. \ prefix to reserved name in this language (`\III`)
1. { Start of expanded *Line* (as in `+ring{^fullcell ^fullcell}`)
2. } End of expanded *Line* (as in `+ring{^fullcell ^fullcell}`)
3. $ Prefix to *GenElement* name (`$quadhf,$sext1`)
4. + Prefix to *dead Line* name (as in `+fullcell`) (The + is mnemonic for gravestone)
5. eol End-of-line
6. Not used at present
7. ^ Prefix to *alive Line* name (`^fodo`) (The up arrow is mnemonic for "look above")
8. _ Prefix to *LattElement* name (`_qf311`, `_qd312`). Underscore _ alone serves as dummy name for no-name *LattElement*'s
9. Available
10. Space
11. [Beginning of *GenElement* or *LattElement* record

[‡] Unlike TeX, two or more consecutive end-of-line characters (*i.e.* blank lines) are also equivalent to spaces.

12.] End of *GenElement* or *LattElement* record
13. Special character that can be part of variable name
14. % Start of comment lasting to next eol
15. Letter

Miscellaneous conventions. *Element type* names are to be upper case and fixed length (*e.g.* `QUAD`, not `quad` or `QUADRUPOLE`). *Element attribute* names (such as `ANGLE`) are also to be upper case and full length. *Parameter*, *GenElement* and *LattElement* names are case sensitive, begin with a letter, and can consist of any combination of upper and lower case letters, numbers and (for element names only) underscores).

4. C++ Implementation

This document is loosely based on the Object Modeling Technique[6], particularly on its object model. The previous sections have contained the data requirements and can be considered to be the first stage of the object-oriented methodology, namely the Analysis Model. The next stage, Object Design, describes the structure of the objects in the system and their relationships. It should not depend on a particular programming language; rather it should be based on classic data structures or container classes (such as *Vector*, *Dictionary*, *etc.*) that can be provided by most object-oriented languages as part of their predefined library. We use (with minor modification) conventional object diagrams and notations of this methodology. For example, a diamond indicates aggregation and a triangle inheritance. We introduce one *ad hoc* notation—a "switch" between objects—which means that an object of the class can be initialized one way or the other, with the or being exclusive. For the sake of clarity, the examples in this section are implemented in C++; the data structures are shown in Appendices.†

In the model each accelerator program is considered as a sub-class of the *Pac* (Platform for Accelerator Codes)[3] class. The internal data structures and methods of such codes, being determined by particular algorithms and developer's styles, may be different. But,

† In the figures some of the so-called "classic data structures" such as *Vector*, which is a fixed-size collection of values of uniform type, *List*, similar except the number of elements is unknown in advance, and *Dictionary* appear. Also the use of class templates is indicated. For example, *Vector*<Parameter> is a class generated by template Vector<class T>.

as if "derived" from the *Pac* class as regards lattice description, each code inherits the ability to communicate with other codes via a common data set of class *Lattice*.

The structure of the class *Lattice* being in one-to-one correspondance with the *SMF* described in Sect. 2, data is organized in the following levels:

- Level I: vector of *Parameter*'s. Instances of class *Parameter* include the name and value (Appendix B.1) and share the static member *Dictionary* < *String, Parameter**> *collection* that services the unique set of their names.†
- Level II: vector of *GenElement*'s. In accordance with the proposed flat description, each physical element (class *GenElement*) has the same internal structure and can be implemented as a direct sum of linear spaces of design attributes (class *ElemAttributes*) and their r.m.s. uncertainties (class *Error*) (Appendix B.2.1):

double	*lq*	= 0.6;
double	*kq1*	= 0.3789;
Error	**quadhf_rms**	= kq1*0.01***K1**;
GenElement	**quadhf**	= lq***L** + kq1***K1** + **quadhf_rms**;

Here **L** and **K1** are SIF attributes, and GenElement object **quadhf** is an element with length equal to 0.6 m and gradient 0.3789 m^{-1}. In this example the quadrupole gradient is defined with 1 percent uncertainty. Notationally, quantities like **L** and **K1** can be thought of as "unit vectors" along the axes of element length and quadrupole strength, the * operator is overloaded to convey "multiplication" by a "scalar" (not to be confused with *pointer*) and the + operator is overloaded to convey vector addition. Because each physical element has an arbitrary number of attributes, it seems useful to implement its data as a dictionary (associative array) linking some predefined basis vector *ElemAttribID* (such as **L, K1**) with its corresponding attribute value *ElemAttribValue*. The instance of class *ElemAttribValue* can be defined by numerical value or by *Parameter* (Appendix B.2.2). Additionally it includes a pointer to

† The word *collection* is used here to reduce ambiguity that might result from the fact that the more idiomatic word *store* can be either a noun or a verb. The word static associated with *Dictionaries* in the figures also conveys the meaning *global*.

some (potentially time varying) strength that describes the real physical processes (magnetic field ramp, power supply ripple *etc.*) The class *GenElement* serves as base class for all different types of physical elements, such as *Sbend*, *Quadrupole*, *etc* (Appendix B.2.3). These classes have the same structure and only differ by variable *type*.

- Level III: hierarchical tree of *Line*'s and *GenElement*'s. Design of this structure is presented in Appendix B.3. A tree consists of a collection of nodes, instances of class *Line*. A physical element is considered as a *leaf* in this hierarchy and defines the corresponding *Line*'s member. The interior node, *subtree*, is initialized by concatenation of *GenElement*'s and *Line*'s:

Quadrupole	**quadhf**	= lq***L** + kq1***K1**;
Sextupole	**sext1**	= ls***L** + ks1***K2**;
Line	**fullcell**	= **quadhf * sext1 ***. . . ;
Line	**ring**	= **fullcell***. . . ;

- Level IV: vector of *LattElement*'s. This structure represents a flat list of elements in one-to-one correspondence with actual physical lattice elements. The data of *LattElement* is implemented as a superposition of design attributes (*GenElement** genElement) and their deviations (*ElemAttribute* deviation).
- Level V: definitions of "families" and "ganging" is being worked on.

In addition to physical data, the classes *Parameter, GenElement*, and *Line* include also a static collection of pointers to their instances. The set of these collections can be regarded as a "distributed" database that serves several purposes. For example it provides the unique set of object names and lattice sectors permitting the same lattice sector to be shared by different rings, as in

Lattice	`arc1` << "ARC1"; // *Definition by*
Lattice	`arc2` << "ARC2"; // *external files*
Lattice	`ir` << "IR";
Lattice	`ring1` = `arc1` ∗ `ir`;
Lattice	`ring2` = `arc2` ∗ `ir`;

A C++ header file that describes the toy fodo lattice is presented in Appendix A.3. We regard it to be obviously advantageous, in comparison with the present *SIF* standard language, to have the lattice described in the same programming language as the modeling code.

Appendix A.1. Sample *Ascii* (*SIF* a.k.a. *MAD*) Standard Input Format

The following *SIF* listing is to be compared with the listing in Appendix A.2. At the hierarchical level the two listings describe identical lattices, but there are differences at the flat level. The last five entries of this file are examples of instantiated deviations supported by *Mad*.

```
TITLE
"Toy fodo lattice"
!
! Level I  - define parameters
lb  = 4.0
lq  = 0.6
ls  = 0.3
ld  = 0.15
deltheta = 0.62831853
kq1 =  0.3789
kq2 = -0.4132
ks1 =  0.07
ks2 = -0.14
!
! Level II  - define generic elements
bend   : sbend, l = lb, angle = deltheta
quadhf : quadrupo, l = lq, k1 = kq1
quadvf : quadrupo, l = lq, k1 = kq2
sext1  : sext, l = ls, k2 = ks1
sext2  : sext, l = ls, k2 = ks2
drift1 : drift, l = ld
!
! Level III - define hierarchical structure
fullcell : line  = ( &
 quadhf, drift1, sext1, drift1, bend, drift1, sext2, drift1,&
 quadvf, drift1, sext2, drift1, bend, drift1, sext1, drift1)
ring    : line = (5*fullcell)
!
```

```
! Commands
use, ring
!
!Level IV - define misalignments and measured field of
!individual elements
ealign, bend[3], DX = -0.003, DY = -0.003
ealign, bend[4], DX = -0.001, DY = -0.001
ealign, quadvf[2], DX = -0.003, DY = -0.003
efield, quadhf[2], DKL(1) = 0.0024
efield, quadvf[2], DKL(1) = -0.0024
```

Appendix A.2. Thick Flat Lattice *Ascii* Representation of the Same Lattice

This file, written in the *Ascii* representation of the model described in this report, is to be compared with the previous appendix. The main extra attribute that would have to be included for *Teapot* or any other symplectic integration program would be the parameter `N` for those elements requiring subdivision. As explained in the text, any element attribute potentially appearing in level II can validly appear in level IV, but in level IV its assigned value is interpreted a deviation from the generic value inherited from level II.

```
% Tpot thick flat file ( ASCII-formatted )
% Toy fodo lattice
%
\I
lb  4.0
lq  0.6
ls  0.3
ld  0.15
deltheta 0.62831853
kq1  0.3789
kq2 -0.4132
ks1  0.07
ks2 -0.14
%
\II
$bend[SBEND L lb ANGLE deltheta]
$quadhf[QUAD L lq K1 kq1]
$quadvf[QUAD L lq K1 kq2]
$sext1[SEXT L ls K2 ks1]
$sext2[SEXT L ls K2 ks2]
```

```
$drift1[DRIFT L ld]
%
\III
^fullcell{
 $quadhf $drift1 $sext1 $drift1 $bend $drift1 $sext2 $drift1
 $quadvf $drift1 $sext2 $drift1 $bend $drift1 $sext1 $drift1}
^ring{^fullcell ^fullcell ^fullcell ^fullcell ^fullcell}
%
\IV
+ring
   {
   ^fullcell
   +fullcell
      {
      $quadhf[ _qf311 K1 0.004 ]
      $drift1[ _ ]
      $sext1[ _fred ]
      $drift1[ _ ]
      $bend[ _ ]
      $drift1[ _ ]
      $sext2[ _ ]
      $drift1[ _ ]
      $quadvf[ _qd312 DX -0.003 DY -0.003 K1 -0.004 ]
      $drift1[ _ ]
      $sext2[ _S_123 ]
      $drift1[ _ ]
      $bend[ _ DX -0.001 DY -0.001 ]
      $drift1[ _ ]
      $sext [ _ ]
      $drift1[ _ ]
      }
   ^fullcell
   ^fullcell
   ^fullcell
   }
```

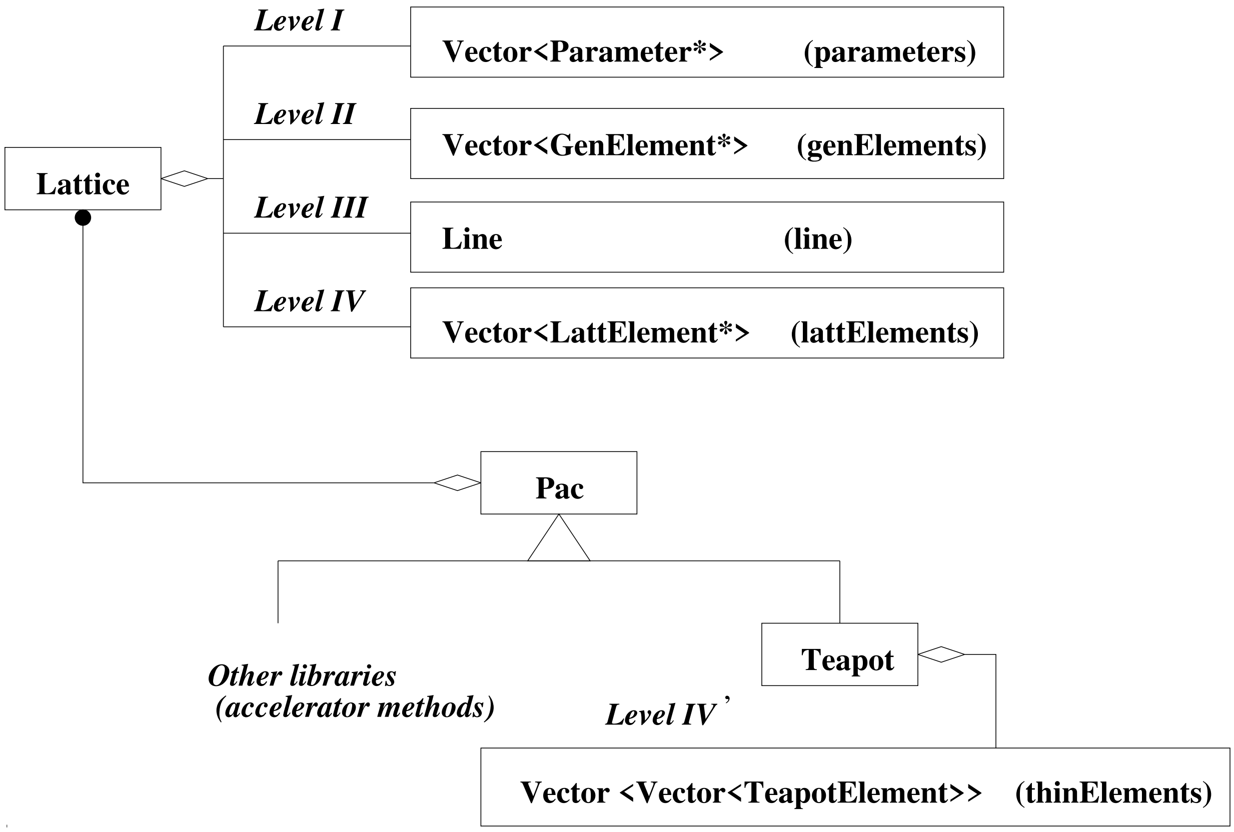
Appendix A.3. Header File, C++ Implementation of Same Lattice

This is a C++ header file representation of essentially the same lattice as described by the previous two pages. This is intended for illustration only and does not pre-suppose that compiling the lattice description into the modeling code is necessarily an economical approach.

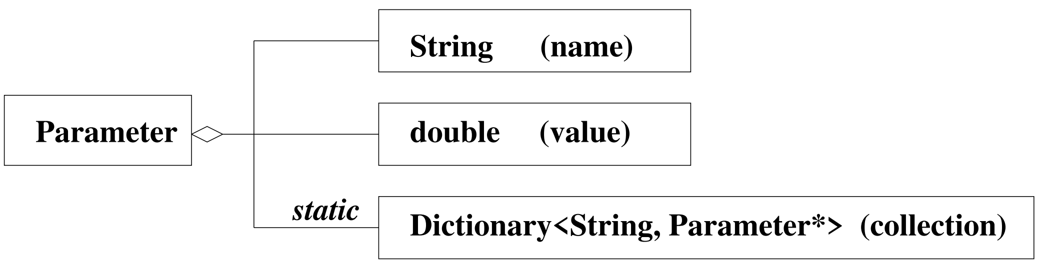
```
// fodo.h
// Toy fodo lattice
//
// Level I  - define parameters
Parameter
 lb("lb", 4.0),
 lq("lq", 0.6),
 ls("ls", 0.3),
 ld("ld", 0.15),
 deltheta("deltheta", 0.62831853),
 kq1("kq1", 0.3789),
 kq2("kq2",-0.4132),
 ks1("ks1", 0.07),
 ks2("ks2",-0.14);
//
// Level II  - define generic elements
Sbend  bend("bend", lb*L+deltheta*ANGLE);
Quadrupole
     quadhf("quadhf", lq*L+kq1*K1),
     quadvf("quadvf", lq*L+kq2*K1);
Sextupole
     sext1("sext1", ls*L+ks1*K2),
     sext2("sext2", ls*L+ks2*K2);
Drift  drift1("drift1", ld*L);
//
// Level III - define hierarchical structure
Line  fullcell("fullcell",
      quadhf*drift1*sext1*drift1*bend*drift1*sext2*drift1*
      quadvf*drift1*sext2*drift1*bend*drift1*sext1*drift1);
//
// Define lattice
Lattice ring("ring", power(fullcell,5));
//
// Level IV - define misalignments and measured field of
// individual elements
Sequence fullcell_1("", ring.sequence("fullcell", 1);
//
fullcell_1[ 0].define("qf311", 0.004*K1);
```

```
fullcell_1[ 2].define("fred");
fullcell_1[ 8].define("qd312", -0.003*DX-0.003*DY-0.004*K1);
fullcell_1[10].define("S_123");
fullcell_1[12].define("", -0.001*DX-0.001*DY);
```

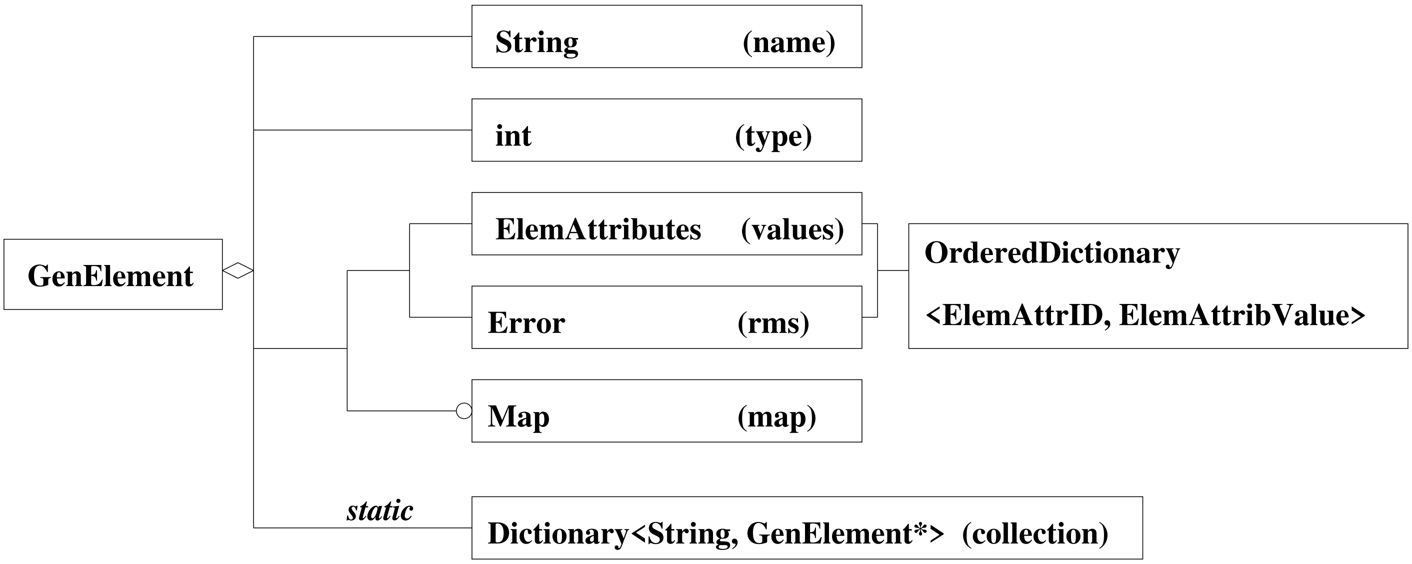
Appendix B. Accelerator Lattice Object Model.



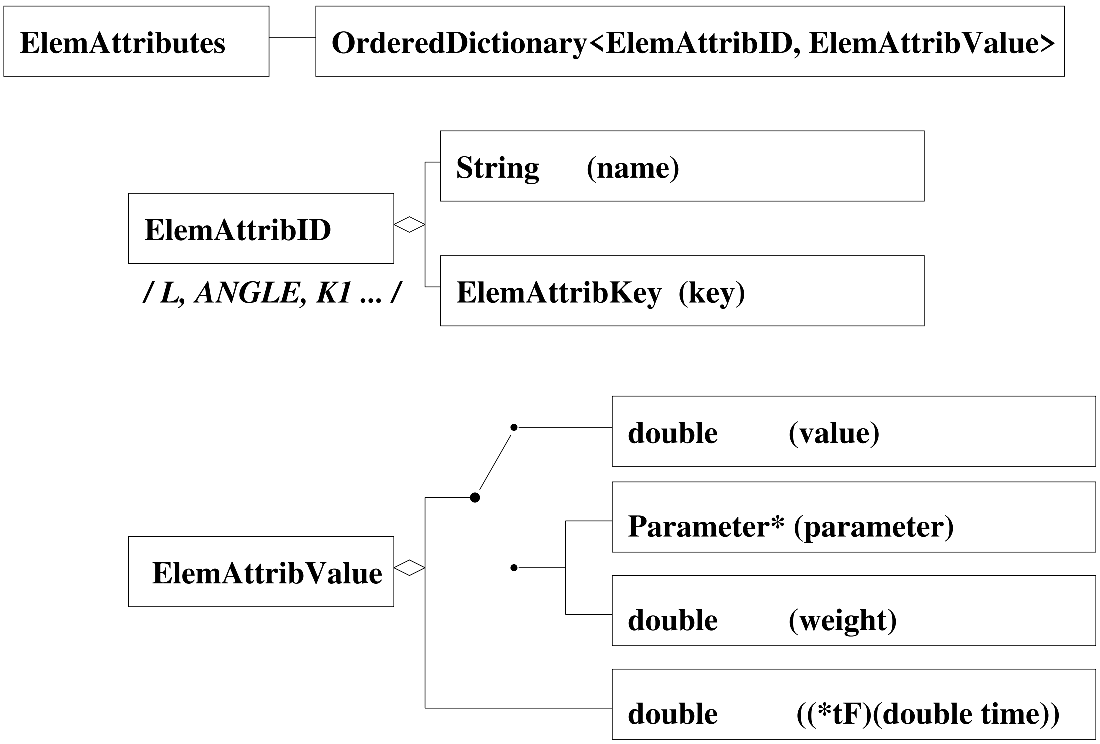
Appendix B.1. Level I. Parameter.



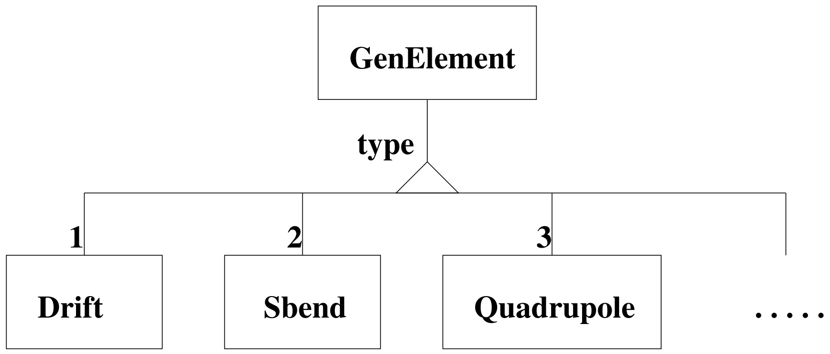
Appendix B.2.1. Level II. Generic Element.



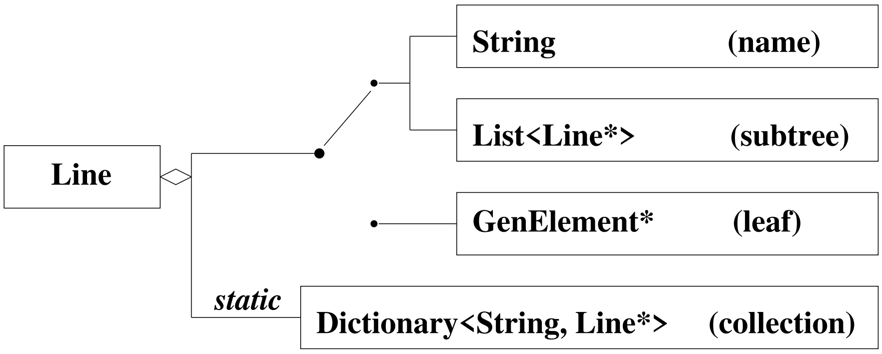
Appendix B.2.2. Level II. Element Attributes.



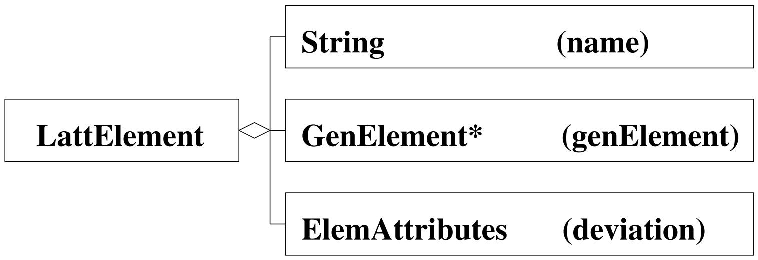
Appendix B.2.3. Level II. Physical Elements.



Appendix B.3. Level III. Line.



Appendix B.4. Level IV. Lattice Element.



5. References

1. Standard Input Format, Snowmass, 1984.
2. C. Saltmarsh, RHIC AP Note 29, 1994.
3. N. Malitsky, A. Reshetov, and G. Bourianoff, SSCL-675, 1994.
4. S.Cotter and M.Potel, *Inside Taligent Technology*, Addison-Wesley Publishing Company, ISBN 0-201-40970-4.
5. CORBA reference document, accessible via http as http://www.acl.lanl.gov/sunrise/DistComp/Objects/speclist.html.
6. J.Rumbaugh, M.Blaha, W.Premerlani, F.Eddy, W.Lorensen, *Object-Oriented Modeling and Design*, Englewood Cliffs, New Jersey, Prentice Hall, 1991.