Qualification and Management of Analysis Attributes with Application to Multi-Procedural Analyses for Multichip Modules

by

Vincent S. Wong

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Approved:

Prof. Mark S. Shephard

Thesis Advisor

Rensselaer Polytechnic Institute

Troy, New York

May 1994

(For Graduation August 1994)

Table of Contents

List of Tables

List of Figures

List of Pseudo-Codes

ACKNOWLEDGMENT

As with any major undertakings in life, behind the work of this thesis is a host of people who guided, supported, and generally encouraged me from the start to the completion of the process. I am indebted to Dr. Mark Shephard for his mentorship and great patience in guiding me through this work. Aside from technical aspects, he has also taught me valuable lessons in project planning and presentation. I also want to thank Dr. Peggy Baehmann and Mark Beall and other colleagues at SCOREC for their help in the many technical issues. Our many hours of discussions had definitely cleared the way for this manager to take shape as it is. I must also thank Rao Garimella for his valuable input to the work of the manager, and it was a pleasure to work alongside him through the watches of the night to get the project completed on time.

Even with all the technical support my adviser and colleagues gave me, the completion of this work would not have been possible without the emotional encouragements from my loved ones. I want to thank my parents, brother, and my fiancee for their love, patience, and their many prayers lifted up on my behalf. Of course, without the enabling, strength, and hope of my Lord and Saviour Jesus Christ, sustaining me in His love, none of this would be possible.

ABSTRACT

Presented in this thesis is a generalized attribute manager that provides a consistent analysis framework in which analysis data may be defined, organized, and associated with appropriate model entities. An attribute is that information, beyond the geometric domain, needed to qualify the physical problem to be solved. This thesis is divided into two major portions: 1) presentation of the design and implementation of the SCOREC Attribute Manager (SAM), and 2) presentation of the application of the manager in conjunction with the preprocessing activities of the Rensselaer Electronic Packaging Analysis Software (REPAS) project.

The first portion of this thesis focuses on the design and implementation of the attribute manager. From the viewpoint that every attribute may be characterized as a tensor, four aspects of a tensor are used to qualify an attribute: the attribute tensor order, its symmetry, the coordinate system with respect to which the tensor is defined, and the distribution functions specifying the tensor components of the attribute. In addition to the qualification of an attribute, the design and implementation of an attribute organizational framework is also presented. The organizational framework implemented is a flexible four-level hierarchy structure. This structure allows for organizing attributes appropriate to single or multiple analysis cases. To complete the specification of an analysis problem, the attributes must be associated to analysis model entities. Issues of this relational aspect of the manager are discussed, with minimum capabilities implemented. Finally, a suite of application interface operators, as well as I/O formats for SAM, are presented. These operators allow analysis applications to use the facilities of the attribute manager.

The second portion of this thesis focuses on the integration efforts of the REPAS project. SAM was used as the information manager that seamlessly tied together the global/local thermal/thermomechanical analyses. One source, in the form of three input files, is used to build the necessary idealized models for the analyses. The formats of these input files are presented, as well as an overall picture of the preprocessing step from the users' point of view. The approach taken in the data preprocessing is to first build a physical model from the given data of the interconnect. When the physical model is complete, most of the required attributes are defined, organized, and associated to the physical model entities using SAM operators. The data structure and algorithm for building the physical model is presented. The analysis models are derived from the physical model. Of the three analysis models, only the construction of the global layer-wise model of the interconnect is presented.

1 INTRODUCTION

Previous efforts at the Scientific Computation Research Center (SCOREC) had focused on finite element algorithms and methods. The maturing of such methods and the demand for faster performance and wider scope of geometric representation began to drive a shift towards building more comprehensive data structures (for example, Weiler's radial edge data structure [23][24] to handle complete non-manifold modelling information, then mixed-mesh data structure [22] followed by reduced mixed–mesh structure [3]). Increasingly, demands for automated analyses and multi-discipline/ multi-procedural solution methods are pushing the data structure to capture analysis information beyond that of the geometric domain, defined here as *attributes* [17][18]. Attributes include (but are not limited to) material properties, loads, and boundary conditions. Because the nature of multi-procedural analyses involves common data sets and different analysis procedures may model the analysis domain differently, a more efficient attribute handling mechanism must be in place to manage the potentially large number of attributes required for the analyses. Furthermore, each attribute must be attached to the appropriate domain entities. Thus, recent efforts involving multi-discipline multi-analysis modules require a much more complex set of data structure to accommodate the demand. The Rensselaer Electronic Packaging Analysis Software (REPAS) project to analyze the thermal and thermomechanical properties of Multi-Chip Modules (MCMs) is such an example [4][5][19].

An MCM is composed of a complex network of signal paths that span multiple dielectric layers. As such, it is not practical to model every intricate detail of the entire MCM for analysis — only the critical areas require detailed analysis. The

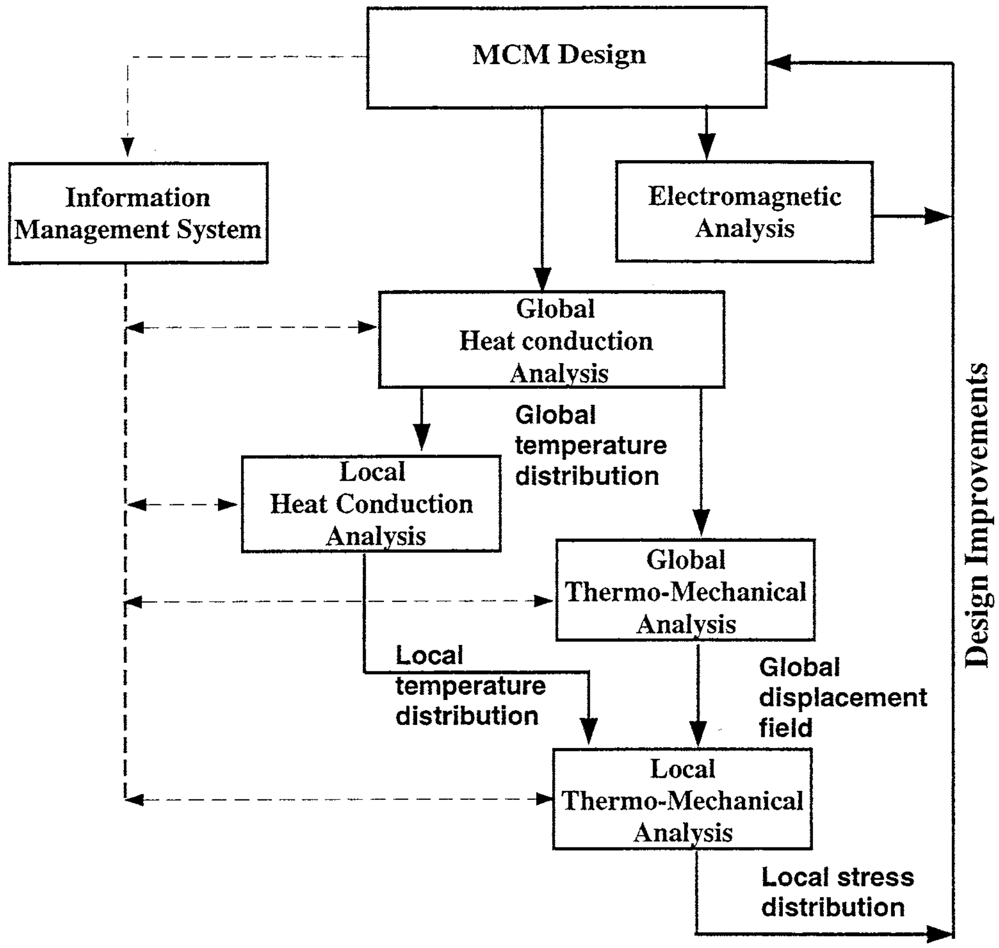

MCM Design
Information Management System
Electromagnetic Analysis
Global Heat conduction Analysis
Global temperature distribution
Local Heat Conduction Analysis
Global Thermo-Mechanical Analysis
Local temperature distribution
Global displacement field
Local Thermo-Mechanical Analysis
Local stress distribution
Design Improvements

Figure 1 REPAS Analysis Modules Overview

approach taken in the REPAS project is to analyze the interconnect in several stages, as shown in Figure 1. First, the MCM is idealized by modelling the interconnects in a layerwise fashion. The effective layerwise properties of the interconnect is generated from an averaging procedure based on the volume fraction of the materials in each layer. This idealized layerwise model of the interconnect is used by a set of global thermal and thermomechanical analysis procedures based on a variational approximation technique[21][15]. The steady-state temperature from the global thermal analysis is passed as thermal loading conditions onto the global thermomechanical analysis. The results from the two global analyses show the critical areas affected by the given temperature load. In turn, a set of local analyses are invoked to focus in on these critical sections. The calculated temperature field from the global thermal analysis is passed on to a local thermal analysis procedure as boundary conditions. Based on a fast random walk technique [11], this local thermal procedure calculates the temperature distribution of a detailed representation of the critical domain. The calculated displacement field from the global thermomechanical analysis and the temperature field from the local thermal analysis are then passed to a local thermomechanical analysis procedure as boundary conditions. This local thermomechanical procedure utilizes adaptive finite element techniques [2] to calculate the displacement and stress fields of the corresponding critical section of the MCM. Along with the results from the electromagnetic analysis, this knowledge is fed back to the MCM design for any design improvements.

An important criterion of the REPAS project is that the execution of and communication between one analysis and the next be seamlessly integrated. Such integration requires that the analyses can efficiently communicate with each other without any information loss or duplication. The information must be consistently defined and must be easily accessible. The set of information required for the analyses must also be flexible in that new information required can be easily integrated into the analysis process. The integration effort must also involve providing support for the generation of analysis models suitable for each analysis with the attributes attached to the correct analysis model.

The REPAS project is but one example of the requirements of multi-analyses integration efforts (another similar requirement can be found in the composite project). In response to this need, a generalized attribute manager, the SCOREC Attribute Manager (SAM), was developed. This manager coordinates the definition and flow of analysis data from one module to the next, as shown in the role of the "Information Management System" module of Figure 1. The basic focus of this thesis is on SAM and the preprocessing step needed to provide for the modelling requirements of the REPAS project. The approach of this thesis is to first describe the generalized attribute framework in detail. Chapter 2 discusses the theory and design behind SAM. Chapter 3 details implementation specifics and related issues of SAM, and chapter 4 documents the interface operators, as well as the format of the I/O files, developed for applications to make use of the manager. The set of interface operators provides a foundation for powerful analysis tools to dynamically change the input attribute during an analysis on an as needed bases.

Having laid down the foundation of the framework with the attribute manager, SAM is used as the basis to integrate all five analyses into a seamless whole. The remaining chapters discuss the preprocessing activities that are closely tied with the manager, with Chapter 5 beginning the discussion with introductions to some of the issues facing the data preprocessor. Figure 2 shows the scheme used for the REPAS Information Management System. For the purposes of discussion, all analyses that query SAM for attribute information are called *client analyses*. For this REPAS project, the input source comes from three files: start-up file, CIF file, and attribute file. Chapter 6 describes each of these three files as the initial data requirements for the client analyses. A set of preprocessing routines parse the input data, construct the global model, call the attribute manager to create the appropriate attributes, and associate the attributes to the model entities created in the preprocessing routine. Chapter 7 describes the preprocessing stage from the point of view of the user. Chapters 8 and 9 look at the preprocessing from the implementation point of view — describing the building of the physical model and global idealized models. At the conclusion of the preprocessing step, all of the attributes

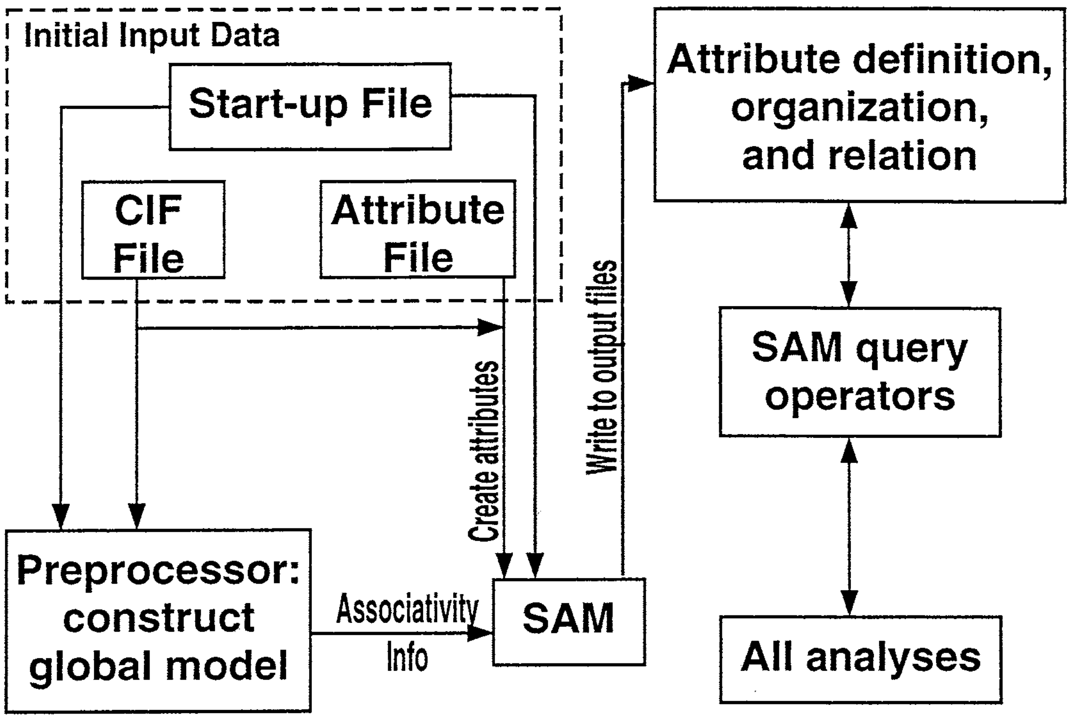


Figure 2 Schematic of of SAM as the Heart of Integration for the REPAS Project

for the client analyses are defined in SAM. This information is written into output files, which are used as input files for clients that wish to use the attribute information. Each client analysis uses the SAM query operators to access all necessary information. In this way, SAM becomes the framework that pulls together all the analyses and satisfies all the integration requirements as specified above. The extend of this thesis is limited only to the SAM design, implementation, interface operators, and the preprocessing steps of the REPAS project. The final chapter summarizes what is discussed in this thesis and details potential future work that will expand the robustness and functionality of the manager.

2 GENERALIZED ANALYSIS FRAMEWORK (a) — DESIGN OF ATTRIBUTE MANAGER

2.1 Qualification of Attributes

Important, yet common, attributes such as boundary and loading conditions can generally be characterized as tensorial in nature [18]. This viewpoint not only is the foundation of the design and implementation for the SCOREC Attribute Manager (SAM), but it also provides a means for mathematical consistency in describing attributes. Thus an attribute can be defined fully given the tensor order, the tensor symmetry, the distribution of each tensor component, and the coordinate system in which the tensor is defined. Therefore, a general structure for a full qualification of an attribute is shown in Figure 3. The subsequent sections of this chapter discuss in detail each of the components under **Attribute Physical Information**.

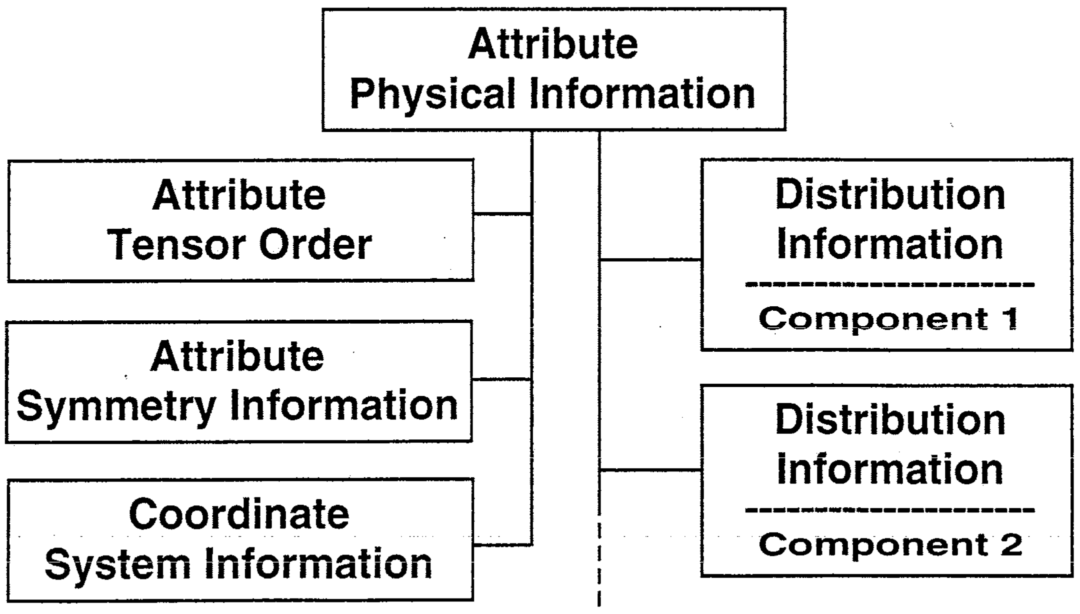


Figure 3 General structure of attribute physical information specification

2.1.1 Attribute Tensor Order

The order of the tensor, p, and the dimension of the coordinate system in which the tensor is defined, n, determines the maximum number of components, n^p allowable for the tensor. For example, a second-order tensor in 3–space has at most, $3^2 = 9$ independent components.

2.1.2 Symmetry

Two types of notational symmetries can be observed intrinsically in a tensor [6]: 1) A tensor is said to be symmetric with respect to a pair or groups of indices if the value of the tensor remains unchanged when the pair or groups of indices are interchanged. (e.g. $C_{ijkl} = C_{klij}$), and 2) A tensor is said to be antisymmetric (or skew-symmetric) with respect to a pair or groups of indices if the value of the tensor remains the same, but with a sign change, when the pair or groups of indices are interchanged (e.g. $C_{ijkl} = -C_{klij}$). A tensor that is symmetric with respect to any pair of indices is called a *totally symmetric tensor*, and likewise, a tensor that is antisymmetric with respect to any pair of indices is called a *totally antisymmetric tensor*. An example of a totally antisymmetric tensor of order n is the Ricci symbol (with n indices), namely,

$$\epsilon_{k_1 k_2 \ldots k_n} = \left\{ \begin{array}{l} 1, \text{ if } \begin{pmatrix} 1 & 2 \ldots n \\ k_1 & k_2 \ldots k_n \end{pmatrix} \text{ is an even permutation,} \\ -1, \text{ if } \begin{pmatrix} 1 & 2 \ldots n \\ k_1 & k_2 \ldots k_n \end{pmatrix} \text{ is an odd permutation,} \\ 0 \quad \text{otherwise.} \end{array} \right\} \qquad (1)$$

The number of components needed to characterize the tensor can be determined from these two types of symmetries. For example, the number of components needed to characterize the fourth-order stiffness material tensor reduces from 81 to 36 for basic minor symmetries ($C_{ijkl} = C_{jikl}$ and $C_{ijkl} = C_{ijlk}$) and it drops further to 21 for a major symmetry ($C_{ijkl} = C_{klij}$). In the case of solids, the stiffness tensor with these symmetries characterizes a generally anisotropic linearly elastic material; the stiffness tensor can be represented as the following array [25]:

$$C_{ijkl} = \begin{bmatrix} C_{1111} & C_{1122} & C_{1133} & C_{1123} & C_{1113} & C_{1112} \\ & C_{2222} & C_{2233} & C_{2223} & C_{2213} & C_{2212} \\ & & C_{3333} & C_{3323} & C_{3313} & C_{3312} \\ & & & C_{2323} & C_{2313} & C_{2312} \\ & SYM. & & & C_{1313} & C_{1312} \\ & & & & & C_{1212} \end{bmatrix} \tag{2}$$

Although these are the only two notational symmetries inherent in a tensor notation, another type of symmetry, what will be called here as spatial property symmetries, also affects the minimum number of independent components needed to specify a tensor. This spatial property symmetry can be viewed as symmetries of the spatial properties of that which the tensor describes with respect to the particular coordinate system in which the tensor is defined. For example, take the stiffness tensor mentioned above, the material that the tensor describes might be symmetric with respect to a plane in space, say the z-plane in the Cartesian coordinate system. That is, the strain energy remains unchanged when z is replaced by $-z$. Physically, this means that the elastic properties of the material are the same when viewed from either the position (x, y, z) or $(x, y, -z)$. As a consequence of the symmetry, the strain components are transformed as:

$$\bar{\varepsilon}_{ij} = \begin{bmatrix} \bar{\varepsilon}_{11} = \varepsilon_{11} & \bar{\varepsilon}_{12} = \varepsilon_{12} & \bar{\varepsilon}_{13} = -\varepsilon_{13} \\ & \bar{\varepsilon}_{22} = \varepsilon_{22} & \bar{\varepsilon}_{23} = -\varepsilon_{23} \\ SYM. & & \bar{\varepsilon}_{33} = \varepsilon_{33} \end{bmatrix} \tag{3}$$

The coefficients of the terms that change signs are forced to vanish to maintain the symmetry. That is, any coefficient having an odd number of 3's as indices must be zero, making the stiffness tensor:

$$C_{ijkl} = \begin{bmatrix} C_{1111} & C_{1122} & C_{1133} & 0 & 0 & C_{1112} \\ & C_{2222} & C_{2233} & 0 & 0 & C_{2212} \\ & & C_{3333} & 0 & 0 & C_{3312} \\ & & & C_{2323} & C_{2313} & 0 \\ & SYM. & & & C_{1313} & 0 \\ & & & & & C_{1212} \end{bmatrix} \tag{4}$$

The material this tensor describes is called a monotropic (or monoclinic) material. Furthermore, if the material is elastically symmetric with respect to the other two surfaces, all the stiffness terms with indices repeating an odd number of times must vanish. That is

$$C_{ijkl} = \begin{bmatrix} C_{1111} & C_{1122} & C_{1133} & 0 & 0 & 0 \\ & C_{2222} & C_{2233} & 0 & 0 & 0 \\ & & C_{3333} & 0 & 0 & 0 \\ & & & C_{2323} & 0 & 0 \\ & SYM. & & & C_{1313} & 0 \\ & & & & & C_{1212} \end{bmatrix} \tag{5}$$

The material this tensor describes is called an orthotropic material. If the material is symmetric with respect to a line, say, the z–axis and to the z surface, then it can be proved that the index numbers 1 and 2 can be interchanged, that is,

$$C_{1111} = C_{2222} \qquad C_{1133} = C_{2233} \qquad C_{1313} = C_{2323}$$

In addition,

$$C_{1212} = 0.5(C_{1111} - C_{1122})$$

The stiffness tensor can then be characterized by five independent values:

$$C_{ijkl} = \begin{bmatrix} C_{1111} & C_{1122} & C_{1133} & 0 & 0 & 0 \\ & C_{1111} & C_{1133} & 0 & 0 & 0 \\ & & C_{3333} & 0 & 0 & 0 \\ & & & C_{1313} & 0 & 0 \\ & SYM. & & & C_{1313} & 0 \\ & & & & & \frac{1}{2}(C_{1111} - C_{1122}) \end{bmatrix} \tag{6}$$

The material this tensor describes is called a transversely isotropic (or hexagonally symmetric) material. For a material that has no preferred direction, that is, one having a point symmetry, the roles of the indices 1, 2, and 3 can be fully interchanged, allowing further simplifications to be made. Namely,

$$C_{1111} = C_{2222} = C_{3333} \qquad C_{1122} = C_{1133} = C_{2233}$$

$$C_{1212} = C_{1313} = C_{2323}$$

In light of the above discussion on symmetries, whatever data structure and scheme used for the symmetry must allow for the storage and differentiation of these different types of symmetry. The types of symmetry are: symmetric, totally symmetric, antisymmetric, totally antisymmetric, plane, line, point, and plane-line [6]. This list, though not exhaustive, is sufficient for most applications. In addition to the symmetry types found in the tensor and the specific symmetries, any pertinent information about each symmetry must also be stored. For example, if a symmetry is with respect to a certain axis in a certain angle, this information must be reflected in the data stored.

2.1.3 Distribution

The value and "direction" of a tensor may vary as functions of space, time, and other variables. Moreover, each component of the tensor may have different dependencies. The Attribute Manager considers each of the dependencies a distribution. Take the case of the pressure load as shown in Figure 4, for example. The load in the radial direction varies linearly around the cylinder, while the axial and tangential components of the pressure load are identically zero. One issue becomes immediately apparent is that one must be able to specify a distinct distribution for each of the tensor components. Often times, these distributions need to be expressed with various independent variables, such as spatial and time variables, and in the form of some well defined functions, such as sine and cosine. The attribute manager needs to be able to handle these types of dependencies.

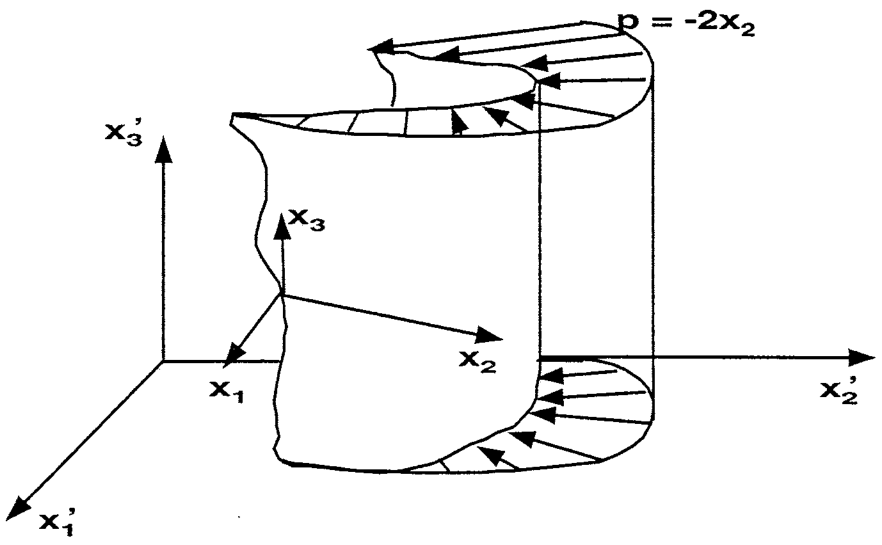


Figure 4 Example of pressure load

Also, the attribute manager needs to have the capability to evaluate the distributions in any given coordinate system.

In order to do any manipulation and/or evaluation of the distribution, the attribute manager needs to understand fully the properties of each parameter of the equation. In this discussion, a *parameter* is taken to be the smallest unit that an equation may be decomposed. For example, "*3*", "+", and "x" are the parameters of the equation, $y = 3 + x$, whereas "*7*" is the sole parameter of the equation, $k = 7$. In light of the above discussion, a schema for the distribution information that satisfies these requirements is shown in Figure 5.

Each distribution has information regarding the parameter relationship, the individual parameters, and their respective dependencies. The relationship between these parameters is captured in the string *parameter relationship* and stored in a parsed binary tree array. For example, Figure 6 shows the tree resulting from a distribution of $3sin(2\theta - \phi)$. The distribution is stored in a reversed-Polish format for easy evaluation. The content of the tree node is shown in the first column of the table entry and is considered to be a

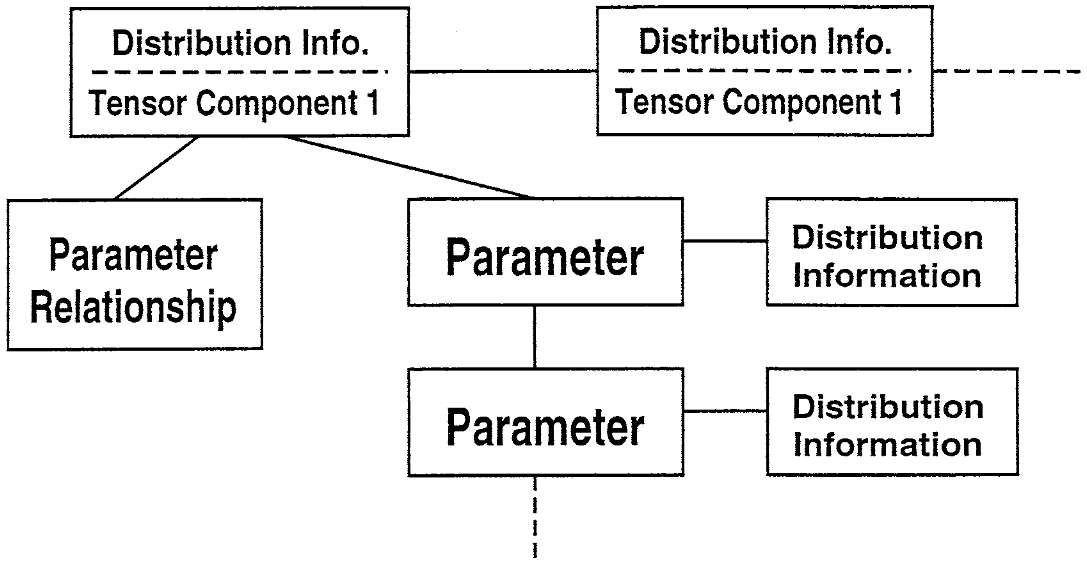
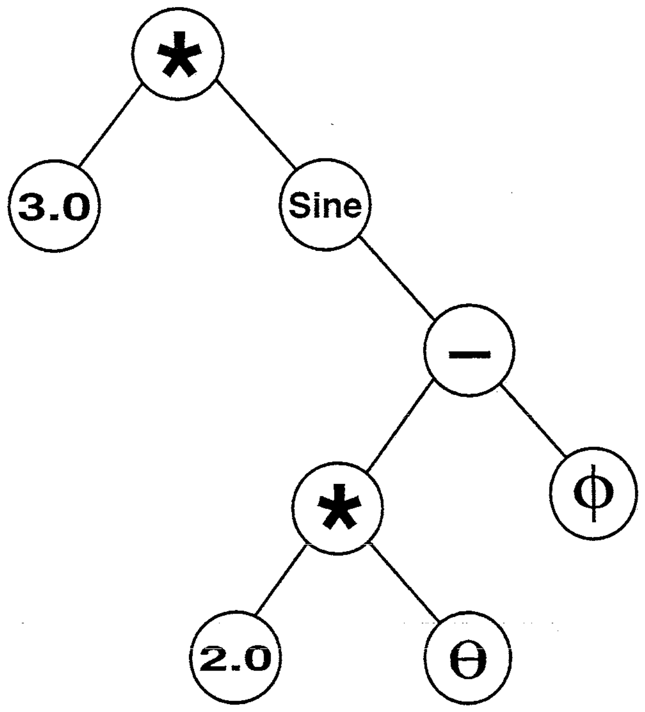


Figure 5 Data structure for distribution information

parameter. The position of the left child of the tree node is shown in the second column of the table entry, whereas the position of the right child is shown in the respective third column. For the example tree, the last row of the table indicates that the operator,



	tree node	left child	right child
1	3.0	0	0
2	2.0	0	0
3	θ	0	0
4	*	2	3
5	φ	0	0
6	–	4	5
7	sine	0	6
8	*	1	7
9			

Figure 6 Example of a parsed binary tree

"*," is operating on the two elements stored in rows 1 and 7, whose nodal contents are, respectively, "3" and "sine". This corresponds to the top three nodes of the above binary tree. The two zeros (nodal leaves) in row one of the table indicate that this node, "3", is an operand, not an operator. Next, row seven indicates that "sine" is an operator operating on the nodal content stored in row six, whose content is "-". The rest of the table entries are corresponded in the same way. It is apparent that the tree nodes stored in rows 1, 2, 3, and 5 (with respective contents of "3", "2", "θ", and "ϕ") are operands, and the tree nodes stored in rows 4, 6, 7, and 8 (with respective contents of "*", "-", "sine", and "*") are operators operating on their respective operands.

Allowing for maximum flexibility, seven types of parameters are used in defining a distribution: numbers, basic operators, constants, built-in functions, variables, attribute components, and distributions. In the example shown in Figure 6, parameters "3" and "2" are numbers, whereas parameters "*" and "-" are basic operators. The remaining types of parameters are:

- Constants: fixed numerical values such as π and e;
- Built-in functions: selected functions from the system math library and any user defined functions such as sine and cosine functions;
- Variables: any undefined variables in which the user needs to specify the value at the time of evaluation, such as any spatial variables of a particular coordinate system;
- Attribute components: the value of a given attribute component at a specified evaluation point; and
- Distributions: the value of a given distribution at a specified evaluation point.

This distribution structure allows for the possibility of a distribution of a distribution, of a distribution, and so on.

Thus far, only those distributions that are numerical equations are considered. Another common form of dependencies also need to be considered. For multiple analyses operations, the results of one analysis may be passed as input into another analysis. For example, the displacement calculated of a body from a global analysis might be used as boundary condition of a local analysis for a critical area. The local analysis might require displacements at discrete points along the boundary of the body. The scheme discussed above allows for this displacement to be defined as a distribution, where this displacement may be defined as a parameter of user defined function (Built-in function). At the time of evaluation for this distribution, a spatial position is entered as input to the user function. The function evaluates the displacement at the given point (for example, from a results table or an equation) and returns to the evaluator the resulting displacement. Thus, this scheme greatly increases the robustness of the distribution definition and evaluation.

2.1.4 Coordinate System

A set of linearly independent vectors spanning a given space is called a basis for this space [1]. Given a set of basis vectors spanning a 3–D space, for example $\{\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3\}$, with origin at O, any vector $\overline{\mathrm{OP}}$ from O to point P in this space can be expressed as a linear combination of this basis, $\overline{\mathrm{OP}} = a\mathbf{v}_1 + b\mathbf{v}_2 + c\mathbf{v}_3$. The coefficients of this linear combination, expressed as a coordinate matrix $[a,\ b,\ c]^T$ can be thought of as the coordinates of point P relative to the coordinate system defined by this set of basis vectors. Thus a change of coordinate system implies a change in the basis vectors. For example, if for a vector space an old coordinate system (basis) [B] is changed to the new coordinate system (basis) [B′], with

$$B = \{\mathbf{u}_1, \mathbf{u}_2\} \qquad \text{and} \qquad B' = \{\mathbf{u'}_1, \mathbf{u'}_2\}$$

which are related by:

$$\mathbf{u'}_1 = a\mathbf{u}_1 + b\mathbf{u}_2$$

$$\mathbf{u'}_2 = c\mathbf{u}_1 + d\mathbf{u}_2$$

Given a vector **v** defined with respect to the new basis as

$$\mathbf{v} = k_1\mathbf{u'}_1 + k_2\mathbf{u'}_2$$

or written as a coordinate matrix

$$[\mathrm{v}]_{B'} = \begin{bmatrix} k_1 \\ k_2 \end{bmatrix} \tag{7}$$

The coordinate matrix $[\mathbf{v}]_{B'}$ is then related to the new basis B by

$$[\mathrm{v}]_{B} = \begin{bmatrix} a & c \\ b & d \end{bmatrix} \begin{bmatrix} k_1 \\ k_2 \end{bmatrix} \tag{8}$$

or

$$[\mathrm{v}]_{B} = \begin{bmatrix} a & c \\ b & d \end{bmatrix} [\mathrm{v}]_{B'} \tag{9}$$

The basis vectors do not have to be orthogonal so long as they span the space in which the coordinate system is defined. The matrix

$$P = \begin{bmatrix} a & c \\ b & d \end{bmatrix} \tag{10}$$

is called a transition matrix and is invertible. It transforms the coordinates from one coordinate system to another. Notice the transition matrix can be obtained easily once the relation between the old basis (coordinate system) and the new basis (coordinate system) is defined. As an example, when a set of coordinates in 2–space are transformed by rotation to another coordinate system by an angle θ as shown in Figure 7, the resulting transition matrix is

$$P = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{11}$$

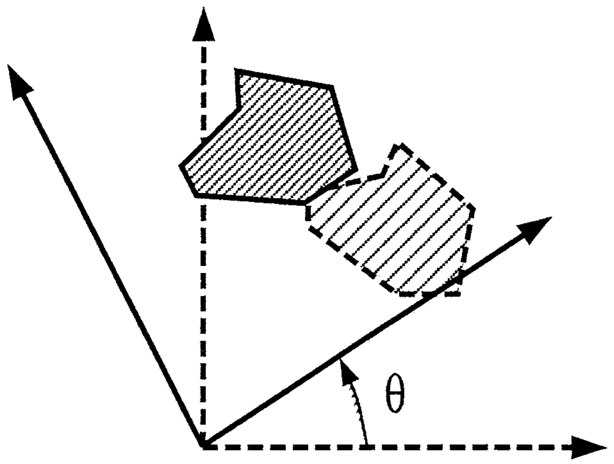


Figure 7 Example of Transformation of Coordinate System by Rotation

This discussion can be generalized to curvilinear coordinate systems [26]. The transformation matrix becomes more complex because the new basis can no longer be written as a linear combination of the old basis. An example of a common curvilinear system is the spherical coordinate system as shown in Figure 8, which can be written with respect to the Cartesian coordinate system as

$$r = \sqrt{x^2 + y^2 + z^2} \tag{12}$$

$$\phi = \arccos\left(\frac{z}{\sqrt{x^2 + y^2 + z^2}}\right) \tag{13}$$

$$\theta = \arcsin\left(\frac{y}{\sqrt{x^2 + y^2}}\right) \tag{14}$$

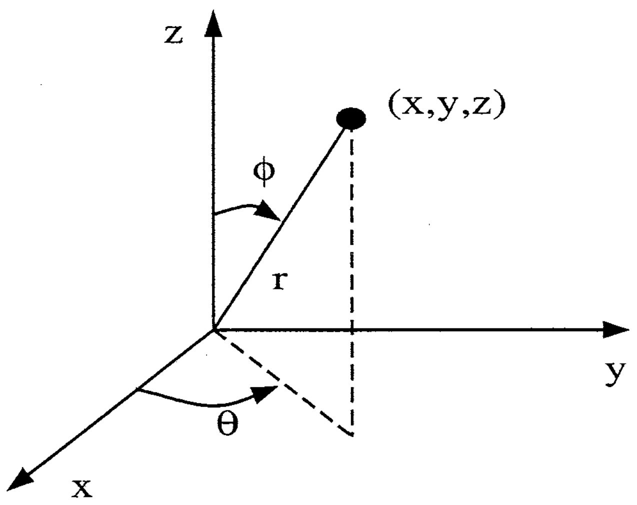


Figure 8 Spherical Coordinate System

Let x, y, z be the coordinates of the new coordinate system, and let the relationship between the coordinates to the x', y', z' of the old coordinate system be described by

$$x = f(x', y', z') \qquad y = g(x', y', z') \qquad z = h(x', y', z')$$

These functions, f, g, h, must have continuous first derivatives in some domain D of the xyz-space, and at point P_0:(x'_0, y'_0, z'_0) in D the Jacobian of the functions f, g, h is written as

$$\frac{\partial(x,y,z)}{\partial(x',y',z')} = \begin{vmatrix} \frac{\partial f}{\partial x'} & \frac{\partial f}{\partial y'} & \frac{\partial f}{\partial z'} \\ \frac{\partial g}{\partial x'} & \frac{\partial g}{\partial y'} & \frac{\partial g}{\partial z'} \\ \frac{\partial h}{\partial x'} & \frac{\partial h}{\partial y'} & \frac{\partial h}{\partial z'} \end{vmatrix} \neq 0 \qquad (15)$$

This insures that at point P_0 in D it is possible to determine each point (x', y', z') in terms of the coordinates x, y, z. This means that there are inverse functions that map the coordinates in the new coordinate system back to coordinates in the old one. In other words there exist functions f', g', and h' such that

$$x' = f'(x, y, z) \qquad y' = g'(x, y, z) \qquad z' = h'(x, y, z)$$

Accordingly, the position vector of a point in terms of its curvilinear coordinates is now given by

$$\mathbf{R}(x, y, z) = f'(x, y, z)\,\mathbf{i} + g'(x, y, z)\,\mathbf{j} + h'(x, y, z)\,\mathbf{k}$$

At each point where the Jacobian

$$\frac{\partial(x', y', z')}{\partial(x, y, z)} = \frac{\partial \mathbf{R}}{\partial x} \cdot \frac{\partial \mathbf{R}}{\partial y} \times \frac{\partial \mathbf{R}}{\partial z} \qquad (16)$$

is non-zero, the three vectors $\frac{\partial \mathbf{R}}{\partial x}, \frac{\partial \mathbf{R}}{\partial y}, \frac{\partial \mathbf{R}}{\partial z}$ are tangents to the coordinate curves. They are linearly independent and hence form a basis for this space defined in this curvilinear coordinate system [8][26].

From the above discussion, one can see that the specification of one coordinate system with respect to another coordinate system requires three pieces of information: 1) a reference that identifies the base coordinate system on which the current coordinate system is defined, 2) the coordinates of the origin of the current coordinate system in the space of the base coordinate system, and 3) the functions (three for a coordinate system in 3–space) defining the relationship between the coordinates in the current coordinate system and the base coordinate system.

2.2 Organization of Attributes

SAM is designed to be a generalized attribute manager that is application independent. This means that the organization structure needs to be constructed for robust manipulation of attributes for a wide variety of analysis applications that also accommodates for multiple load cases. Additionally, it needs to provide a structure for the sharing of common analysis and domain information, and it should be one that properly reflects the relationships of attributes within an analysis and between analyses.

Consider an application consisting of multiple analysis cases. The application might analyze the thermal, electrical, and thermal-mechanical behaviors of an electrical component. Though the three analyses are quite different, the three analysis domains will have an overlap of information. The material properties are the same for all three domains, but the parameters of interest may vary. While the boundary conditions and loading conditions might be very different among the three analyses, both the thermal and thermal-mechanical analyses might use the same geometric description of the domain. The proper way to specify such data is to avoid information duplication. All the common information should be shared between the three analyses. Furthermore, all three analyses might have multiple load cases to be analyzed. The thermal analysis might study various load distributions to determine the optimal design of the component. Then the thermal-mechanical analysis might study the component loaded with the resulting load distribution in various strengths. In the latter case, the distributions do not need to be duplicated for each load case — only multiplication factors need to be specified. In addition to the ability of the organization structure to share information, it must also need to have a way to describe the relationship between attributes. For example, results from the thermal analysis might need to be used as input for the thermal mechanical analysis. Thus the organization structure needs to provide a means to describe the "results-input" relationship between analyses.

The design of the organization structure provides a flexible framework to deal effectively with the many aforementioned requirements according to their functional needs. The organization structure employs a hierarchy of different analysis cases that are assembled into groups of specific sets of attributes. It provides a means to organize the many attribute sets in a manner that is appropriate to the analysis at hand. Some terminologies used to define this structure are as follows:

1. **Attribute Type**: The type of the physical information of the attribute, such as displacements, body forces, traction loads, and material properties
2. **Attribute**: A single attribute specification, such as thermal conduction coefficients or a prescribed displacement boundary condition
3. **Set**: A combination of attributes of a given type
4. **Group**: A combination of sets and/or attributes in a meaningful manner, such as sets of body forces and tractions that make up a particular load case
5. **Case**: A combination of groups, sets, and/or attributes representing a complete specification for an analysis, such as the global heat conduction of the REPAS project
6. **Collection**: All the attributes associated with a given function to which the attributes are being applied in a modeling system. The collection of attributes that this section had been focused on are mainly analysis attributes. Other important collections are idealization attributes and numerical model attributes, both of which will be discussed

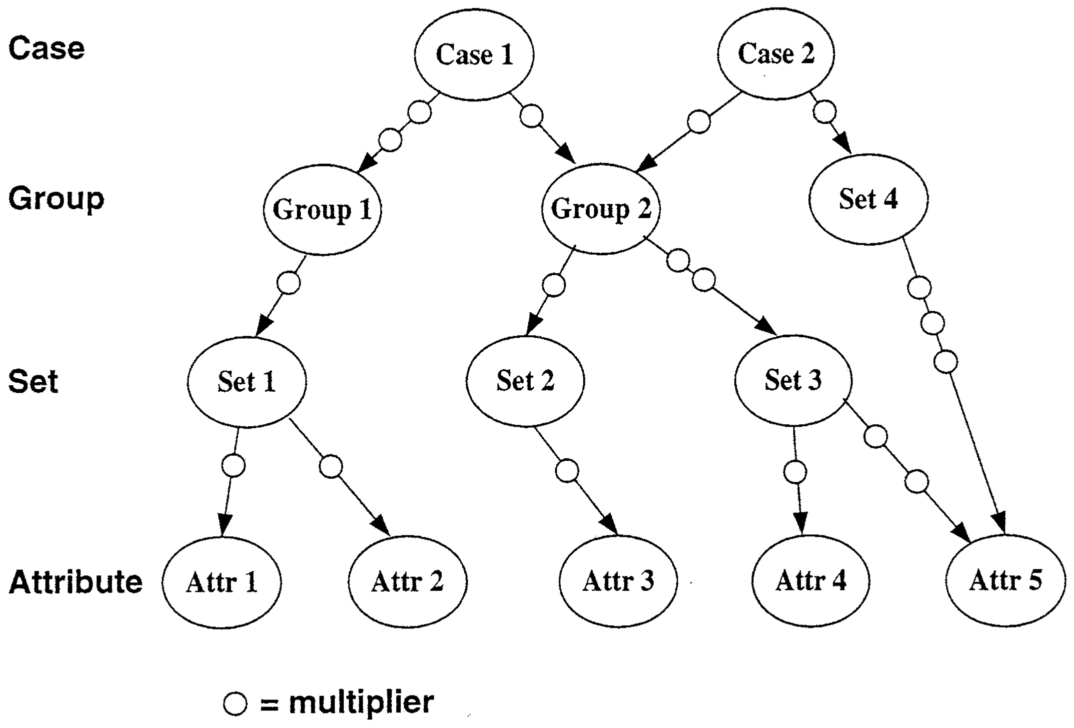


Figure 9 Overall Framework of Organization Structure

in later sections of this paper.

7. **Multiplier**: Factor used to scale any of the organization hierarchies under a case

The overall framework is illustrated in Figure 9. Several features of this organization structure is worthy of mention here. Given an analysis case, all the attributes belonging to this case are known. The picture shows that the node at the tail of the arrow knows only the node to which the head of the arrow points. Multiple multipliers can be specified between any hierarchical levels. The value of the attribute of a given set in a given group under a given case will be modified with all the multipliers specified along the path. The hierarchical associations are also flexible in that, depending on what makes sense for the analysis, a case can directly be associated with a set or an attribute — thus "by-passing" the group. Likewise, a group can be directly associated with an attribute, "by-passing" the set.

2.3 Association of Attributes

The third major requirement of the attribute manager is the ability to associate given attributes to the correct analysis model entities. Generally speaking, an analysis model is a description of the domain being analyzed by an application. Some examples of an analysis model include the physical, geometric, idealized, and numerical models. One could generalize from the discussion of Shephard and Finnigan [16] the relationship between data sets for finite element modeling to that of general modeling needs. This generalized relationship is depicted in Figure 10. Each of the boxes represents a distinct

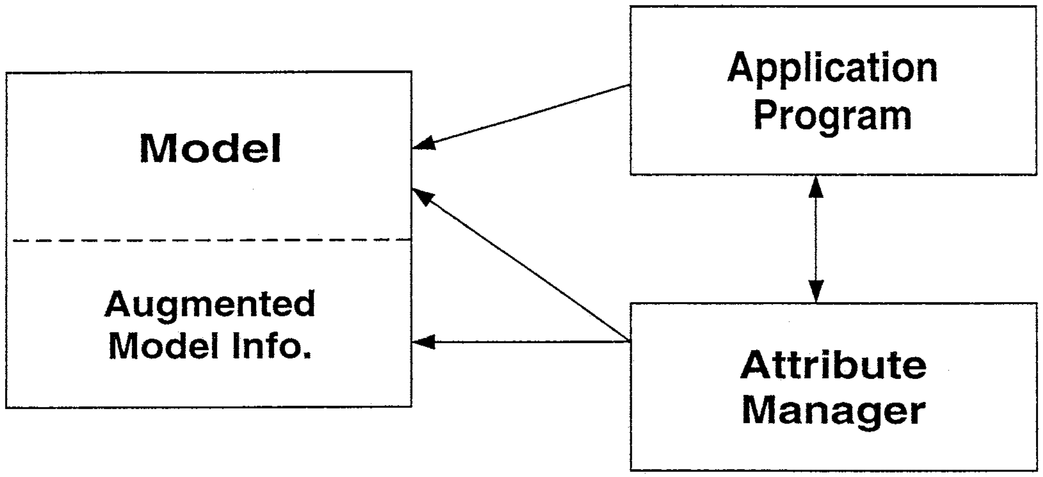


Figure 10 Relationship Among Modeling Modules

set of modules. The module at the tail of the arrow queries or gives commands to the module to which the arrow is pointing. Consider the interaction of these modules in the REPAS application. Attributes are attached to four different models: a physical model for general information queries, an idealized model for the global analyses, a geometric model for the local stress analysis, and a numerical model for the local thermal analysis. (The first two "models" will be more carefully defined and discussed in later chapters; the descriptions of the latter two "models" maybe found in the REPAS — User's Manual

[4][5]. Given an analysis model, one application (such as the preprocessing module of the REPAS project) might invoke the attribute manager to create attributes, organize them appropriately and attach the attributes to the model entities for a particular run, while another application (such as the local stress analysis) might use the model and attribute information to drive its analysis procedure. In these two instances, the "Model" in Figure 10 represents two entirely different types of models. Similarly, the "Application Program" module for the REPAS project includes the two global analysis procedures, the local heat conduction analysis procedure, and all the supporting modules of the local thermal elastic stress analysis. Clearly, the Attribute Manager must be able to handle a wide variety of modeling specifications, querying needs, and applications.

To aid in the discussion, four terms need to be identified and differentiated:

1. **Original Model**: The very first model before any modifications to the model are performed; this original model is never modified and is the model from which other models are derived.
2. **Idealized Model**: A model that has gone through one or more idealizations of the original model. For example, an idealization can be a simplification of an existing model, such as removal of a model feature irrelevant to the analysis at hand or dimension reduction of a model. An idealized model may also be derived from another idealized model.
3. **Auxiliary Model Entities**: Additional model entities not defined in the idealized model but needed for the definition of attributes. The auxiliary model entities are of the same types of model entities as that of the idealized model, but the auxiliary model entities have no direct geometric or topological associations to the idealized model.

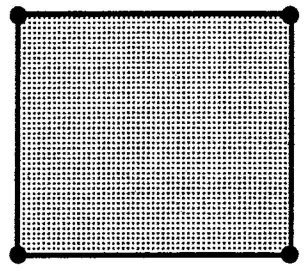
4. **Augmented Model**: Model created by a combination of the idealized model and relevant auxiliary model entities associated with the attributes of the analysis case.

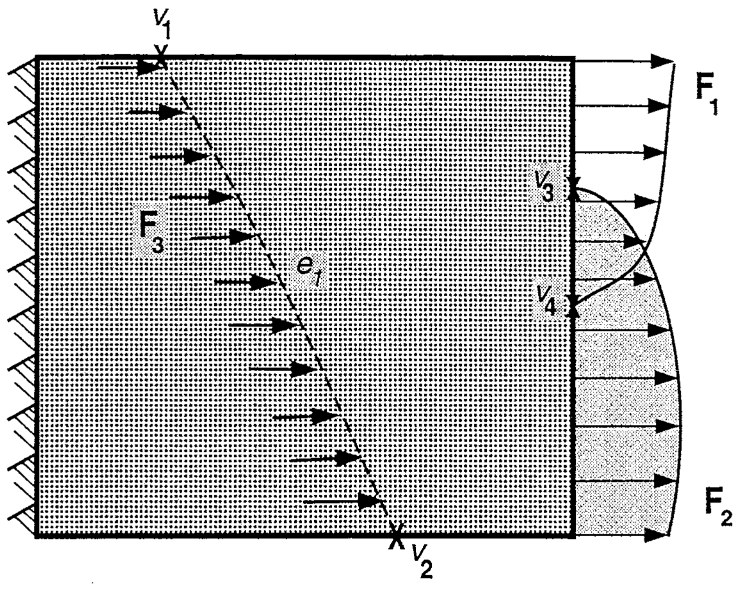
This section focuses on the interactions between the attribute manager module and the model and augmented model information modules. A large part of the discussion focuses on the specification and association of the augmented model information. Finally, a scheme emerging from the discussion to drive the augmentation process is discussed.

2.3.1 Model Interactions

A geometric model is a mathematical domain of an object in physical space, defined by a collection of geometric entities consisting of points, curves, surfaces, and volumes and topological entities of vertices, edges, faces, and topological regions. The geometric entities are needed to define the shape of the domain, whereas the topological entities are needed to describe the relationship between the geometric entities. The present discussion is focused on only the geometric model, with the discussion limited further to the interactions between the idealized model, auxiliary model entities, and the augmented model. A simple example shows some of the complexities of these interactions. For the analysis case shown in Figure 11, one can consider the augmentations of the idealized model [16] required to properly account for the attributes.

The idealized model is a simple block with a total of four vertices and four edges, as shown in Figure 11a. A set of loads and boundary conditions is imposed on the block, as shown in Figure 11b. In order to reflect properly the load attributes on the finite element mesh (with nodes aligning with points v_1, v_2, v_3, v_4, and finite element edges aligning with internal edge e_1), the mesh generator that constructs the mesh needs to see these geometric entities at their respective locations. This means that the side


a. Idealized Model



b. Imposed Loads and Boundary Conditions

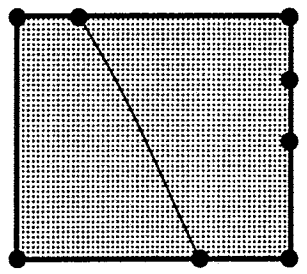

c. Augmented Model

Figure 11 Example Analysis Domain

edge on the right needs to be split into three edges, with vertices at points v_3, and v_4 (compare Figure 11a and Figure 11c). Likewise, the rectangular face needs to be split at the junction marked by edge e_1, creating two faces with an edge at the same place and two bounding vertices at points v_1 and v_2. Associated topologies also need to be modified to reflect the correct augmentations. The vertices (v_1, v_2, v_3, and v_4) and edges (e_1 and the seven edges resulting from the splits created by the four vertices) created as a result of the need for accurate representation of the loading conditions are auxiliary geometries that were not in the idealized model. The resulting augmented model, then, is shown in Figure 11c. As one can see, the nature of the analysis attributes necessitates the creation of auxiliary geometries to insure correct analysis results. Consequently, a set of modeling operators need to be made available from the geometric modeler to the attribute manager so that these augmentation operations can be performed. A list of the needed operators are defined and listed in Appendix A. These augmentation operators are separated into categories of 1) definition of geometric and topological entities, 2) interrogation of any existing geometric and topological data base, and 3) manipulation of geometric and topological entities.

Implicit in the specification of the problem is that more than one attribute can be specified on the same model entity. The reverse is also true — more than one geometric entity may have the same attribute definition. Of course, this last point is contingent upon the correctness of specifying such attribute on both entities. The requirements of having the ability to check for attribute definition correctness and even automatically inherit appropriate attributes for adjacent lower order entities require much more intelligence from the attribute manager than is currently implemented.

2.3.2 Augmentation Scheme

Once the relevant auxiliary model entities are created and the attributes are specified, the proper association of the auxiliary entities poses an additional level of complexity. Consider dividing the analysis shown in Figure 11 into two analysis cases — both using the same geometric domain: one with only p_1 applied on the right side edge and the second with only p_2 applied on the same edge. If all the auxiliary geometries (defined for all the analyses) are augmented to the idealized model indiscriminately, the discretization based on the resulting augmented model will be very inefficient. For example, for the analysis case with only p_1 applied, the extra vertex on the right side edge created by p_2 would have imposed unnecessary constraints on the finite element mesh for something that has nothing to do with the problem at hand. This is only a simple example; a typical suite of analysis cases might have many more sets of attribute with complicated overlapping of attributes and auxiliary geometries.

A solution to this problem is to have the attributes for the analysis case drive the augmentation of the model. In this scheme, the auxiliary geometries are selectively augmented to the idealized model at the start of an analysis run. The attributes are organized according to the analysis needs and are consistent with the organizational rules as described in section 2.2. During the setup for an analysis case, augmentation operators are called to choose only those auxiliary geometries attached to the attributes that are relevant to that analysis. The idealized model is then augmented to reflect the condition of the analysis. A different augmented model may be created for another analysis case that has a different set of attributes. In this way, the augmented model always reflects the current analysis, without redundancies.

This scheme introduces two important issues that need to be addressed: 1) the augmentation process must have rules that clearly keep track of the attributes and associated auxiliary geometries from creation through all the inheritance and evolution of the augmented models, and 2) rules and operators need to be developed to handle the augmentation logistics of what information to retain, what to delete, and how to switch from one augmentation level to another. Although these features have not been implemented yet in SAM, both issues will be discussed briefly in the following sections.

2.3.2.1 Model-Attribute Relations and Associations

To facilitate the discussion, the relationship between the model and the attributes need to be defined and clarified. The notation convention used to specify the model entities is:

${}_{Y}T^{d}_{i}$	Topological entity i of dimension d from model Y, where $d = 0, 1, 2, 3$, corresponding to a vertex, edge, face, and region respectively and $Y = I, G, A$, corresponding to Idealized, Geometric, and Augmented models respectively.

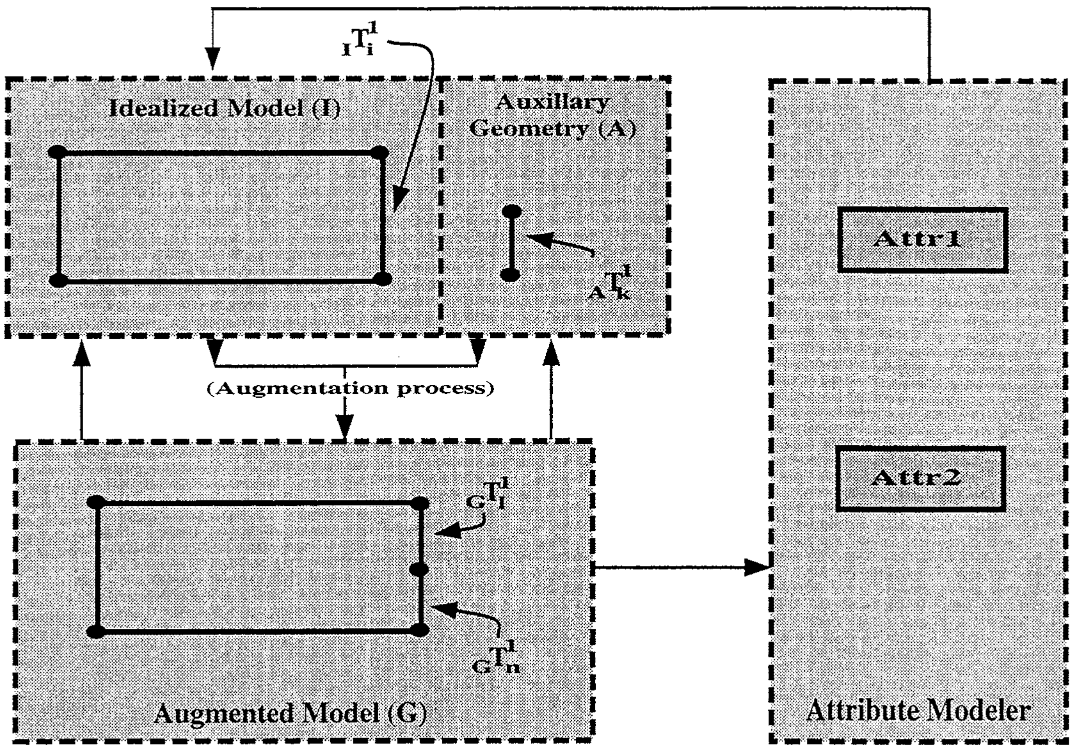


Figure 12 Relationship between the idealized model, auxiliary geometries, augmented model, and the attribute manager

The overall picture of the approach taken to relate the idealized model, the auxiliary geometries, the augmented model, and the attribute manager can be diagrammed as in Figure 12. The arrows shown between the boxed modules depict who has a knowledge of whom. For the given example in Figure 12, the only relation before the augmentation process is:

$$\mathrm{Attr1} \rightarrow {}_{\mathrm{I}}\mathrm{T}_{\mathrm{i}}^{1} \qquad \mathrm{Attr2} \rightarrow {}_{\mathrm{A}}\mathrm{T}_{\mathrm{k}}^{1} \tag{17}$$

where the first expression reads: *attribute 1 is being attached to the idealized (I) model entity i of the first order (i.e., a model edge).* The second expression of equation 17 reads: *attribute 2 is being attached to the k^{th} auxiliary edge, ie, auxiliary (A) geometric entity of the first order.* Before augmentation, the analysis attributes are attached to only the idealized model and auxiliary model entities. The relations after the augmentation process are:

$$\mathrm{Attr1} \rightarrow {}_{I}T_{i}^{1} \qquad \mathrm{Attr2} \rightarrow {}_{A}T_{k}^{1} \tag{17}$$

$$ {}_{G}T_{l}^{1} \leftrightarrow {}_{I}T_{i}^{1} \qquad {}_{G}T_{n}^{1} \leftrightarrow {}_{I}T_{i}^{1}, {}_{A}T_{k}^{1} \tag{18}$$

$$ {}_{G}T_{l}^{1} \rightarrow \mathrm{Attr1} \qquad {}_{G}T_{n}^{1} \rightarrow \mathrm{Attr1}, \mathrm{Attr2} \tag{19}$$

As discussed in the previous section, the attributes defined for an analysis case drive the analysis (and the model augmentation). From the knowledge captured in the attributes, the relevant auxiliary geometries are integrated with the idealized model — creating the augmented model along with a new set of relationships. A two-way mapping is created between the augmented model entities and their parent idealized and auxiliary geometric entities (equation 18). In this way, the attribute manager only works with the idealized model (and, if applicable, the auxiliary geometries), as expressed in equation 17. Any inquiry of the augmented model is through the forward mapping from the idealized model (and auxiliary geometries) to the augmented model. The backward mapping from the augmented model entities to the idealized model and auxiliary model entities is essential for applications that need to reassociate the augmented model domain information to the idealized model domain — for example, reassociating a mesh back to the idealized model for analysis. To simplify searching, the augmented model entities inherit the knowledge of the associated attributes (equation 19).

Now consider the example shown in Figure 13. The auxiliary geometries shown in the right of Figure 13 are needed to specify the distributions p_1, p_2, and p_3. The resulting augmented model is shown in the box on the foreground, with the right edge split at points b, c, and d. For the sake of discussion, let the augmentation order be p_1,

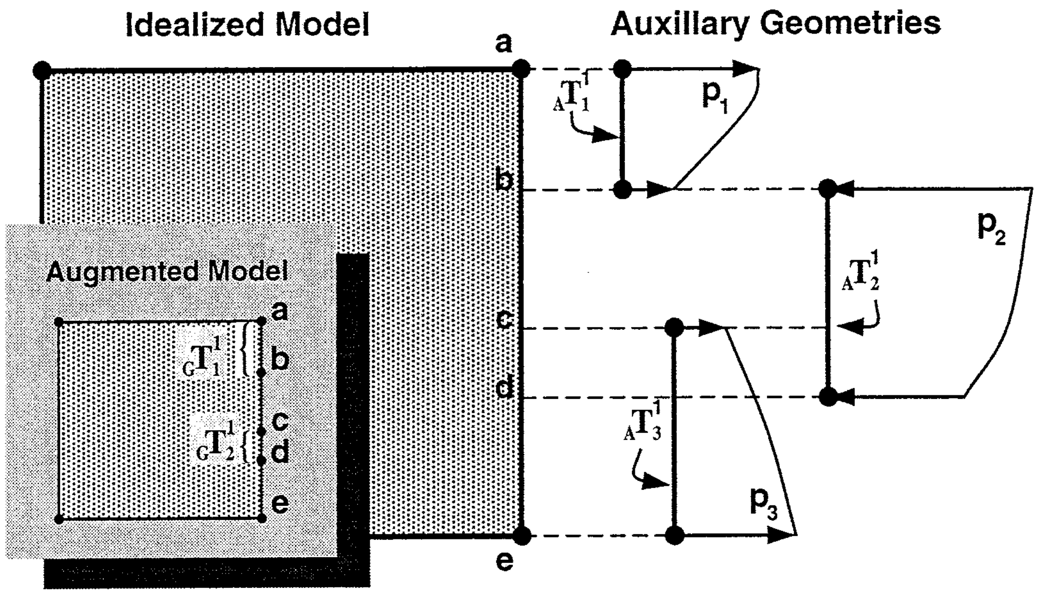


Figure 13 Example of Augmentation Process

p_2, and p_3. The augmentation of p_1 will cause the right edge of the idealized model to split at point b. After the augmentation, the attribute specifying p_1 will be reassociated (or propagated) to the newly created edge bounded by points a and b. Notice in this first augmentation, the auxiliary geometry is not modified. Rather, it is used to augment the idealized model, and the attribute attached to the auxiliary edge, $_AT_1^1$, is propagated to augmented edge, $_GT_1^1$.

Next, the auxiliary geometry associated with p_2 is augmented to the partially augmented model. This second auxiliary geometry will cause the right edge of the partially augmented model to split at point d. As with the first auxiliary geometry, the attribute is propagated from $_AT_2^1$ to the newly created edge (bounded by points b and d) of the partially augmented model.

Next, the auxiliary geometry associated with p_3 is augmented to the partially augmented model. This third auxiliary geometry will cause the right edge of the partially augmented model to split at point c, which also splits the edge (bounded by points b and d) created from the previous augmentation. In this third case, edge ${}_{G}T^{1}_{2}$ bounded by points c and d has association to two attributes: the attribute associated with ${}_{A}T^{1}_{2}$ and another associated with ${}_{A}T^{1}_{3}$. Since these two attributes have different distributions, the attribute manager must provide operator(s) to consolidate multiple distributions from overlapping attributes (as with the case of ${}_{G}T^{1}_{2}$ where portions of p_2 and p_3 overlap). Moreover, because in this scheme the augmented model has a knowledge of the applied attributes, the geometric modeler must have the facility to store multiple references to the attribute manager. In most CAD modelers, this might come in the form of entity attributes that are specific to the modeler. Otherwise, the attribute manager must provide another operator to store the attribute association of modeler entities.

From the above demonstration of the augmentation steps, it is clear that a set of rules must be developed to keep track of the attributes and their associations with partially augmented entities throughout the evolution of the augmented model. Furthermore, appropriate attribute manager operators to handle consolidation of attribute distributions must be developed.

2.3.2.2 Model-Attribute Augmentation Information

The augmentation process as discussed in the previous section reveals many potential complexities. A particularly important issue is in the augmentation logistics of the retention and deletion of augmented modelling information. From the definitions presented in section 2.3 and the scheme discussed in section 2.3.2, a major working requirement is that the augmentation process is driven by attributes. Strictly speaking, these attributes are defined on idealized and auxiliary entities, and an augmented model is the result of the process. This definition is not very flexible for processes that require multiple augmentations, such as in metal-forming analyses in which the geometry evolves through time — thus requiring multiple levels of augmentation. To resolve this issue, an augmented model can be considered a pseudo-idealized model opened for another augmentation. Restricting the augmentation to only one level simplifies the implementation logic to only one level. On the other hand, a set of logistics needs to be developed to retain the pseudo-idealized model for further augmentations, to transfer (and create) the appropriate attributes for the second augmentation, and to transfer (and create) appropriate auxiliary entities for the new analysis. This set of logistic can then be recursively applied for any subsequent augmentations.

A scheme that should aid the logistics of multiple augmentations is that all the information for one augmentation be isolated and put into a file, which consists of a *packet* (or grouping) of auxiliary entities. The attributes also will have a flag with an identifier specifying to which auxiliary they belong. In analyses that involve multiple interdependent idealized models, all the packets need to be retained for moving back and forth between idealized models.

The entire augmentation process can be summarized thus: given an idealized model, it is enhanced with a set of attributes specific to the particular analysis to be run. To retain consistency, auxiliary entities must be created for those attributes that act on only portions of the model entities that are not explicitly defined (such as load on a portion of a face or a projected wind load). These auxiliary geometries are then augmented to the idealized model using geometry manipulation operators. The resulting augmented model can then be discretized for analysis. The augmented model entities will know of the attributes attached to the entities of the idealized model and auxiliary geometries.

3 GENERALIZED ANALYSIS FRAMEWORK (b) — IMPLEMENTATION OF ATTRIBUTE MANAGER DESIGN

Implementation of the attribute manager is performed in three stages: 1) basic data structure and functionalities, 2) interface operators, and 3) higher level functionalities. The data structure is defined based on the needs and issues as discussed in the previous chapter on the design of the attribute manager. Basic functionalities to support the attribute manager capabilities are also developed. As such, this chapter focuses on only the first of the three implementation stages. Interface operators that allow for general usage of the attribute manager are discussed in the next chapter. Enhancements and higher level functionalities are outlined as future work in the last chapter. The basic structure of this chapter follows that of the previous chapter. The first section focuses on the schemes used to capture the required physical information of an attribute. The second section details the implementational specifics of the organizational aspects of the attribute manager. Finally, the implementation for the relational aspects are discussed.

3.1 Physical Information Implementation of Attributes

An attribute is fully qualified by the collection of physical information as described in the first section of the previous chapter. Corresponding to section 2.1 and visualized in Figure 3, this section discusses the implementational details of each piece of the physical information, which includes the tensor order of the attribute, attribute symmetry information, coordinate system information, and the tensor component distributions. The corresponding data structure for the physical information of an attribute is shown in

```
/* Define the fields containing the physical information of the attribute */
 struct PHYSICAL_INFO {
   long attrib_tensor_order;
    /* a flag specifying the order of the attribute tensor */
  COORDINATE_SYS *attrib_coord_sys_ptr;
    /* pointer to the local coordinate system information of the attribute */
  SYMMETRY *attrib_tensor_symmetry_ptr;
    /* pointer to the tensor symmetry information */

  struct comp_distrib {
    struct comp_distrib *next;
      /* pointer to next in linked list */
    DISTRIBUTION *distrib_info_ptr;
      /* pointer to the distribution information of the tensor components */
  } *cdist_ptr;

  struct list_distrib {
    struct list_distrib *next;
      /* pointer to next in linked list */
    DISTRIBUTION *listDist_ptr;
      /* pointer to the distribution that is kept separate from the
       * global distribution list. This is also different from
       * struct comp_distrib in that other attributes could have
       * the same name for distribution but with different values.
       * Also could be a list of "non-tensorial"
       * numbers (such as stacking sequence). */
  } *ldist_ptr;
 } attrib_physical_info;
```

Pseudo-Code 1 Attribute Physical Information Data Structure

Pseudo-Code 1. The following subsections discuss each portion of the data structure in detail.

3.1.1 Attribute Tensor

The implementation of the attribute tensor data structure is the most trivial. An integer is used to store the tensor order of the attribute. This value controls the maximum number of tensor components one may input for a specific attribute.

3.1.2 Symmetry

In the implementation of symmetry, two major types need to be accounted for: *intrinsic* and *spatial property* symmetries. A data structure that captures the necessary symmetry information is presented, followed by two simple examples of how this structure may be used.

The data structure for the symmetry information is shown in Pseudo-Code 2. Since there may be many levels of symmetries within one tensor, a linked list structure is used to define the symmetries of the tensor. A linked list is suitable because it allows for flexible definition of list members (in this case, symmetries) and memory is allocated only on an as needed basis. The first field of the symmetry structure is a pointer

```
/* Define the data structure for the symmetry information */

typedef struct symmetry {

  struct symmetry *next;
    /* pointer to the next symmetry */

  long symmetry_type;
    /* code for the different types of symmetry */

  SYMMETRY_INFO *LHS_symmetry;
    /* define the structure to store information about the left-
     * hand-side of the symmetry */

  SYMMETRY_INFO *RHS_symmetry;
    /* define the structure to store information about the right-
     * hand-side of the symmetry */

      } SYMMETRY;
```

Pseudo-Code 2 Symmetry Data Structure

to the next symmetry definition. The second field specifies the type of the current symmetry. At the time of implementation, twelve types are identified: 1-symmetric, 2-totally symmetric, 3-antisymmetric, 4-totally antisymmetric, 5-plane symmetry, 6-line symmetry, 7-point symmetry, 8-plane-line symmetry, 9-anisotropic, 10-orthotropic, 11-transversely isotropic, 12-isotropic. Each type is represented by the corresponding integer. The third and fourth fields are pointers to linked lists of real numbers. The structure is shown in Pseudo-Code 3.

```
/* define the structure to store symmetry information */

typedef struct symmetry_info {
  struct symmetry_info *next;
    /* linking to the next piece of symmetry item */

  float sym_info;
    /* This stores the first piece of symmetry information.
     * For LHS, this could be the left-hand-side index of
     * the symmetry, or it could be the angle of symmetry
     * For RHS, this could be the right-hand-side index of
     * the symmetry, or it could be the axis number of the
     * axis or axis plane that is coincidental with the
     * line or plane of symmetry */

} SYMMETRY_INFO;
```

Pseudo-Code 3 Symmetry Information Structure

To explain the third and fourth fields, recall from the discussion on symmetry in section 2.1.2 the symmetries can be separated into types of intrinsic and spatial property. For intrinsic tensor symmetries, only the index locations of the symmetry need to be identified. These locations are stored as linked lists of numbers. For example, to specify a major symmetry for the stiffness tensor ($C_{ijkl} = C_{klij}$), the information stored in the data

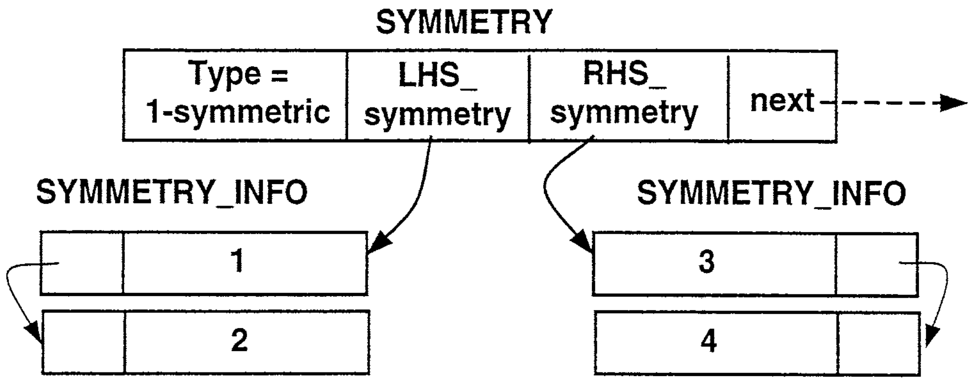


Figure 14 Example of storing major symmetry information

structure is shown in Figure 14. The *symmetric* label indicates the nature of the tensor symmetry. For a major symmetry, index positions *1* and *2* can interchange with index positions *3* and *4* while still retaining the value of the tensor. Therefore, the numbers *1* and *2* are stored in the first linked list while the numbers *3* and *4* are stored in the second. Thus all symmetry components are specified.

For the spatial property symmetries, if the symmetry is with respect to a particular angle (e.g. a line symmetry), this angle is stored in the first slot in the first linked list. The first slot in the second linked list is used to identify the axis that coincides with this line. It is assumed that the coordinate system is defined such that its axes or axis surfaces coincide with the line, plane, or plane-line symmetries of the system. For example, if a tensor has a symmetry with respect to the *z*-axis every 36 degrees, the symmetry information stored is shown in Figure 15.

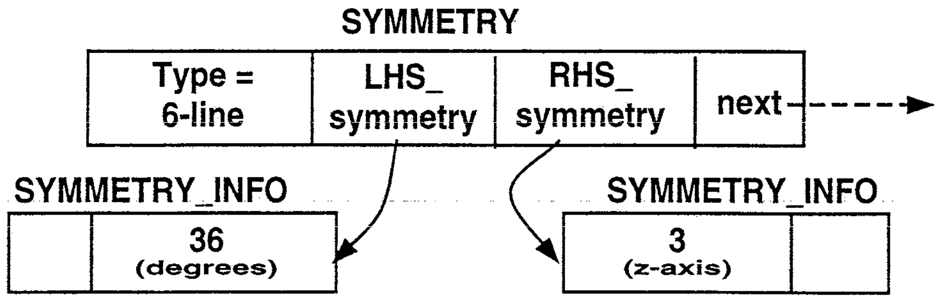


Figure 15 Example of storing of line symmetry information

3.1.3 The Parser

A robust and efficient handling capability of distributions is essential to the operation of an effective general purpose attribute manager. Many of the issues regarding distribution definition requirements had already been discussed in section 2.1.3. Some of these issues include the capability of complete understanding and storage of equations, equation evaluation capabilities (with various dependent and independent variables), and the ability to incorporate results from other analyses. In response to these requirements, a general purpose equation parser is developed to parse distributions into a format suitable for equation manipulation and evaluation. The implementation of the parser is described in this section, followed by a description of the distributions in the next.

Part of the parsing routines is a lexical analyzer that provides the parser with a workable block of data from the given equation. The parsing routines developed has the ability to take almost any equation, sort out the individual equation parameters, and put them in a form useful for manipulation and evaluation. The parser is generated by **yacc** — Yet Another Compiler Compiler, a program that generates a parser based on a set of grammatical rules. **yacc** was used for compilers of many conventional and unconventional computer languages, desk calculator languages, and a debugging system [9][10][20]. The parser is written so that it can obtain an equation from a given character string or an input file. Given an equation, the lexical analyzer grabs the next recognizable parameter. The parser then takes this parameter and, according to a predefined set of rules, invoke the appropriate routines to build up the parse tree with the new parameter information. Once the entire equation is parsed, the parsed tree is then stored as a new distribution. This distribution is then given to the calling routine.

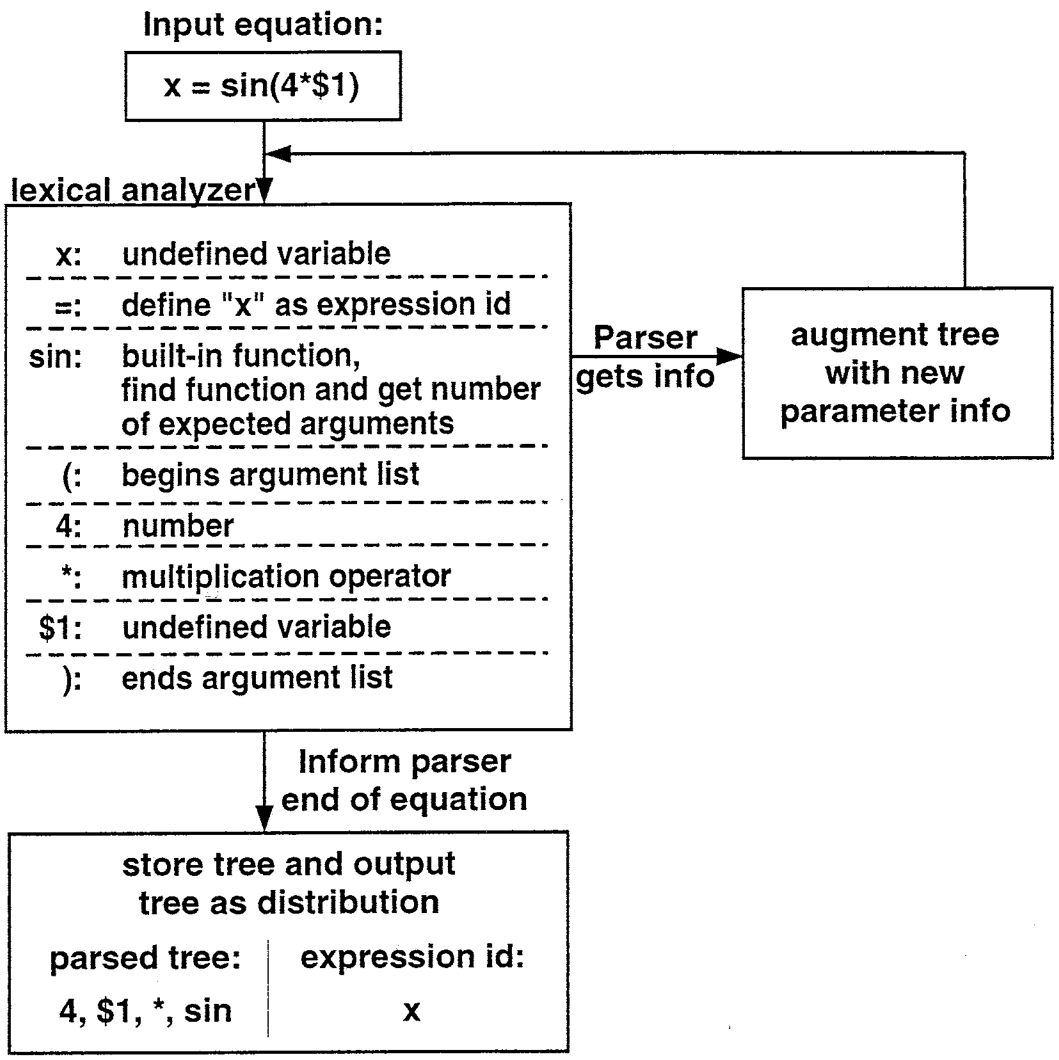

Input equation:
x = sin(4*$1)
lexical analyzer
x: undefined variable
=: define "x" as expression id
sin: built-in function, find function and get number of expected arguments
(: begins argument list
4: number
*: multiplication operator
$1: undefined variable
): ends argument list
Parser gets info
augment tree with new parameter info
Inform parser end of equation
store tree and output tree as distribution
parsed tree:
4, $1, *, sin
expression id:
x

Figure 16 Example of Equation Parser

The basic flow of the equation parser can be illustrated with a simple example as shown in Figure 16. Given the example input equation of $x = sin(4 * \$1)$, the lexical analyzer takes the parameter of the equation from left to right. The first parameter it takes is "*x*", which is marked as an undefined variable. The parser then decides to have the lexical analyzer grab the next parameter, which is "=". With this new piece of information, the parser now knows that "*x*" is actually an id for the expression. So it retags "*x*" as an expression id. When the lexical analyzer gets to "*sin*", it checks and recognizes that the parameter is one of the built-in functions. So it is stored into the parse tree tagged with a type of "built-in function." The set of built-in functions are predefined and built into the database at the time of compilation. Pseudo-Code 4 shows the data structure for the definition of the built-in functions and the constants. The built-in functions are described in the latter part of this subsection. Noting that "*sin*" is actually a built-in function, the parser expects the parameters following "*sin*" to be arguments to this function — beginning with the open parenthesis "*(*", deliminated with commas "," and ending with a closed parenthesis "*)*". So, for the argument list, the first parameter the parser encounters is "*4*". The parser tags it as a number. Next, a "*" is encountered. The parser correctly interprets it as a multiplication operator — thus knowing the preceding "*4*" is actually one of the operands of "*". The parser than gets the second operand of "*$1*". Since it doesn't match any of the constants or the built-in functions, it is tagged as an undefined variable. Finally, the parser encounters the closing parenthesis, thus knowing that the argument consists of one single expression of "*4 * $1*". Each time the parser receives a piece of information that unambiguously defines the parameter, it is augmented to the parse tree. At the end of the parsing effort, the resulting parse tree has the expression stored in a reversed Polish format: *4*, *$1*, *, and *sin* with an expression id of "*x*".

```
extern double Log(), Log10(), Exp(), Sqrt(), Integer();
extern double Sin(), Cos(), Tan(), Asin(), Acos(), Atan();
extern double Sinh(), Cosh(), Tanh(), Atan2(), Mod(), Fabs();
extern double GT_temp(), LT_temp(), GS_disp(), GT_amp();

static struct { /* Constants */
  char *name;
```

Pseudo-Code 4 Data Structure for Built-in Functions and Constants (Continued) . . .

```
  double cval;
} consts[] = {
  "$PI", 3.14159265358979323846,
  "$E", 2.71828182845904523536,
  "$GAMMA", 0.57721566490153286060, /* Euler */
  "$DEG", 57.29577951308232087680, /* deg/radian */
  "$PHI", 1.61803398874989484820, /* golden ratio */
  0, 0
};

static struct { /* Built-ins */
  char *name; /* name of the input function */
  double (*func)(); /* pointer to the function */
  int noargts; /* number of arguments */
} builtins[] = {
  "sin", Sin, 1,
  "cos", Cos, 1,
  "tan", Tan, 1,
  "asin", Asin, 1,
  "acos", Acos, 1,
  "atan", Atan, 1,
  "sinh", Sinh, 1,
  "cosh", Cosh, 1,
  "tanh", Tanh, 1,
  "atan2", Atan2, 2,
  "mod", Mod, 2,
  "log", Log, 1, /* checks argument */
  "log10", Log10, 1, /* checks argument */
  "exp", Exp, 1, /* checks argument */
  "sqrt", Sqrt, 1, /* checks argument */
  "int", Integer, 1,
  "abs", Fabs, 1,
  "GT_temp", GT_temp, 3,
  "LT_temp", LT_temp, 3,
  "GS_disp", GS_disp, 3,
  "GT_amp", GT_amp, 3,
  0, 0, 0

};
```

Pseudo-Code 4 Data Structure for Built-in Functions and Constants

As one can see from Pseudo-Code 4, any function that the lexical analyzer needs to recognize must be included in the structure. At the moment, this structure is defined in the same physical file as the initialization routine called **init()**, which puts the functions and constants into a table. To define a constant, one must give the name by which the constant is to be referenced. For the attribute manager, a constant always starts with a "$" sign, followed by an alpha-numerical string, and is always capitalized. As described in Section 3.1.5, the variable "*$n*", where *n* is a numerical number, is a special variable syntax reserved for the specification of coordinate system independent variables. Other than constants and coordinate system independent variables, no other types of variables may begin with a "*$*". The second piece of information needed to define a constant is the value of the constant. To define the built-in functions, notice that the data structure named "builtin[]" takes three fields to store the function: 1) the name which the lexical analyzer can recognize and by which the actual function is referenced, 2) the pointer to the entry point of the actual function which corresponds to the name (specified as the actual name of the routine), and 3) the number of arguments the function takes. Notice, too, that two types of functions are stored here: 1) functions in the system math library, such as sin() and cos(), and 2) functions defined by the user, such as "GT_temp". When a new user function is defined, it must be incorporated into the structure "builtin[]" and the corresponding prototype must be specified at the top of the file. In addition to providing the three fields mentioned above, the input argument specification must be in the format of

```
double function_name (long number_of_arguments,
                      double argument_array[])
```

where the **function_name** must return a double. The first argument, **num-**

ber_of_arguments, must always specify the number of arguments this function takes, and the second argument, **argumente_array[]**, must be a double precision array of the argument values. This is done so that the attribute manager will have a consistent way of handling user defined functions. Incidentally, this is also the approach taken in incorporating the results of one analysis as boundary conditions for another analysis. That first analysis simply has to provide an operator in which one can retrieve the needed information to the attribute manager. In the way described, this operator can be incorporated into the structure and be used as a distribution for attributes. The operator/user defined function can then be used as a distribution once it is linked with the parsing routines.

3.1.4 Distribution

The versatile capabilities of the parsing routines is vital in a complete understanding of any input equations. As was discussed, equations are stored as distributions. Pseudo-Code 5 shows the data structure of a distribution. Each distribution is treated as a cell in a linked list. The data type is defined to be DISTRIBUTION. The first field is a pointer that points to the next distribution. The second field provides for a way to reference this distribution. For example, *label will be "b" for the distribution "b = 4 sin($\theta\pi$)." For those distributions given without name (such as "4 sin($\theta\pi$)"), a sequential name of "@n" is given to that distribution — where n is taken to be the n^{th} such equation without any name. The relationships between the distribution components are stored in the third field. Within this third field, there is a character string that stores the entire distribution (for example, "b = 4 sin($\theta\pi$)"). This string is also parsed and stored in reverse-Polish format

```
/* data structure for distribution */
typedef struct distribution {
 struct distribution *next;
 /* pointer to the distribution information of the next distribution */

 char *label;
 /* a label for this distribution for later identification
  * for example: "a" for the equation, a = 2 + sin(x) */
 /* define the structure that stores the parameter relationship of the tensor
  * component distribution in a string and a 2-D binary tree array */

 struct parameter_reln {
  char *char_string;
  /* a character string that stores the function that describes the distribution
   * of the tensor component in a string form, including the label. */

  long tree_size;
  /* an integer that specifies the row size of the binary tree below */

  BINARY_TREE *bin_tree;
  /* The above character string is first parsed into a binary tree, which is
   * then stored in this structure called binary_tree. Notice its array size
   * is not specified until at definition */
 } PARAMETER_RELN;

 PARAM_INFO *param_ptr;
 /* define the structure that define each of the parameters
  * in the character string in PARAMETER_RELN */
} DISTRIBUTION;
```

Pseudo-Code 5 Distribution Data Structure

in a 1–D binary tree array in this third field. The exact nature of each of the parameters are stored in the last field of DISTRIBUTION and is defined to be PARAM_INFO.

The PARAM_INFO data structure is shown in Pseudo-Code 6. Again, a linked list for the structure of the parameter information is most appropriate since the number

```
/* the parameter info of the distribution */
typedef struct param_info {
 /* NOTE: only the parameters that are not numbers nor alphanumeric (such
  * as "+", "-", "*", "/") are listed here as a separate parameter.  */

 struct param_info *next;
 /* pointer pointing to the next parameter */

 char *param_string;
 /* a string that contains one of the parameters
  * described in the character string in parameter_reln */

 long param_type;
  /* type code for the parameter:
   * 1 = constant, 2 = builtin function,
   * 3 = variable, 4 = attribute , 5 = distribution */

 /* informaton to the parameter dependencies */
 union param_dependency {
  double const_value;
  /* value of the constant (type 1) */

  struct bltinfo {
   long noargts;
   /* number of arguments for this function.  */

   double (*ptr)();
   /* pointer to a builtin function that
    * returns a double.  */
  } BLTINFO;

  struct distribution *distrib_ptr;
  /* pointer to the distribution on which the parameter is defined (types 5) */

  struct attribute *attrib_ptr;
  /* pointer to the attribute on which the parameter is defined (type 4) */

 } PARAM_DEPENDENCY; /* param_dependency */
} PARAM_INFO;
```

Pseudo-Code 6 Structure of Parameter Information of parameters may vary with each equation. The first field is a pointer to the next PARAM_INFO, and the current parameter in character string form is stored in the second field. Aside from the basic operators (such as "+" and "-") and numbers there are five types of parameter dependencies the manager needs to support: constants, built-in functions, variables, attribute components, and other distributions. The third field indicates the type of the current parameter. Depending on the type, the appropriate information to qualify the parameter is stored in the last field. The parser and all the associated routines are responsible to provide, sort, and put all the information for the distribution in the appropriate places.

An added benefit of associating pointers with distributions can be easily illustrated with the case of a prescribed boundary condition as shown in Figure 17. The horizontal displacement at wall A and vertical displacement at wall B are prescribed to be zero, while the other component of the displacements are not prescribed at all. In specifying the prescribed displacements, one must be able to distinguish whether the tensor component is actually defined or is explicitly zero. In this case, only those components explicitly

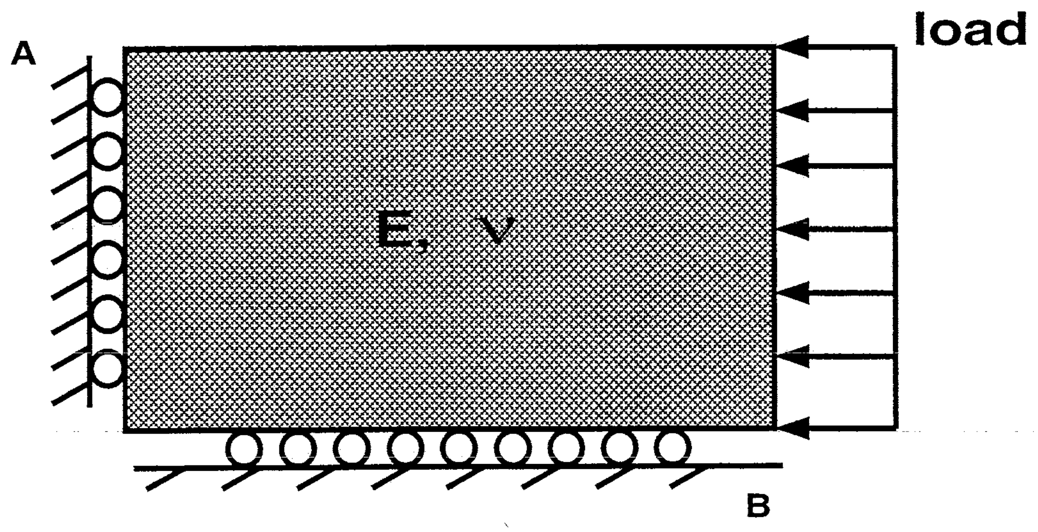


Figure 17 Example of prescribed boundary condition

defined have pointers to the associated distributions. The components not defined point to a NULL pointer.

The distribution structure is not only used in defining the tensor component distribution, but it is also used in defining the relationship of axis directions between coordinate systems. Situations frequently arise such that some analysis expressions become cumbersomely large and complicated or that two distributions of different values might use the same name. For the latter situation, a common occurrence is when specifying the stiffness of two different materials. Both stiffnesses might depend on E and ν, though the values of E and ν might be different. It is unreasonable to require and unintuitive for the user to keep track of the number of different E's and ν's and assign them unique names. Just as in a programming language, such as C, there is need for the distinction of global and local variables, so too there is a need to distinguish between local and global distributions. By local it is meant local to an attribute, i.e., the distributions will be stored and associated only with a particular attribute. The local distribution is not seen by distributions outside of that attribute. This allows for the specification of local distributions whose name might conflict with other globally and locally defined distributions. When the name of a local distribution conflicts with that of a global distribution, the local name overrides the global name. This concept of local and global definitions also provides a powerful way to break down cumbersome distributions into smaller, more readable, chunks.

Recall the data structure defining the physical information of an attribute as shown in Pseudo-Code 1. The fourth field, *comp_distrib*, is used to store the pointers to distributions defined for each tensor component. The fifth field, *struct list_distrib* is a linked list of pointers to the distributions defined local to that attribute. When parsing an equation, this local list of distributions is checked for dependencies first. Only when dependencies are not found that the global distribution list will then be checked.

3.1.5 Coordinate System

Three pieces of information are required to specify a coordinate system (as concluded from the discussion of section 2.1.4): 1) a reference that identifies the base system upon which the current coordinate system is defined, 2) the coordinates of the origin of the current coordinate system in the space of the base coordinate system, and 3) the functions that define the relationship between the coordinates in the current coordinate system and the base coordinate system. For the purposes of implementation, it is also desirable to have two identification labels in addition to the above three pieces of information. The first label is to uniquely identify the coordinate system for later reference, and the second is to identify the coordinate system type: linear, curvilinear, or model, in a label. This second label allows for easier recognition of the coordinate system type to insure correct use of the appropriate transformation relations. The scheme used for defining a coordinate system is shown in Figure 18.

The corresponding data structure to store a coordinate system is shown in Pseudo-Code 7. The structure to hold coordinate system information is specified as a linked list. The first field is a pointer to the next coordinate system. The pointer to the reference coordinate system is stored in the second field. The unique character string identification for the coordinate system is stored in the next field. As was discussed in section 2.1.4, three types of coordinate systems can be identified. The fourth field specifies the coordinate system type: 1–linear, 2–curvilinear, 3–model. A 1–D array is used to store the coordinate of the origin of the current coordinate system in the space

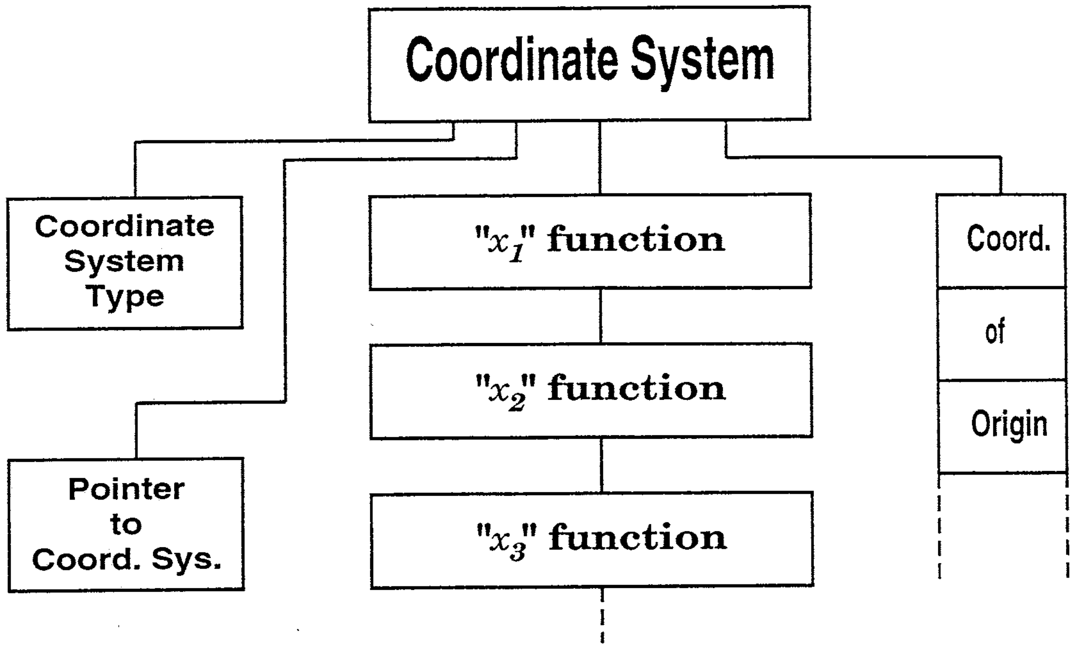


Figure 18 Data structure of coordinate system definition

of the base coordinate system. This fifth field is not defined until the coordinate system is defined. Given the dimension of the coordinate system (the sixth field), the dimension of the coordinate of the origin with respect to the reference coordinate system is set and defined. Finally, the relationship of the current coordinate system with respect to the reference coordinate system is in the last field of the structure. Each of the functions that describes the coordinate system axis are described and stored in the same manner as the distributions. These functions must be defined with respect to the base coordinate system. To reference the functions of the base coordinate system, the syntax of *$n*, where *n = 1, 2, 3, ...* that corresponds to the value of the 1st, 2nd, 3rd, ... base coordinate system function. For example, ($1, $2, $3) specifies the value of the coordinate (x,y,z) defined in an x-y-z coordinate system. The suite of functional operators to evaluate the distributions and transform coordinates from one coordinate system to another has yet to be developed. These operators will make use of the evaluating routines developed for the parser.

```
/* Define the data structure for the coordinate system information */
typedef struct coordinate_sys {
  struct coordinate_sys *next;
  /* pointer to the next coordinate stored in the linked list */

  struct coordinate_sys *coord_sys_ptr;
  /* pointer to the coordinate system based on which the
   * current coordinate system is defined */

  char *coord_sys_id;
  /* an label that uniquely identifies this coordinate system */

  long coord_sys_type;
  /* a flag specifying the type of the current coordinate system */

  float *coord_origin;
  /* an array to store the coordinate of the origin of the current coordinate system
   * with respect to the coordinate system pointed to by coord_sys_ptr – this
   * number is written in terms of the current coordinate system.
   *
   * To provide for arbitrary length of array, the size is not specified until definition
   * time.  It is of the dimension of the coordinate system.  */

  long coord_sys_dimension;
  /* the dimension of the coordinate system */

  struct distribution **func_ptr;
  /* the functional relationship of the current coordinate system with respect to a
   * base coordinate system is defined by an array of distributions.  The size of
this
   * array is the dimension of the coordinate system, which is indeterminant until
   * after definition */
} COORDINATE_SYS;
```

Pseudo-Code 7 Coordinate System Data Structure

Something not discussed above is when the label is "model." This label is used when the user wants to define a coordinate system local to a geometric model entity. In this case, the attribute modeler must query the geometric modeler for information about the coordinate system in which the model entity is defined. This set of operators is modeler specific and has not been implemented. The first coordinate system function defines the direction normal to the model entity. The subsequent coordinate functions define the tangents to the model entity.

3.2 Organizational Implementation of Attributes

Two important characteristics of the organizational structure proposed and discussed in section 2.2 are flexibility and generality — flexibility in the handling of various combinations of associations and generality in the handling of multiple and different applications. These are the main requirements for the implementation of the organizational structure. The organizational types proposed and defined (in order from highest hierarchical order to the lowest) are: case, group, set, and attribute. An additional type of "multiplier" allows for convenient variation to the value of any of the organization hierarchies. This section discusses the data structure and implementation details of each of these organizational hierarchies.

Figure 9 from section 2.2 (reproduced here as Figure 19) provides a global view on the overall framework of the organizational structure. Three characteristics about the organizational structure as depicted are particularly important:

1. The hierarchies shown in Figure 19 must not be violated. That is, an attribute cannot be above a set in its hierarchical levels, neither can a set be above a group, or a group be above a case.
2. Not all the hierarchies need to be present in one particular branch of the tree. Notice that the right most branch of the tree in Figure 19 is composed of a case-set-attribute.

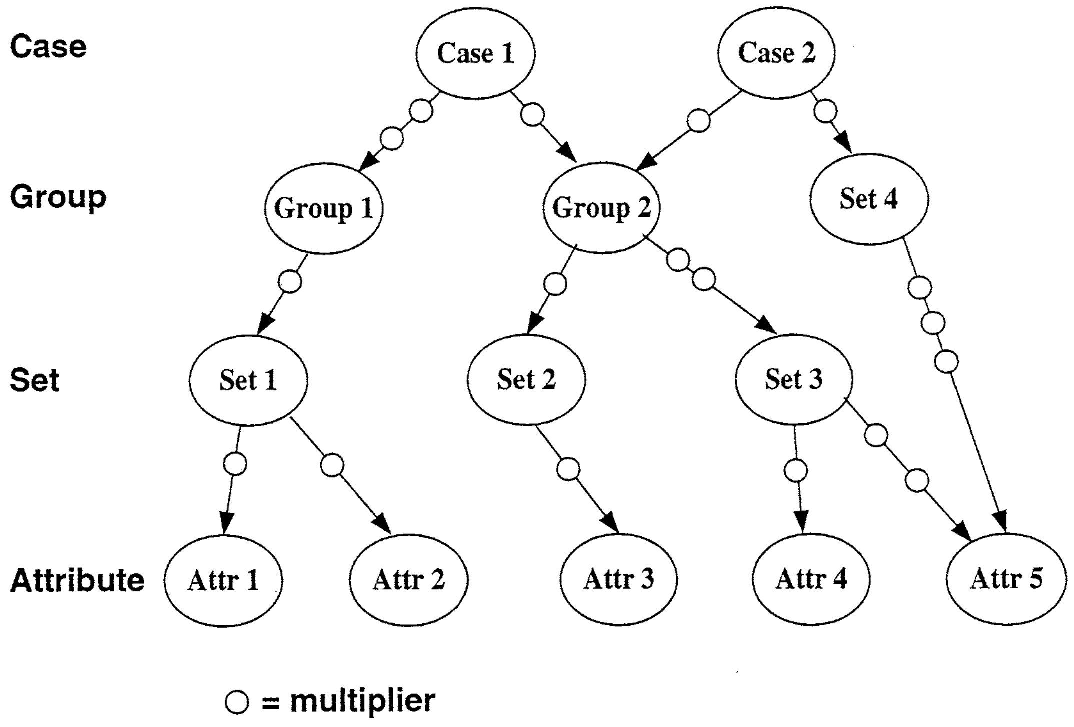


Figure 19 Overall Framework of the Organization Structure

3. Any number of multipliers can be attached in between any pair of hierarchical parent-child.

These hierarchy characteristics are the rules used in the implementation of the organization structure.

Before the connection between the organization elements is established, both elements must be defined. If after a search of the elements and one or both of the candidates were not found, a linked list cell of the type of the element is created and the connection is made. This automatic creation of organization elements only holds for case, group, and set. An attribute *must* be previously defined. Within any pair of such elements, the child node cannot be higher in the hierarchical level than that of the parent node according to the first of the three organizational rules described above. Within this constraint, the parent node can be a case, group, or set, while the child node can be a group, set, or attribute. All of the organization elements are structured as linked lists. Linked lists are advantageous in this application because it gives great flexibility in inserting new nodes or deleting existing nodes without having to rearrange the entire data set. Furthermore, the nodes could be stored as a search tree for efficient retrieval of specific linked list nodal information. Therefore, in addition to the interconnection between case, group, set, and attribute, (as depicted in Figure 19) each of the organization elements are also linked into a list. What emerges from this is an additional layer of the organization structure shown in Figure 20. The user does not see this structure, and it is used primarily in information bookkeeping and searching. All searching are done from left to right and top to bottom. The structure of each of the organization element is examined in detail in the following subsections.

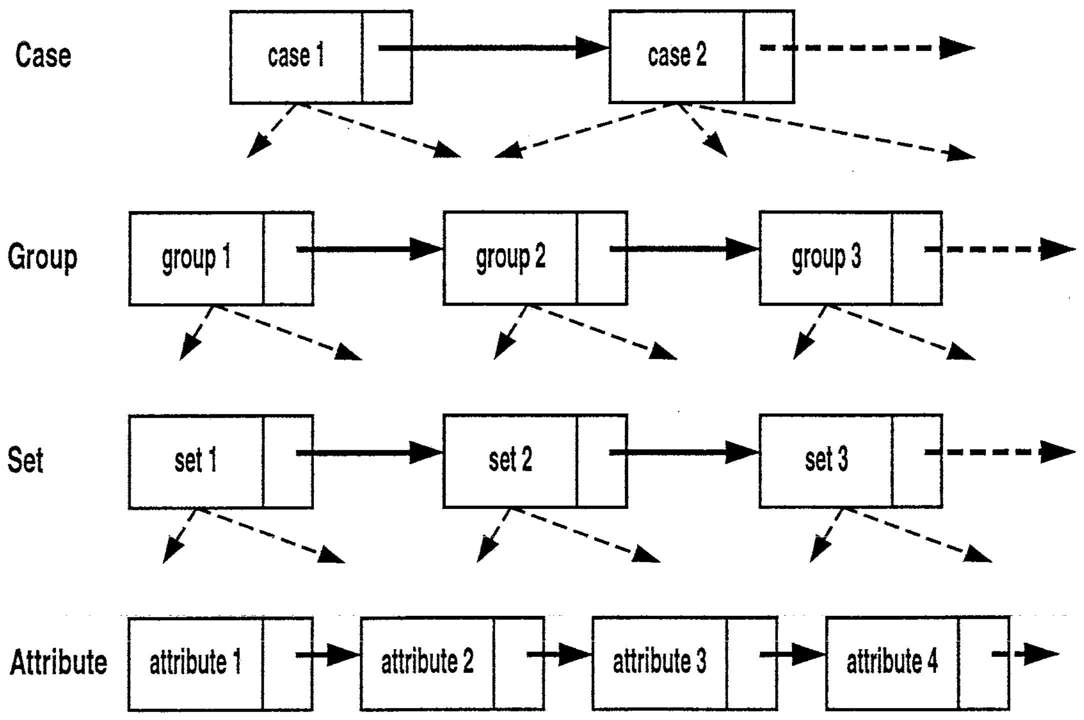


Figure 20 Additional Layer of Organization Structure

3.2.1 Case

The relationship of a case within the organization structure is diagrammed in Figure 21, and the corresponding data structure of a case is shown in Pseudo-Code 8. The circles represent multipliers. The structure is defined to be of type CASE. The first field is a pointer that points to the next node in the case linked list. The second field is a unique character string identification of CASE. A brief description of the case can be stored in the third field. The next field is used to establish the connection between organization elements; it is a pointer to the list of child nodes of the case. The children can be of the group, set, or attribute. The flexibility built into this scheme requires that the case can point to any element of a lower hierarchical level. This is a challenge in the declaration of a group, set, and attribute. They have different declarations, yet they need to be declared in such a way that the down-pointer can recognize as a single declaration. The approach taken here to solve this problem is to make use of the *union* declaration in C. Instead of having the child pointer of CASE pointing directly to a group, set, or attribute,

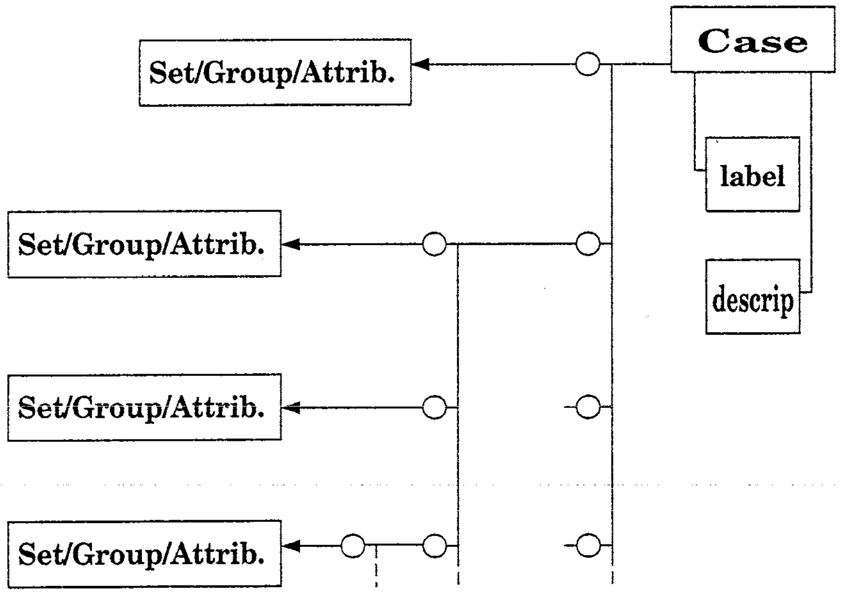


Figure 21 Relationship of a Case Within the Organization Structure

```
/* Define the data-structure of attribute CASE */
typedef struct cas {
 struct cas *next;
  /* pointer to the next case */
 char *label;
  /* a unique case label */
 char *Case_description;
  /* a brief description of the case */

 Generic_List_Cell *ptr_grp_set_att;
  /* pointer to the linked list of pointers to the groups, sets, or attributes */

 struct combined *union_ptr;
  /* for efficiency in nodal traversal, this pointer retains the knowledge of the
   * generic pointer pointing to this case.  */

} CASE;
/* end of the structure of CASE */
```

Pseudo-Code 8 Case Data Structure

it points to a union of the four hierarchical elements. Every organizational element has an associated union. It is convenient to have this pointer to the associated union in the definition of the element to eliminate any searching. The last field is this union pointer.

The general structure of a union is shown in Pseudo-Code 9. The structure shown in Pseudo-Code 9 is defined to be COMBINED, which is used as targets of the up-/down-pointers of the organization element. The first field indicates the element to which this union is actually pointing. Given this knowledge, the appropriate pointer is selected in COMB. This structure is vital in supporting the flexibility demanded in the organization structure and in the writing of compact and general code. This approach of handling down-pointers has also been extended to the way up-pointers are handled. Incidentally, the fifth field of CASE as shown in Pseudo-Code 8 is the pointer to the target to which another organization element can point when establishing a connection to this case.

```
/* Define a union of attributes-sets-groups-cases to be used for the
 * up-/down-pointer */
typedef struct combined {
  /* an integer label specifying what the COMB is pointing at */
  /* ATATT=attribute, ATSET=set, ATGROUP=group, ATCASE=case */
  ORGNCODE label;

  /* variable pointer pointing to any one of the structures below */
  union comb {
    ATTRIBUTE *Uatt;
    struct set *Uset;
    struct group *Ugroup;
    struct cas *Ucase;
  } COMB; /* end of comb */

} COMBINED; /* end of the union of set/group/case */
```

Pseudo-Code 9 Union Data Structure

3.2.2 Group

The relationship of a group within the organizational structure is diagrammed in Figure 22, and the corresponding structure of a group is shown in Pseudo-Code 10. The structure is defined to be of type GROUP. The first field is a pointer that points to the next node in the group linked list. The second field is a unique character string identification of GROUP. A brief description of the group can be stored in the third field. The next two fields are used in establishing connection between organization elements, and the sixth field is the pointer to the associated union of this group. The fourth field is a pointer to the list of child nodes of the group. The children can be of the set or attribute, and the fifth is a pointer to a list of parent nodes of the group. As can be seen in the diagram, the up-pointers are different than the down-pointers by the absence of little circles in the connection. Again, these circles represent multipliers. Since the evaluation routine *always* go from top to bottom, only the down-pointers need to know of the multipliers.

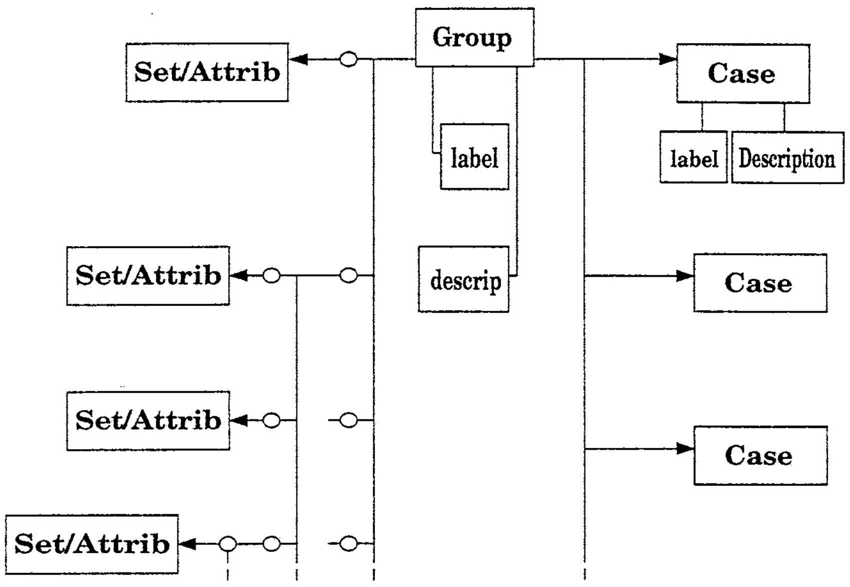


Figure 22 Relationship of a Group Within the Organization Structure

```
/* Define the data-structure an of attribute GROUP */
typedef struct group {
  struct group *next;
    /* pointer to the next group */

  char *label;
    /* a unique group label */

  char *Group_description;
    /* a brief description of the group */

  Generic_List_Cell *ptr_set_att;
    /* pointer to the linked list of pointers to the sets and/or attributes */

  Generic_List_Cell2 *ptr_case;
    /* pointer to the linked list of pointers to the case belonging to the group */

  struct combined *union_ptr;
    /* a back pointer pointing to the associated union. */
} GROUP; /* end of the structure of GROUP */
```

Pseudo-Code 10 Group Data Structure

3.2.3 Set

The relationship of a set within the organization structure is diagrammed in Figure 23, and the corresponding structure of a set is shown in Pseudo-Code 11. The structure is defined to be of type SET. The first field is a pointer that points to the next node in the set linked list. The second field is a unique character string identification of SET. A brief description of the set can be stored in the third field. The next two fields are used in establishing connection between organization elements. The fourth field is a pointer to the list of child nodes of the set. The children can only be of the type attribute, and the fifth is a pointer to a list of parent nodes of the set. The parent nodes can be a group or a case. Again, notice from Figure 23 that the pointers to the parent nodes do not have

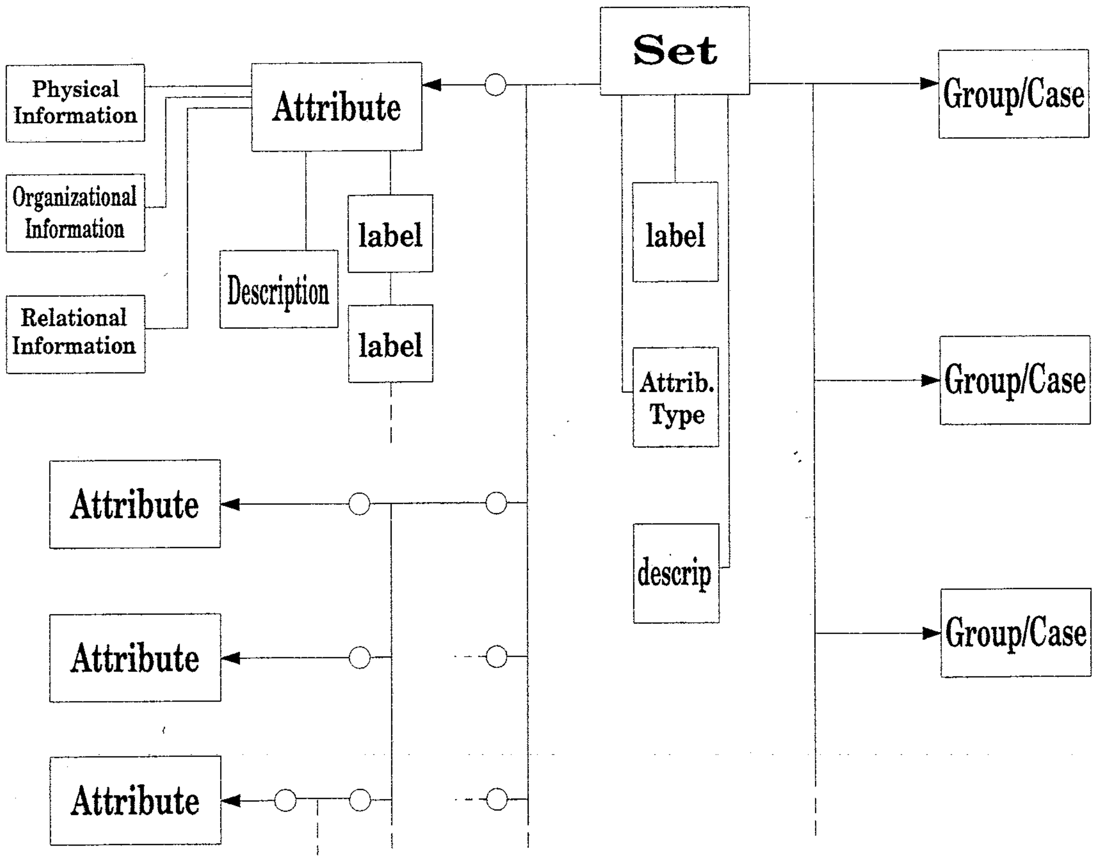


Figure 23 Relationship of a Set Within the Organization Structure

```
/* Define the data-structure of attribute SET */
typedef struct set {
  struct set *next;
    /* pointer to the next set */

  char *label;
    /* a unique set label */

  char *Set_description;
    /* a brief description of the set */

  Generic_List_Cell2 *ptr_grp_case;
    /* pointer to the linked list of pointer to the parent groups */

  Generic_List_Cell *ptr_attrib;
    /* pointer to the linked list of pointers to the children
     * attributes associated with the set */

   struct combined *union_ptr;
    /* a back pointer pointing to the associated union.  */
} SET; /* end of the structure of SET */
```

Pseudo-Code 11 Set Data Structure

any associated multipliers. This is because these pointers are for searching conveniences only. The sixth field is the pointer to the associated union of this set.

3.2.4 Attribute

The structure of an attribute is shown in Pseudo-Code 12. The structure is defined to be of type ATTRIBUTE. It is structured quite differently from SET, GROUP, or CASE. Recall that the design of the attribute requires it to carry considerably more information — namely, information of the attribute physical definition and information of how it relates to other components of the analysis. The structure of the physical of information

```
/* Data structure of Attribute Abstract Data Type */
typedef struct attribute {
  struct attribute *next;
    /* pointer to the next attribute */

  Generic_List_Cell2 *label;
    /* pointer to the linked list of labels of the attribute */

  char *Attrib_description_ptr;
    /* a brief description of the attribute */


  /* Define the fields containing the organizational
   * descriptors of the attributes */
  struct ORGANIZATIONAL_INFO {
    char *attrib_type;
      /* a string specifying the attribute type */

    struct combined *union_ptr;
      /* a pointer pointing to the associated union. this union will be used as the
       * target of the up/down pointers for linked lists of parents/children. */

    Generic_List_Cell2 *attrib_set_grp_case;
      /* pointer to the linked list of pointer to the sets,
       * group, or, case with which the attribute is associated */
  } attrib_organization_info;

  /* Define the fields containing the physical information of the attribute */
  struct PHYSICAL_INFO { ...
  } attrib_physical_info;

  /* Define the fields containing the information regarding the
   * relationship of the attribute with the rest of the system. */
  struct SYS_RELATIONSHIP { ...
  } attrib_sys_relationship;
  /* end of the system relationship information of the attribute*/

} ATTRIBUTE; /* end of the data-structure of the ATTRIBUTE */
```

Pseudo-Code 12 Attribute Data Structure has been discussed in section 3.2, and the relational information will be discussed in Section 3.3. The first field of ATTRIBUTE is a pointer that points to the next node in the attribute linked list. The second field houses a unique character string identification of ATTRIBUTE as well as a finite number of labels for ATTRIBUTE. The labels provide a means for more detailed identification of the attribute. In the REPAS project, for example, the label is a useful means to specify the type of material property of which the stiffness attribute is describing (such as Linear-Isotrpoic-Elastic-Material). As one can see from the declaration, a linked list is used for the specification of labels. The first cell in that list is *always* the id, followed by additional labels, if any. A brief description of the attribute can be stored in the third field. The next field is a structure for the organizational information. Residing in this field are three pieces of information for the organization aspect of the attribute. The first of these is the attribute type. Recall that only attributes of the same type can be combined to form a set. Some examples of attribute types are stiffness, temperature, and power density. The second of the three is the pointer to the associated union of this attribute. The last of the three is a pointer to a list of parent nodes of the set. The parent nodes can be a set, group, or case.

3.2.5 Multipliers

The structure used to define a multiplier is shown in Pseudo-Code 13. Since there can be many multipliers modifying one organization node, the multiplier is structured as a linked list node. The structure is defined to be of type MULTIPLIER. The first field of MULTIPLIER is a pointer to the next node in the multiplier link. Along with the value of the multiplier as the second field, it also has a unique character string identification (id) of this multiplier. After a multiplier is defined, it can be used anywhere else by simply

```
/* define a structure for multipliers */
typedef struct multiplier {

  /* pointer to the next multiplier */
  struct multiplier *next;

  /* a unique name identifying the multiplier */
  char *label;

  /* multiplier value */
  double value;

} MULTIPLIER; /* end of the structure for multipliers */
```

Pseudo-Code 13 Multiplier Data Structure

referring to the id of this multiplier. Any changes in the multiplier value, therefore, are seen by all who use the multiplier.

If a multiplier(s) were specified to be used for the parent/child relationship and if the multiplier(s) were not found in the multiplier linked list, a new multiplier(s) with the specified value(s) is created and added to this linked list. A default multiplier with value of one is used if no value were specified.

One of the features of the multipliers is that one can define an indefinite number of multipliers for the child node of *any* organization pair. This creates an implementation challenge in that this variable list of multipliers needs to become part of the association of any two organization elements. To resolve this issue, the approach taken here is to continue to take advantage of the linked list structure. Each of the child nodes and the multipliers is wrapped in a generic linked list cell. The child nodes and the multipliers then become one and the same on the surface. The down-pointer of each organization element sees only this "wrapper," and the issue is resolved. The data structure of this

```
typedef struct generic_list_cell {

  void *generic_pointer;
   /* this field can be the pointer to any type of structure or data */

  long gptr_type;
   /* If this flag is zero, generic_pointer is pointing to a Generic_List_Cell
    * with an associated multiplier. Otherwise, generic_pointer is pointing
    * to a COMBINED structure */

  struct multiplier *mult_ptr;
   /* this field points to the structure that defines the
    * multiplier. Note this has a value only if gptr_type = 0 */

  struct generic_list_cell *next;
   /* pointer to next node in the linked list of Generic_List nodes */

} Generic_List_Cell; /* end of data-structure of Generic_List_Cell */
```

Pseudo-Code 14 Generic Wrapper Data Structure

wrapper is shown in Pseudo-Code 14.

Note that the generic wrapper is itself a linked list cell, with pointer to the next linked list cell, i.e., the next child node. In addition, the second field is a flag that directs the pointer to either the union of the organization element (first field) or to the multiplier (third field). To remain consistent in operation, the up-pointers also point to wrappers, which, in turn, points to the appropriate union of the organization element. Figure 24 shows the overall structure of one organizational element. Notice the up-pointers are not shown; these up-pointers are exactly the same as the down-pointers with the multipliers taken out. Notice also that the organizational element knows only about the generic wrapper. The wrapper then redirects the element to point to the correct node, either a multiplier or another organizational element. Between any two organizational elements are a set of multipliers that modify the value of the child of the pair.

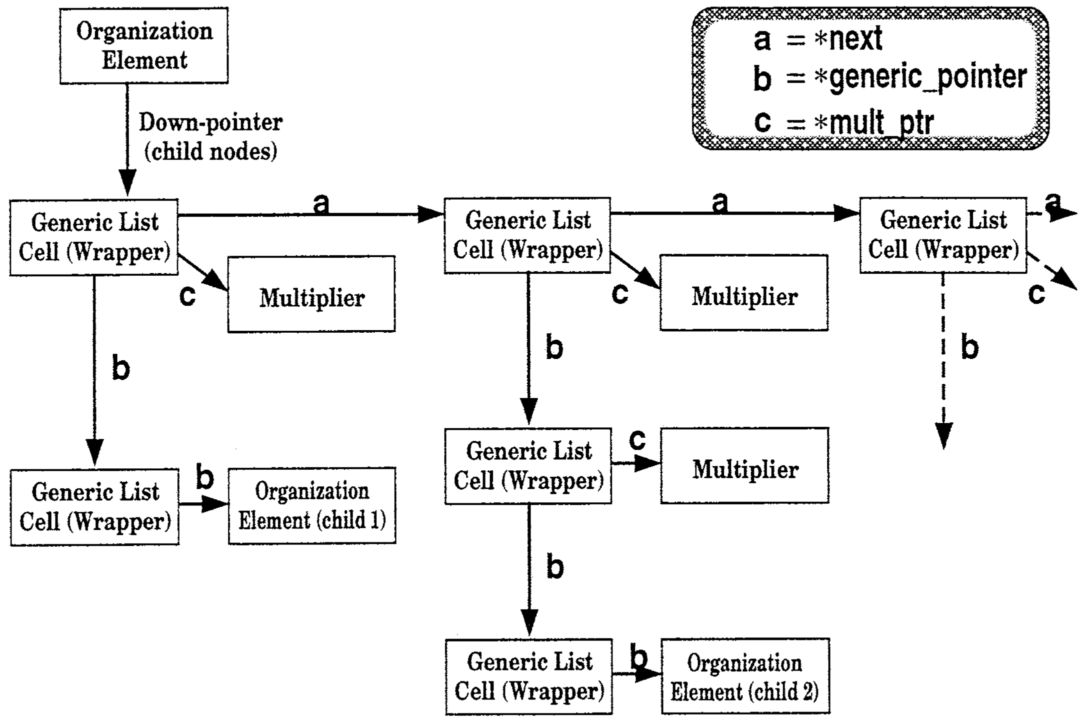

Organization Element
Down-pointer (child nodes)
a = *next
b = *generic_pointer
c = *mult_ptr
Generic List Cell (Wrapper)
Multiplier
Organization Element (child 1)
Organization Element (child 2)
a
b
c

Figure 24 Generic Picture of an Organization Element Structure

3.3 Relational Implementation of Attributes

The final step to completely specify all aspects of an attribute is to associate the attribute to the appropriate model entities. The basic capability has been implemented. Those involving creation and manipulation of auxiliary geometries and model augmentation still need to be implemented. As such the structure to hold the relational information is shown in Pseudo-Code 15.

```
/* Define the fields containing the information regarding the
 * relationship of the attribute with the rest of the system. */
struct SYS_RELATIONSHIP {
  TOPO_POINTER *ptr_topology;
  /* pointer to the header of the linked list containing pointers to the
   * topological entities with which the attribute is associated. */

  Generic_List_Cell *attrib_constraint;
```

Pseudo-Code 15 Relational Information Data Structure (Continued) . . .

```
/* Define the data-structure of TOPO_POINTER that specifies the geometric
 * information regarding the attribute. */
typedef struct topo_pointer {
  struct topo_pointer *next;
  /* pointer to the next topo-pointer */

  long entity_type;
  /* a flag specifying the entity type
   * For now, 1-vertex, 2-edge, 3-face, 4-shell, 5-region, 0–others */

  char *entity_string;
  /* pointer to the source topological entity when the pointer to the source
   * topological entity is in form of machine pointer or a string */

  long entity_number;
  /* pointer to the source topological entity when the pointer
   * to the source topological entity is in the form of an integer. */

  Generic_List_Cell2 *ptr_aux_entities;
  /* pointer to the header of the linked list of pointers to auxiliary entities */

  char *model_name;
  /* an identifier for the geometric model from which the entity is defined */
} TOPO_POINTER; /* End of data-structure of TOPO_POINTER */
```

Exhibit 16 TOPO_POINTER Data Structure

```
    /* pointer to the linked list of pointers to the structures containing
     * information regarding the constraints imposed on the attributes */
  } attrib_sys_relationship;
  /* end of the system relationship information of the attribute*/
```

Pseudo-Code 15 Relational Information Data Structure

Since an attribute can be applied to many model entities, the relational information is specified with a linked list. If the attribute is not applied to any model entities, the first field points to NULL. Otherwise, it points to TOPO_POINTER, where TOPO_POINTER is a structure defined as shown in Pseudo-Code 16

The first field of TOPO_POINTER is a pointer to the next topological entity to which the attribute is applied. The second field indicates the type of the model entity that is being described. For the moment, the type code is mainly for geometric model entities. However, the type code can be expanded as the need arises to account for different models used. Two fields are used to identify the particular model entity used. The third field can store a character string identification of the entity, such as how CATIA labels its model entities, whereas the fourth field holds the numerical identification of the entity, such as how Parasolid labels its model entities. The fifth field points to the auxiliary entities defined for this attribute, and finally, a character string identification of the model to which the entity belongs is specified in the sixth field.

4 GENERALIZED ANALYSIS FRAMEWORK (c) — APPLICATION INTERFACE FOR ATTRIBUTE MANAGER

After the attribute manager is formulated and the design implemented, the next step is to provide mechanisms to interface the manager with application packages. This chapter focuses on the I/O and interface formats used by the manager. Two methods for the manager to communicate with application programs are: 1) I/O files to "statically" create and store attributes and 2) operators to provide for dynamic interface between the manager and the applications.

4.1 I/O Formats

A file format is defined to store attributes that were created during a working session. The output of one working session maybe stored and used as input for another working session. As such, the input files are read only at the beginning of each session with the attribute manager. Any changes/additions to the definition of attributes must then be made through the interface operators during the working session with the manager. Some excerpts from the attribute output files of the REPAS application is given as example in Appendix C. A total of three files are used to specify the complete description of attributes, corresponding to the physical, organizational, and relational aspects of an attribute. These three files are:

1. *filename.lib*: a library containing information that define attributes and all the associated values and definitions,

2. *filename.org*: a file containing information that define the structure of the hierarchical organizational tree, and
3. *filename.rel*: a file containing information that define the associative relationship between the attributes and relevant model entities,

where *filename* is the name of the designated file to store the attribute information. In addition to these three files, all run time messages are written to an error file named, *filename.ATerr*.

These three I/O files are written in ASCII file format, and they are structured so as to be readable and editable by users. Just as all I/O formats have rules to govern the information storage, so do the attribute I/O files. The eight items listed below is a set of general rules that apply to all three input files. The sections following the general rules describe in detail the rules that apply specifically to each of the three input files.

1. Information is written in data blocks. Each data block is called a **unit** and is a block of data defining a particular item, such as a *coordinate system* or *multipliers*.

2. A "*" in the *first* column marks the beginning of a new unit followed by a **keyword** such as *distribution* or *list.*

3. **Sub-keywords** are keywords used within each unit, excluding the keyword that identifies the beginning of the unit. Each sub-keyword, as well as all character string inputs following the keywords and sub-keywords, must be one single word, i.e., "This_is_a_single_word", versus, "This is NOT a single word." The only deliminators for keywords are white spaces.

4. When sub-keywords are specified with no associated values or the sub-keyword is omitted in the input file, a default value is put into the data structure and a warning message is printed out to the error file. The default values for each sub-keyword are shown in the subsequent examples under the column **Default Values**.

5. Any of the following may be used as deliminators to separate data:

   :   ;   =   ,   [space]   (   )

6. All input is case insensitive.

7. A "#" in the first column of a line marks the beginning of a comment line. A carriage return marks the end of the comment line.

8. Comments may be used anywhere in the file.

4.1.1 Library Attribute File

To date, the three keywords that identify the unit blocks in *filename.lib* are:

***distribution**   ***coordinate system**   ***multiplier**

Any other keywords read are assumed to be keywords of type attribute. These keywords are used as the identifier for the attribute. Because it is a block structure, additional keywords (and consequently new units) can easily be added at a later date. The code that reads in the attribute definitions are structured in the same way for ease of expansion. The approach taken here to explain the format is to give examples of each of the units with appropriate explanations on the right-hand-side. When applicable, default values are given in the center column.

```
#
# The following is a sample distribution unit
#
```

# keywords	default values	comments
***distribution**		
a = 12; b=a+3; c=$1	-None-	Each distribution must be an equation of the form *id* = *expression* where *id* is used to identify the distribution, and *expression* is used to evaluate the distribution. In the examples given on the left, distribution **a** is evaluated to "12," where distribution **b** is evaluated to 15, with a dependency on the value of **a**. A semi-colon is used to separate two distributions. The last distribution defined in the unit must NOT have a semi-colon. There is no limit as to the number of distributions that can be defined in one distribution unit, and more than one distribution unit may be defined in a single file. The distributions defined in this unit are immediately available for use for any subsequent unit definitions, ie. these distributions are global in scope. Because of their global nature, distributions defined here have no knowledge of coordinate systems. They inherit the coordinate system of the attribute to which the distribution is applied. As a way to relate to the coordinate system in which the distribution is defined, $i is used as the convention to identify the i^{th} spatial coordinate variable of the coordinate system in which the distribution is defined (e.g., $1 represents "x" in an x-y-z coordinate system.).

#
The following is a sample coordinate system unit
#

# keywords	default values	comments
***coordinate system**		
id:	global	***<Required>***
base_id:	global/NULL	ID of coordinate system with respect to which this coordinate system is defined. If **id** were defaulted to global, **base_id** will be defaulted to NULL
origin:	(0.0, 0.0, 0.0)	Origin of the current coordinate system with respect to the indicated base coordinate system
type:	1	1=linear; 2=curvillinear (e.g., spherical coordinate system); 3=model (local to model entity)
dimension:	3	If the dimension is not given, it will be taken from the number of functions defined. The dimension must be the same as the number of functions. The functions define the current coordinate system space with respect to the base coordinate system and must be written in terms of an equation in the form:
function:	-None-	*id = expression (see distribution unit)*
function:	-None-	There must NOT be a semicolon after the function expression. Use $i to denote the ith spatial coordinate of the base coordinate system. None of the functions defined here may be used as general global distributions. The functions defined here are recognized only as the coordinate system functions.
function:	-None-	
•	•	
•	•	
•	•	

#
# The following is a sample attribute definition unit
#

# keywords	default values	comments
***displacement1**	-None-	***<Required>*** This keyword minus the "*" is taken as the id for the attribute.
Type:	-None-	A character string identifying the type of the attribute, e.g., displacement or stiffness.
labels:	-None-	A list of labels separated by deliminators may be defined here. These labels may serve to distinguish between attributes of similar types.
tensor_order:	0	If the attribute were used solely for labeling or that it is used to store a list of values, the tensor order will need to be set to **-1**, since this is not a tensor. If the tensor_order were set to **-1**, the coordinate system, symmetry, and distribution information should not be defined nor used.
coord_sys:	global	The reference coordinate system will be defaulted to "global" if no coordinate system information were given.
list:	-None-	Whereas the "distribution" units are global in scope, the expressions defined here are local in scope in that they are recognized *only* in this attribute. If a distribution defined in the list has the same name as one defined in a global distribution unit, the distribution defined in the list takes prescendence over the latter global distribution.

# keywords	default values	comments
symmetry:	-None-	The default is no symmetry.
distribution:		This sub-unit defines the components of the attribute in terms of distributions. The default for each component is: def = 1.0. The id of the distribution defined in the distribution unit may be used as variable names for the component distributions - this means that all the expressions used for components must have been previously defined in the distribution unit or in the local list.
component:	def	
component:	def	
component:	def	
•	•	
•	•	
•	•	

#
The following is a sample multiplier unit
#

# keywords	default values	comments
***multiplier**		
label:	Mdefault	A label is an ID reference.
value:	1.0	

4.1.2 Organization Attribute File

The recognizable keywords that can mark the beginning of the organization unit, *filename.org* are:

***Case** ***Group** ***Set**

By design (section 2.2, **Organization of Attribute**), the organizational units are hierarchical in nature. That is, a case can only be made up of groups, sets, and/or attributes; a group can only be made up of sets and/or attributes; and a set can only be made up of attributes. The ways to specify the dependencies are described in the following tables. Note that in the description, within a unit only one of each allowable types are described. In actual applications, one unit may have zero or multiple sub-units of the same type. For example, a case may consist of multiple sets and attributes but no groups. Figure 19 is reproduced here as Figure 25 to aid in visualizing the organization structure.

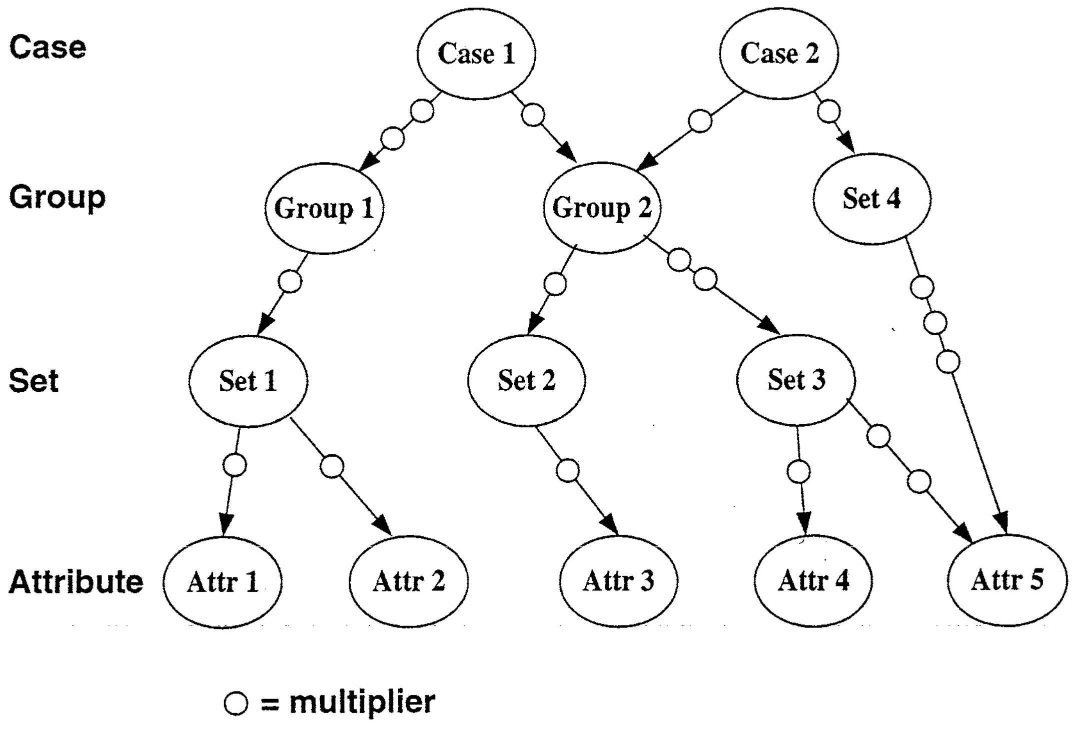


Figure 25 Overall Framework of the Organization Structure

#
The following is a sample case unit
#

# keywords	comments
***Case**: case_name	If a case is to be defined, the "case_name" must be given. Because this case cannot be referenced without an explicit name, the entire unit will be ignored if no name were given.
set = set_name	Depending on the need, any combination of the sub-keywords for this case, such as **group**, **set**, or **attribute** may be used. To have a complete/correct specification of the problem, each sub-keyword should have descendants leading down to an attribute.
group = group_name	The ID of the sub-keyword, in this instance, "group_name", must be given. The members of this group may be defined before or after this case unit.
multiplier: multa **multiplier**: multb **multiplier**: multc	The multiplier id must be that of a multiplier previously defined in the library attribute file (*filename.lib*). If it were not previously defined, a default of "Mdefault" with value of "1.0" will be used. As many multipliers as needed may be defined here. At the time of evaluation, all attributes under this organization node (in this instance, group) will be multiplied by the product of these multipliers.
attribute = attribute_name	Again, the ID must be given (attribute_name) and it will be searched and matched from the library file (*filename.lib*).

```
#
# The following is a sample group unit
#
```

# keywords	comments
***Group** = group_name	
set = set_name	Depending on the need, any combination of the sub-keywords for this group, such as **set** or **attribute** may be used. To have a complete/correct specification of the problem, each sub-keyword should have descendants that leadi down to an attribute.
multiplier: multa	Again, an arbitrary number of multipliers may be specified, so long as the ID matches one of the predefined multipliers in *filename.lib.*
attribute = attribute_name	Again, the ID of the attribute must be given (attribute_name) and it will be searched and matched from the library file (*filename.lib*).

```
#
# The following is a sample set unit
#
```

# keywords	comments
***Set**: set_name	
attribute = attribute_name	Again, the ID of the attribute must be given (attribute_name), and it will be searched and matched from the library file (*filename.lib*). The attributes specified under one set should have the same type.

4.1.3 Relational Attribute File

To date, the recognizable keywords that can mark the beginning of the relational, *filename.rel* are:

***model_name** ***attribute**

```
#
# The following is a sample unit for model_name
#
```

# keywords	comments
***model_name** = model_identification	The model identification should uniquely identify the model to which the subsequent named attributes are applied. The entities specified in the subsequent attribute units are associated with this model identification until a new id is given in another model_name unit.

```
#
# The following is a sample unit for attribute relation to model entities.
# Note: an attribute may be associated to more than one model entity, with the
# model entities belonging to the same or different models. Therefore, only a
# combination of the model_name and the attribute uniquely specifies
# the association of attributes and model entities
#
```

# keywords	comments
***attribute** = attribute_name	The attribute identified by the attribute name must be defined in the attribute library file.

# keywords	comments
model_entity_number: number	This must be a number that identifies one of the model entities of the given model, and can be a numerical tag or id of the entity.
model_entity_string: string	This must be a string that identifies one of the model entities of the given model. Either this or the "model_entity_number" may be used to for model entity identification. If necessary, both may be used.
model_entity_type: type	This is the entity type (an integer) of the given model entity. A suggested convention for entity types of a geometric model is: 0=vertex, 1=edge, 2=face, 3=shell, 4=region, 5=vertex use, 6=edge use, 7=face use

4.2 Interface Operators

The input/output files are useful for the initial creation of attributes and to store the definitions for use in a later working session. However, it is also desirable to have a set of operators that can be invoked to create, manipulate, and query for attribute information on an as needed basis. An operator is a procedure designed to carry out a specific task when given a set of instructions. The user is shielded from the inner workings and data structure of a program. Communication is done solely through the argument variables of the operator. This "blackbox" philosophy alleviates the user or other programs from the burden of knowing the nuts and bolts of a code on the one hand and allows greater versatility for use and access of data on the other. An effective operator needs to be general enough to be useful for a relatively large audience, yet it also must be specific enough to carry out the assigned tasks. This section describes four sets of operators callable from C routines: 1) Set-up/wrap–up, 2) creation, 3) interrogation, and 4) manipulation. The first set of routines are used for the beginning and ending of an attribute manager session. The second set of routines are used to create the specified attributes. The third set of routines are used to query information from the attribute manager. The final set of routines are used to make changes in the definition of the existing attributes. For quick referencing, a summary of these operators is listed in Appendix ?. The following naming convention is used:

- ☐ normal-font() = routine name
- ☐ *italic*() = routine that has a return value of the given type
- ☐ **bold** = input parameter
- ☐ *italic* = output parameter
- ☐ ***bold-italics*** = input and output parameter

4.2.1 Attribute Manager Set-up/Wrap-up

This section describes the requirements to invoke and start the attribute manager. The very first step to using the attribute manager is to set the environment variable that specifies the path where the attribute home directory is. To do so, simply type at prompt

```
setenv ATT_HOME "<the full path to the attribute directory>"
```

Those routines using any of the interface operators must include the following header in the calling routine:

```
$ATT_HOME/attribute/include/header.h
```

The first operator described in this section must be invoked before any attribute operators may be called. This routine allocates memory for the appropriate variables and sets up the data structure for the creation and manipulation of attributes. To start a SAM session, simply call **AT_setup**(*filename*) in the application program. The attribute manager is activated from this point forward. To end the SAM session, call **AT_stop**(*filename*) at any point on the application program when the attribute manager is no longer needed. The following is a detailed description of the operators.

1. void AT_setup(char ***filename**)

   This routine allocates and initializes the variables needed by the attribute manager. This operator also checks to see if *filename.lib*, *filename.org*, and *filename.rel* exist. If so, the contents of the files are read into the attribute manager. Attributes are created, organization structure defined, and association performed as defined in these attribute files. If only new attributes are intended to be created, care must be taken to first remove these files before calling AT_setup() to avoid duplication of attributes. If no input file is found, a message is echoed to the screen noting that nothing is read in initially. A file with the name *filename.ATerr* is created, where any warning and error messages are written. The present restriction is that if one of the three files is present, then all three must be present.

   ☐ **filename**: char *

   /* *The name to be used for input files; it is also used for the error file,* **filename.ATerr**, *where all warning and error messages are written.*

2. void AT_stop(char ***filename**)

   This routine writes to file everything that the attribute manager has in memory, including the attribute definitions, organization structure, and attribute associations. The files to which these attributes are written are *filename.lib*, *filename.org*, and *filename.rel*. Any files with the same names are overwritten.

☐ **filename**: char *

/ The name to be used to write out all attribute information.*

4.2.2 Attribute Creation Operators

This section describes the interface operators that can be called by application programs to create all the necessary information for an attribute. Just as the three major types of information can be categorized as physical, organizational, and relational information, so the attribute creation operators are also described in like categories. One way of using these operators is to call **AT_cratt()** to create a new attribute. The pointer returned by this operator may be used to define the remaining physical information, organizational, and relational information. Given the pointer to the attribute, **tensor_order()** may be used to specify the order of the tensor. The user may then call **coord_sys_ptr()** to attach a new (or previously defined) coordinate system to the attribute. With the returned coordinate system pointer, one may call **csys_info()** and **csys_func()** to specify the coordinate system information. The functions specifying the coordinate system can be parsed with **parser()** before invoking **csys_func()** and **csys_info()**. In the same way, one may call **symmetry_ptr()** to attach symmetry to the attribute and call **symm_info()** and **symmetry_type()** to complete the attribute symmetry.

As for the distributions, there are three ways to create distributions. The operator **parser()** may be used to create "global" distributions as dependencies in other distributions, **AT_mktencomp()** may be used to create and attach a distribution to the next tensor component, and **AT_mklist()** may be used to create and attach local distributions to an attribute. Finally, after the physical information of an attribute are all defined, one may call **AT_bultorgn()** to build the organization tree and **AT_astopat()** to associate this attribute to the appropriate model entities.

4.2.2.1 Physical Creation Operators

1. *ATTRIBUTE *AT_cratt* (char ***attribute_id**)

   This routine creates an attribute with the given id. The pointer to the newly created attribute is returned.

   ☐ *AT_cratt*(): ATTRIBUTE *

   /* *The pointer to the newly created attribute is returned here.*

   ☐ **attribute_id**: char *

   /* *This must be a unique character string that identifies the attribute.*

2. *MULTIPLIER *AT_crmult* (char ***multiplier_id**, double **multiplier_value**)

   This routine creates a multiplier with the given id and value. The pointer to the newly created multiplier is returned.

   ☐ *AT_crmult*(): MULTIPLIER *

   /* *The pointer to the newly created multiplier is returned here.*

   ☐ **multiplier_id**: char *

   /* *A character string that uniquely identifies the multiplier.*

   ☐ **multiplier_value**: double

   /* *A double precision value that the multiplier is to have.*

3. void tensor_order (long **retr_store_flag**, ATTRIBUTE ***attribute_pointer**,

   long **tensor_order*)

   This routine retrieves/stores the tensor order information, dependent on what **retr_store_flag** is.

   - ☐ **retr_store_flag**: long

     / 1–retrieve, 2–store*

   - ☐ **attribute_pointer**: ATTRIBUTE *

     / pointer to the attribute*

   - ☐ *tensor_order*: long *

     / order of the tensor*

4. void coord_sys_ptr (long **retr_create_flag**, ATTRIBUTE ***attribute_pointer**,

   COORDINATE_SYS **coord_sys_ptr*)

   This routine retrieves the pointer to the coordinate system used by the given attribute, pointed to by attribute_pointer. If the command is to create, a pointer is assigned to a new coordinate system and returned to the calling routine. The calling routine is responsible to call **csys_info**() and **csys_func**() to complete the definition of the new coordinate system.

   - ☐ **retr_create_flag**: long

     / 1–retrieve, 2–create new coordinate system*

   - ☐ **attribute_pointer**: ATTRIBUTE *

     / pointer to the attribute*

   - ☐ *coord_sys_ptr*: COORDINATE_SYS *

     / pointer to coordinate system*

5. void csys_func (long **retr_store_flag**, COORDINATE_SYS ***coord_sys_pointer,**
DISTRIBUTION ***func_relation*, long *func_position*,
char ***func_string*)

This routine retrieves or stores the pointers to the current function that defines the coordinate system with respect to the given base coordinate system. If the *retrieve* option is selected, the function returned is the next available function or a NULL if the end of the function list is reached. If the *store* option is selected, the next coordinate system function is stored to the data structure.

☐ **retr_store_flag**: long

/ 1–retrieve, 2–store*

☐ **coord_sys_pointer**: COORDINATE_SYS *

/ pointer to the coordinate system*

☐ *func_relation*: DISTRIBUTION **

/ Retrieval: returns the next available distribution — up to the dimension of the coordinate system. // Store: stores this distribution as the next basis function for the coordinate system pointed to by* **coord_sys_pointer**. *Although memory is allocated in this routine, memory deallocation for this variable will be handled with another routine that is yet implemented.*

☐ *func_position*: long *

/ Retrieve: input of NULL = get the first function; "a number" = get the next available function after this one; output of NULL = no more function to retrieve. // Store: NULL = first function and return the number "one"; "a number": go to next function and return the incremented number.*

☐ *func_string*: char **

/ Retrieve: This is the function in the same character string form as before the function was parsed. // Store: This is a dummy routine that is not used.*

6. void csys_info (long **retr_store_flag**, COORDINATE_SYS ***coord_sys_pointer*,

   long **coord_sys_type*, float ***coord_origin*,

   long **coord_sys_dimension*, char ***coord_sys_id*,

   COORDINATE_SYS ***base_coord_sys_pointer*,

   DISTRIBUTION ****func_ptr*)

Given the particular pointer to the coordinate system, this routine returns/stores the following information from/into the attribute data base:

o Coordinate system type

o Origin coordinates of the current coordinate system

o Dimension of the coordinate system

o Coordinate system ID

o Pointer to the base coordinate system

o Pointer to an array of distribution functions of coordinate axes

☐ **retr_store_flag**: long

/ 1–retrieve, 2–store*

☐ *coord_sys_pointer*: COORDINATE_SYS *

/ pointer to coordinate system*

☐ *coord_sys_type*: long *

/ 1–linear, 2–curvilinear, 3–model*

☐ *coord_origin*: float coordinate_origin[*coord_origin_size*]

/ coordinates of the origin of the coordinate system with respect to the base coordinate system. For retrieval, a pointer should be used as input. Although memory is allocated in this routine, memory deallocation for this variable will be handled with another routine that is yet implemented.*

- ☐ *coord_dimension*: long *

  / dimension of the coordinate system*

- ☐ *coord_sys_id*: char **

  / label that uniquely identifies the coordinate system*

- ☐ *base_coord_sys_pointer*: COORDINATE_SYS *

  / pointer to the base coordinate system*

- ☐ *func_ptr*: DISTRIBUTION ***func_ptr

  / an array of distributions describing the basis functions of the coordinate system. The difference between this and "func_relation" of* **csys_func** *is that this handles all of the coordinate system functions at one time, whereas the previous routine takes the functions one at a time. The address of a variable declared to be DISTRIBUTION ** should be used as input. Although memory is allocated in this routine, memory deallocation for this variable will be handled by another routine that is yet implemented.*

7. void symmetry_ptr (long **retr_store_flag**, ATTRIBUTE ***attribute_pointer**, SYMMETRY ***symmetry_pointer*)

This routine retreives the pointer to the symmetry information or stores the information into the data base. Note, this routine must be called first (before any calls to the other symmetry related creation operators) since memory for each new symmetry is created here.

- ☐ **retr_store_flag**: long

  / 1–retrieve, 2–create new symmetry*

- ☐ **attribute_pointer**: ATTRIBUTE *

  / pointer to the attribute*

- ☐ *symmetry_pointer*: SYMMETRY **

  / Retrieve: input of NULL = get the first symmetry, pointer location = get the next available symmetry after this one, output of NULL = no more symmetry to retrieve. // Store: NULL = start a new linked list and return the address of the first link, pointer location = create new linked list element and link to pointer location and return this new pointer.*

8. void symmetry_type (long **retr_store_flag**, SYMMETRY **symmetry_pointer*,
   long **symmetry_type*)

   This routine stores/retreives the symmetry type to/from the data base.

- ☐ **retr_store_flag**: long

  */*1–retrieve, 2–store*

- ☐ *symmetry_pointer*: SYMMETRY *

  */*Pointer to the symmetry*

- ☐ *symmetry_type*: long *

  */*1–symmetric, 2–totally symmetric, 3–antisymmetric, 4–totally antisymmetric, 5–plane symmetry, 6–line symmetry, 7–point symmetry, 8–plane-line symmetry*

9. void symm_info (long **retr_store_flag**, SYMMETRY **sym_pointer,* long *lrflag*,

   SYMMETRY_INFO **lr_sym_pointer*, float **lr_sym_info*)

   This routine stores/retrieves the information for a single symmetry into/from the data base. The implemented routine has not been tested.

   ☐ **retr_store_flag**: long

   / 1–retrieve, 2–store*

   ☐ *sym_pointer*: SYMMETRY *

   */*Pointer to the symmetry linked list.*

   ☐ *lrflag*: long

   / 1–left-hand-side symmetry, 2–right-hand-side symmetry*

   ☐ *lr_sym_pointer*: SYMMETRY_INFO *

   / Pointer to the symmetry information linked list. Retrieve: input of pointer to NULL = first piece of symmetry information, pointer location = get the next available piece of symmetry information after this one, output of pointer to NULL = no more symmetry information to be retrieved. // Write: NULL = start a new linked list and return the address of first link, pointer location = create new linked list element and link to pointer location and return the new pointer.*

   ☐ *LR_SYM_INFO*: float *

   / variable contains information pertaining to the particular symmetry type*

   types 1–4: index numbers

   types 5, 6, 8:

   LHS: angle of symmetry, where applicable

   RHS: axis or axis plane of symmetry, where applicable *type 7: no information needed, since assuming point coincides with the origin of the coordinate system.*

10. void dist_comp (long **retr_store_flag**, ATTRIBUTE ***attribute_pointer**,

struct comp_distrib ***distrib_comp_pointer*)

This routine gets the pointer to the distribution of the next tensor component.

☐ **retr_store_flag**: long

/ 1–retrieve, 2–create new coordinate system*

☐ **attribute_pointer**: ATTRIBUTE *

/ pointer to the attribute*

☐ *distrib_comp_pointer*: struct comp_distrib *

/ pointer to the distribution component of the tensor: Retrieve: input of NULL = get the first component, pointer location= get the next available component after this one, output of NULL = no more component to retrieve. // Store: NULL = start a new linked list and return the address of first link, pointer location = create new linked list element and link to pointer location and return the new pointer.*

11. *DISTRIBUTION *parser* (long **input_flag**, void ***input_source**,

ATTRIBUTE ***attribute_pointer**, long **store_flag**)

Given an equation in character string form, this routine parses the equation and returns the parsed equation as a distribution.

☐ *parser*(): DISTRIBUTION *

/ A pointer to the distribution that is created from the given equation.*

- ☐ **input_flag**: long

  /* *A flag that indicates whether the source is a string or an input file: 1= input file and 2= character string.*

- ☐ **input_source**: void *

  /* *If* **input_flag** *were 1, then this should be a pointer to an input file from which the distribution is to be read. The file position should be set to immediatly before the equation, for example, using a combination of fseek() and getc(). If* **input_flag** *were 2, then this should be a character string containing the equation that is to be parsed.*

- ☐ **attribute_pointer**: ATTRIBUTE *

  /* *If the equation to be parsed contains references to other distributions defined locally in an attribute, the pointer to that attribute needs to be specified here.*

- ☐ **store_flag**: long

  /* *Specify the manner of storage of the distributions.*

  0 = read from a distribution unit, marking the distributions created as "global" distributions. After parsing one equation, keep on parsing until there are no more equations to be parsed.

  1 = read from a distribution unit, marking the distribution created as "global" distribution. Parse only the next distribution.

  2 = the equation given is that of a coordinate system function. The distribution is to be separated from the global distribution.

  3 = read from a distribution list, marking the distributions created as *"local" distributions.*

12. void AT_mktencomp (char ***input_equation**, ATTRIBUTE ***attribute_pointer**)

    This routine parses the given equation and attaches the resulting distribution to a new tensor component of the given attribute. No provision is made to distinguish which component of the tensor is being created. The order that **AT_rtdstcomp()** returns the tensor components is the order in which the input equations are given in this routine.

    ☐ **input_equation**: char *

    / The equation that is to be parsed and inserted as a tensor component of the given attribute.*

    ☐ **attribute_pointer**: ATTRIBUTE *

    / Pointer to the attribute whose next tensor component were to defined with the created distribution.*

13. void AT_mklist (ATTRIBUTE ***attribute_pointer**, char ****equations**, int **num_eqns**)

    Given an array of equations, this routine parses each equation and attaches the parsed distribution to a list, which is attached to the attribute given in attribute_pointer.

    ☐ **attribute_pointer**: ATTRIBUTE *

    / Pointer to the attribute to which the list is to be attached.*

    ☐ **equations**: char **

    / An array of equations in character string form. These equations are parsed and stored as distributions.*

☐ **num_eqns**: integer

/* *The number of equations passed in.*

14. void AT_mktmplat (ATTRIBUTE ***attribute_pointer**, char ****equations**,

int **num_eqns**)

Given a pointer to an attribute and a set of defining equations, this routine creates and assigns distributions, tensor order, and symmetries using a previously defined template. The appropriate template is selected based on the specific label and attribute type on the given attribute. This implies that the appropriate labels (and coordinate system) and type are assigned prior to the calling of this routine. What has been implemented as of this writting is only a set of stiffnesses for the following materials: **LOEM** for Linear Orthotropic Elastic Material and **LIEM** for Linear Isotropic Elastic Material. To use this template operator, the first label of the attribute must be one of the two keywords just mentioned. Also, the attribute type must be **stiffness**.

☐ **attribute_pointer**: ATTRIBUTE *

/* *Pointer to the attribute for which the template is to be instansiated.*

☐ **equations**: char **

/* *An array of equations in character string form. The assumption in calling this routine to instance the template is that the set of tensorial components of this attribute are dependent upon the set of equations that are passed in as argument. For example, to use the stiffness template for isotropic elastic material, the calling routine must pass in values for E and nu, as shown in Figure 26. The*

Figure 26 Input example of *equations* and *num_eqns* (Continued) . . .

```
equations[0] = "E=3000000";
equations[1] = "nu=0.33";
num_eqns = 2;
AT_mktmplat( attribute_ptr, equations, num_eqns );
```

Figure 26 Input example of *equations* and *num_eqns*

corresponding equations for orthotropic materials are E1, E2, E3, nu12, nu21, etc.

☐ **num_eqns**: integer

/* *The number of equations passed in.*

4.2.2.2 Organizational Creation Operators

1. *SET *AT_crset* (char ***set_id**)

   This routine creates a set with the given id. The pointer to the newly created set is returned.

   ☐ *AT_crset*(): SET *

   /* *The pointer to the newly created set is returned here.*

   ☐ **set_id**: char *

   /* *A character string that uniquely identifies the set.*

2. *GROUP *AT_crgrp* (char ***group_id**)

   This routine creates a group with the given id. The pointer to the newly created group is returned.

   ☐ *AT_crgrp*(): GROUP *

   /* *The pointer to the newly created group is returned here.*

   ☐ **group_id**: char *

   /* *A character string that uniquely identifies the group.*

3. *CASE *AT_crcas* (char ***case_id**)

This routine creates a case with the given id. The pointer to the newly created case is returned.

☐ *AT_crcas*(): CASE *

/ The pointer to the newly created case is returned here.*

☐ **case_id**: char *

/ A character string that uniquely identifies the case.*

4. void AT_bultorgn (char ***parent_id**, ORGNCODE **parent_type**, char ***child_id**,
   ORGNCODE **child_type**, char ****multitplier_id**,
   double ***multiplier_values**, int **num_mults**)

This routine builds a segment of the organizational structure with the given parent, child, and the multipliers to the child.

☐ **parent_id**: char *

/ A character string that uniquely identifies the parent node. If there are no node with such id, a new node of the type "parent_type" is created.*

☐ **parent_type**: ORGNCODE

/ The type of the parent node. The parent types are restricted to ATSET, ATGROUP, and ATCASE.*

☐ **child_id**: char *

/ A character string that uniquely identifies the child node. If there are no node with such id, a new node of the type "child_type" is created for all types except for child type of attribute.*

- ☐ **child_type**: ORGNCODE

  /* *The type of the child node. The child types are restricted to ATATT, ATSET, and ATGROUP.*

- ☐ **multiplier_id**: char **

  /* *A 1–D character string array of multiplier ids. If the multipliers had not been defined previous to the calling of this routine, the value of this multiplier must be specified in the corresponding array slot in multiplier_values. A new multiplier with such value is then created. If the multiplier had been defined previously, a value is not needed. This argument can be a NULL if no multiplier were defined for this segment.*

- ☐ **multiplier_values**: double *

  /* *An array of double precision multiplier values. As was detailed above, these numbers are used only in the event where the corresponding multiplier specified in multiplier_id was never defined previously. If no value were found for such a case, a default value of 1 is used for the multiplier that is being defined. The size of mutplier_values and multiplier_id should be the same. This argument can be NULL if no multiplier were defined for this segment.*

- ☐ **num_mults**: integer

  /* *This number specifies the number of multipliers are actually passed in. If zero, the two arguments, multiplier_values and multiplier_id, can both be NULL. Otherwise, they must have the same size as num_mults.*

4.2.2.3 Relational Creation Operators

For the moment, only one relational creation operator has been implemented:

void AT_astopat (ATTRIBUTE ***attribute_pointer**, long **model_entity_number**,
char ***model_entity_string**, long **model_entity_type**,
char ***model_name**)

This function associates a given model entity with the attribute to which the given pointer points. It takes as input a pointer to the attribute, the entity identification in numerical or string form, the entity type, and the model identification.

☐ **attribute_pointer**: ATTRIBUTE *

/ Pointer to the attribute that is to be applied toward the model entity.*

☐ **model_entity_number**: long

/ The model entity in numerical form to which the attribute is to be applied.*

☐ **model_entity_string**: char *

/ The model entity in character string form to which the attribute is to be applied.*

☐ **model_entity_type**: long

/ A flag that indicates the type of the model entity. For example, if the entity is part of a geometric model, a suggested convention is 1=vertex, 2=edge, 3=face, 4=shell, 5=region, -1=vertex use, -2=edge use, -3=face use*

☐ **model_name**: char *

/ A character string identifier of the model being used.*

4.2.3 Attribute Interrogation Operators

The interrogation operators are called to query and retrieve attribute information from the manager. Similar to the attribute creation operators, the attribute interrogation operators are categorized into attribute retrieval operator, organization retrieval operators, and attribute physical information retrieval operators. One way to use the operators described in this section is to call **AT_rtatts()** to retrieve pointers to all the attributes that satisfy a given set of relational and organizational conditions (such as all attributes from a specified organizational branch that were applied on a particular model entity edge of a particular geometric model). Along with the pointers returned are also a set of relevant information (mostly relational) about the retrieved attributes (such as model entity type, attribute type, the product of all the multipliers relevant to the searched organizational branches, etc.). To find the list of available attributes, **AT_rtatts()** searches through only the branches of the organizational tree specified by the calling routine (for example, the branch might be of a set called *temperature* of the first group of *loading_condition* in the case of *thermal_analysis*). One important note is that **AT_rtatts()** always begins the search at the CASE level. Any attributes not attached directly or indirectly to a case is not searched. If a set of attributes is found, they are compared to the given relational criteria to find the desired attributes.

If the calling routine does not know in advance the exact shape of the organizational structure, the organizational operators can be used to systematically traverse the attribute organization structure. **AT_rtnxorgn()** traverses the organizational structure horizontally, while **AT_rtnxchild()** allows the calling routine to traverse the organizational structure vertically (please see Figures 19 and 20 for reference). The pointers retrieved from **AT_rtnxorgn()** can be used as input to **AT_rtnxchild()** to get the next child. In addition, at any time of traversal the calling routine can use **AT_rtnodeid()** to retrieve the id of the organizational element (traction and temperature, for example). These routines can be used in conjunction with **AT_rtatts()** to systematically retrieve all the desired attributes.

For multipliers, both **AT_rtmprod()** and **AT_rtmults()** retrieve multipliers between two specified connected nodes. The difference is that the former returns the product of all the multipliers, while the latter returns all the individual multiplier ids and values between these nodes.

Once all the attributes that satisfy the given organizational and relational conditions are retrieved, the physical information of the attributes can be retrieved through the physical information retrieval operators. Each of the operators (with the exception of **AT_rtcsysinfo()**) uses the attribute pointers as a means to identify the attribute whose information is to be retrieved. The physical information include information on the labels, type, tensor order, coordinate system, symmetry, and distribution of the attribute. **AT_rtcsysinfo()** retrieves information on the coordinate system when given either the id of the coordinate system or the pointer to the coordinate system.

Again, the naming convention is reiterated here:

- normal font () = operator name
- **bold** = input
- *italic* = output
- ***bold-italic*** = the input might be modified by the output value

4.2.3.1 Attribute Retrieval Operator

Listed in this section is the operator to retrieve pointers to attributes that satisfy a set of searching criteria.

void AT_rtatts (long **retrieval_method**, char ***case**, char ***group**, char ***set**,

char ***geom_model_name**, long **model_entity_type**, long **model_entity_integer**, char ***model_entity_string,**

char ***attribute_type**, long **array_size,**

ATTRIBUTE ***attribute_pointer**, double **mult_value*,

long **number_retrieved*)

This routine returns pointers to attributes which satisfy the organizational and relational condition specified in the input. Any information specified in the input (arguments third through tenth) are used to narrow down the selection of available attributes. The third, fourth, and fifth arguments are used to restrict the search to the given branches in the organizational structure. The sixth through tenth arguments are used to pin-point the desired attributes in the already shrunken selection of qualified attributes. NOTE: This routine only searches the complete links of the organizational structure, beginning with the CASE. If only a particular SET of attributes are sought, for example, the calling routine must use **AT_nxchild()** instead.

At the time of writing, this routine has not been tested. However, another routine named **attptr_C()** had been used extensively for attribute retrieval. **AT_rtatts()** and **attptr_C()** are essentially the same with one major difference: **attptr_C()** returns (geom_model_name, model_entity_type, model_entity_integer, model_entity_string, and attribute_type) in addition to (attribute_pointer and mult_value). This added feature is desirable if not for one important flaw in the return logic. When an attribute is associated with more than one model entity, only one model entity is returned. The same flaw occurs for both the geom_model_name and model_entity_type. Because of this flaw, another operator, **AT_rtatts()**, is developed that returns only the attribute_pointers and mult_value.

☐ **retrieval_method**: long

/ 1: retrieve pointers to all attributes*

2: retrieve the pointer to the next attribute

☐ **case**: char *

/ If the string is empty or if the* **case** *is pointing to NULL (no case given), AT_rtatts() assumes no particular case information was wanted, i.e., it searches through all available cases.*

☐ **group**: char *

/ If the string is empty or if the* **group** *is pointing to NULL (no group given), AT_rtatts() assumes no particular group information wanted, i.e., it searches through all available groups.*

☐ **set**: char *

/ If the string is empty or if the* **set** *is pointing to NULL (no set given), AT_rtatts() assumes no particular set information wanted, i.e., it searches through all available sets.*

Essentially, the routine traverses only the branches indicated. If none of the case, group, or set are specified, it searches through all the available attributes in memory. Take the organizational structure as defined in Figure 27, for example. Attr1 and Attr2 are returned from AT_rtatts() when it is given an input of **case** *= Case1,* **group** *= Group1, and* **set** *= Set1. For the sake of discussion, the input and output written in short hand is (Case1, Group1, Set1) = (Attr1, Attr2). Other examples are: (Case2, NULL, Set4) = (Attr5). (Case2, NULL, NULL) = (Attr3, Attr4, Attr5), which in effect is all the attributes under Case2. (NULL, NULL, Set3) = (Attr4, Attr5). (Case1, Group2, Set4) = (NULL), since Group2 and Set4 are not connected. On the*

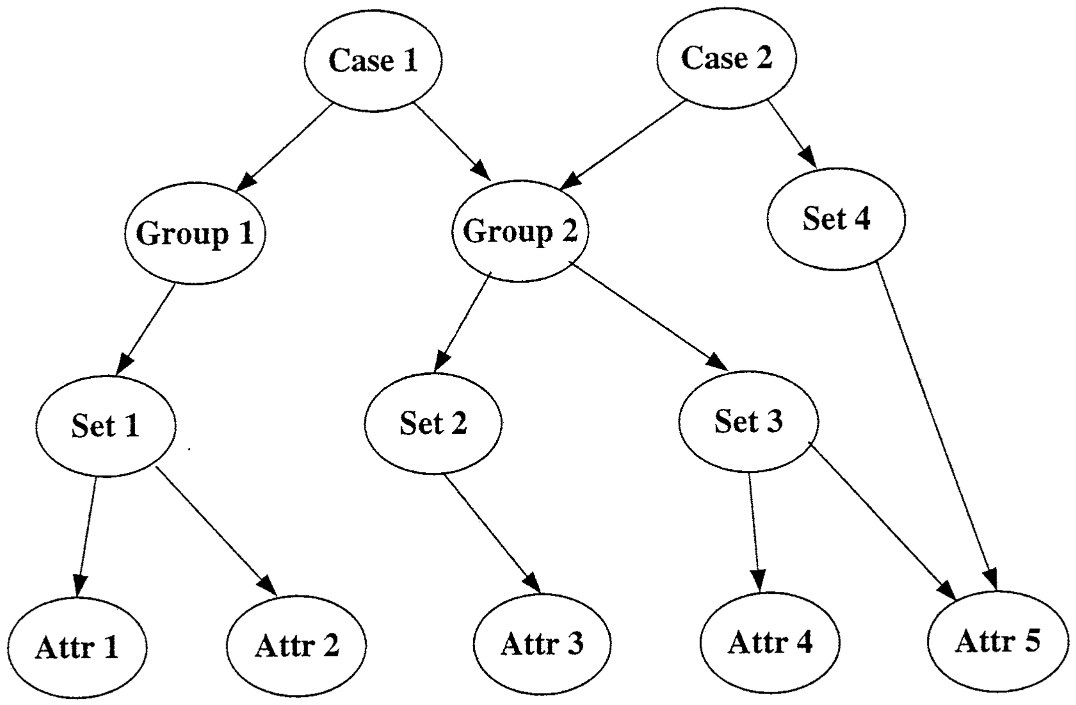


Figure 27 A Sample Organizational Structure

other hand, (NULL, NULL, NULL) = (Attr1, Attr2, Attr3, Attr4, Attr5), that is, all attributes defined in all cases.

- ☐ **geom_model_name**: character *

  / This argument identifies the geometric model the specified entity uses. If the string is empty (pointer to NULL), AT_rtatts() searches through all geometric models.*

- ☐ **model_entity_type**: long

  / This variable can be 0 = vertex, 1 = edge, 2 = face, 3 = shell, 4 = region, 5 = vertex use, 6 = edge use, or 7 = face use.*

  If this is set to –1, AT_rtatts() does not use this as a comparison criterion.

- ☐ **model_entity_integer**: long

  / This variable needs to be an integer that uniquely identifies the model entity. If this is set to –1, AT_rtatts() does not use this as a comparison criterion.*

- ☐ **model_entity_string**: char *

  / This variable needs to be a string that uniquely identifies the model entity. If both this and the* **model_entity_integer** *are specified, attributes that satisfy either one or both of these conditions are returned.*

- ☐ **attribute_type**: char *

  / This variable needs to be the attribute type to be used as a searching criterion. Some examples are:*

  — *Fluid_velocity*

  — *Traction*

  — *Temperature*

  — *label*

  If this string is empty (pointing to NULL), AT_rtatts() does not use the attribute type as a comparison criterion.

- ☐ **array_size**: long

  / If the calling routine wants this operator to dynamically allocate memory for* ***attribute_pointer*** *and* ***mult_value****, then* **array_size** *should be set to zero as input. Note, it is then the responsibility of the calling routine to free up the memory after it is no longer useful. If the calling routine wants to pass a fixed sized array to be used by the operator, then* **array_size** *should be the size of the array passed in. The calling routine should then check the array size against the number_retrieved to make sure that all eligible attributes are returned.*

- ☐ ***attribute_pointer***: ATTRIBUTE **

  / An input of pointer to NULL signals to get the first available attribute for*

the given criteria. When the **retrieval_method** *is set to get the next attribute, an input of a pointer indicates to get the next attribute after the given attribute pointer. An output of pointer to NULL means there are no more attributes to return. This routine assumes consecutive retrieval of attributes for the option of retrieving the next available attribute.*

A word about the meaning of "next": this operator retrieves all the attributes that satisfy the given organizational and relational condition. There is a specific order of traversal in searching for the qualified attributes. The word "next" is viewed against this order. The order of traversal used in this operator is from right to left of the organizational structure. For the example structure diagramed in Figure 27 above, the order is from Attr5 to Attr1.

☐ *mult_value*: double *

/* *This is a list of final multiplier products corresponding to the attributes stored in the pointer array* **attribute_pointer**. *These values are valid only for the path specified by* **case**, **group**, *and* **set**.

☐ *number_retrieved*: long *

/* *When the* **retrieval_method** *is set to get the next attribute, the number returned here is 1. When the* **retrieval_method** *is set to get all, the number returned here indicates how many are retrieved from the data base. An output of 0 means no attribute was returned.*

4.2.3.2 Organization Retrieval Operators

Listed in this section are the operators to retrieve organizational information from the attribute data base:

1. void AT_rtnxorgn (ORGNCODE **node_type**, void ***node_pointer***)

   This routine returns the next organizational element in the organizational structure with the given organizational element type, such as a case, group, set, or attribute.

   ☐ **node_type**: ORGNCODE

   /* *ATATT–attribute, ATSET–set, ATGROUP–group, ATCASE–case*

   ☐ ***node_pointer***: void **

   /* *Input of NULL = get the pointer to the first of the organizational element of the type* **node_type**; *input of a pointer = get the pointer to the next organizational element; a NULL is returned when there is no more organizational element defined for the type of* **node_type**.

2. void AT_rtnxchild (void ***node_pointer**, ORGNCODE **node_type**,
   void ***child_pointer***, ORGNCODE *child_type*)

   This routine returns the next child pointer and child type of a given node pointer and node type.

   ☐ **node_pointer**: void *

   /* *This is a pointer to the node of interest.*

   ☐ **node_type**: ORGNCODE

   /* *ATSET–set, ATGROUP–group, ATCASE–case*

   ☐ ***child_pointer***: void **

   /* *This is a pointer to the child of the node of interest. Input of a pointer to NULL = get the pointer to the first of the children of NODE_POINTER; input of a pointer = get the next child; a NULL is returned when there are no more children defined for the type of NODE_TYPE.*

☐ *child_type*: ORGNCODE

/ ATATT–attribute, ATSET–set, ATGROUP–group*

3. *ATTRIBUTE *AT_rtatt* (char ***attribute_id**)

This routine retrieves the pointer to the attribute that has the same id as given.

☐ *AT_rtatt*(): ATTRIBUTE *

/ The pointer of the desired attribute is returned here. A NULL is returned if no attribute with the given id is found.*

☐ **attribute_id**: char *

/ A unique identifier of the attribute being enquired.*

4. *SET *AT_rtset* (char ***set_id**)

This routine retrieves the pointer to the set that has the same id as given.

☐ *AT_rtset*(): SET *

/ The pointer of the desired set is returned here. A NULL is returned if no set with the given id is found.*

☐ **set_id**: char *

/ A unique identifier of the set being enquired.*

5. *GROUP *AT_rtgrp* (char ***group_id**)

This routine retrieves the pointer to the group that has the same id as given.

☐ *AT_rtgrp*(): GROUP *

/ The pointer of the desired group is returned here. A NULL is returned if no group with the given id is found.*

☐ **group_id**: char *

/ A unique identifier of the group being enquired.*

6. *CASE *AT_rtcas* (char ***case_id**)

This routine retrieves the pointer to the case that has the same id as given.

☐ *AT_rtcas*(): CASE *

/ The pointer of the desired case is returned here. A NULL is returned if no case with the given id is found.*

☐ **case_id**: char *

/ A unique identifier of the case being enquired.*

7. void AT_rtmprod (void ***node_pointer**, ORGNCODE **node_type**,
void ***child_pointer**, ORGNCODE **child_type**,
double **multiplier_value*)

This routine had not been implemented yet. When it is implemented, this routine will return the product of all the multipliers between two given connected nodes in the organizational structure. Hierarchically, the node must be of a type higher than that of the child.

☐ **node_pointer**: void *

/ This is a pointer to the node whose multipliers are of interest.*

☐ **node_type**: ORGNCODE

/ ATSET–set, ATGROUP–group, ATCASE–case*

☐ **child_pointer**: void *

/ This is a pointer to the child of the node between whose multipliers are of interest.*

- ☐ **child_type**: ORGNCODE

  /* *ATATT–attribute, ATSET–set, ATGROUP–group*

- ☐ *multiplier_value*: double *

  /* The product of all the multipliers between the node and the child.

8. void AT_rtmults(void ***node_pointer**, ORGNCODE **node_type**,
   void ***child_pointer**, ORGNCODE **child_ type**,
   long ****number_of_multipliers***, char ***multiplier_id*,
   double **multiplier_value*)

   This routine had not been implemented yet. When it is implemented, this routine will return the identification and value of all the multipliers between two given connected nodes in the organizational structure. Hierarchically, the node must be of a type higher than that of the child.

- ☐ **node_pointer**: void *

  /* *This is a pointer to the node whose multipliers are of interest.*

- ☐ **node_type**: ORGNCODE

  /* *ATSET–set, ATGROUP–group, ATCASE–case*

- ☐ **child_pointer**: void *

  /* *This is a pointer to the child of the node between whose multipliers are of interest.*

- ☐ **child_type**: ORGNCODE

  /* *ATATT–attribute, ATSET–set, ATGROUP–group*

- ☐ ***number_of_multipliers***: long *

  /* *As input, this number indicates the maximum array size allocated for* *the multiplier_id and multiplier_value. Returned here is the actual number of multipliers retrieved. Again, the calling routine should check to see if the number retrieved is not greater than the maximum array size.*

- ☐ *multiplier_id*: character string array

  / The character string identification of the multipliers found between the given node and the child. This must be an array of size number_of_multipliers.*

- ☐ *multiplier_value*: double precision array

  / The value of the multiplier corresponding to the multipliers in multiplier_id. This must be an array of size number_of_multipliers.*

9. void AT_rtnodeid (void ***node_pointer**, ORGNCODE **node_type**, char ***node_id*)

   This routine returns the identification of a node pointed to by the given node pointer of the given node type. For example, if node = attribute (**node_type** = ATATT), **node_pointer** = ATTRIBUTE_POINTER.

- ☐ **node_pointer**: void *

  / This is a pointer to the node of interest.*

- ☐ **node_type**: ORGNCODE

  / ATATT–attribute, ATSET–set, ATGROUP–group, ATCASE–case*

- ☐ *node_id*: char *

  / The character string identification of the node. Note: the user is responsible to free the memory after node_id is used.*

4.2.3.3 Attribute Physical Information Retrieval Operators

Listed in this section are the operators useful for retrieving physical information of a given attribute:

1. *char *AT_rtatid* (ATTRIBUTE ***attribute_pointer**)

   This routine retrieves the id of the attribute pointed to by the given attribute pointer.

   ☐ *AT_rtatid*(): char *

   / A character string of a unique identifier for the given attribute. Note: the user is responsible to free the memory when AT_rtatid is no longer needed.*

   ☐ **attribute_pointer**: ATTRIBUTE *

   / This is a pointer to the attribute of interest.*

2. void AT_rtlabels (ATTRIBUTE ***attribute_pointer**, long ****number_of_labels***,
   char ***attribute_labels*)

   This routine returns all of the labels of the attribute pointed to by the given attribute pointer.

   ☐ **attribute_pointer**: ATTRIBUTE *

   / This is a pointer to the attribute of interest.*

   ☐ ***number_of_labels***: long *

   / As input, this number indicates the maximum array size allocated for the attribute_labels. Returned here are the actual number of labels retrieved. Again, the calling routine should check to see if the number retrieved is not greater than the maximum array size.*

   ☐ *attribute_labels*: char **

   / An array of character string labels of the given attribute. Note: the user is responsible to free the memory when attribute_labels is no longer needed.*

3. *char *AT_rtnxatlab* (ATTRIBUTE ***attribute_pointer,**

   Generic_List_Cells *****label_pointer***)

   This routine retrieves the next label of the given attribute. The id of the attribute is excluded from the definition of a "label."

   ☐ *AT_rtnxatlab*: char *

   / The label is returned here as a character string. A NULL is returned if there are no more labels. Note: the calling routine must free up the memory after the information is no longer needed.*

   ☐ **attribute_pointer**: ATTRIBUTE *

   / This is a pointer to the attribute of interest.*

   ☐ ***label_pointer***: Generic_List_Cell2 *

   / For the first call to this routine, a NULL should be used to get the very first label. The returned pointer should be used for the next call to this routine to get the next label.*

4. void AT_rtatype (ATTRIBUTE ***attribute_pointer**, char ***attribute_type*)

   This routine retrieves the attribute type of the given attribute.

   ☐ **attribute_pointer**: ATTRIBUTE *

   / Pointer to the attribute*

   ☐ *attribute_type*: char **

   / attribute type. Note: the user is responsible to free the memory when attribute_labels is no longer needed.*

5. void AT_rtenord (ATTRIBUTE ***attribute_pointer**, long **tensor_order*)

   This routine returns the tensor order of the attribute pointed to by the given attribute pointer. The name being used now is **tensord_C**(). This name should be changed in the near future to **AT_rtenord**() so that consistent naming convention may be used.

- ☐ **attribute_pointer**: ATTRIBUTE *

  /* *pointer to the attribute*

- ☐ *tensor_order*: long *

  /* *order of the tensor*

6. void AT_rtcsptr(ATTRIBUTE ***attribute_pointer**,

   COORDINATE_SYS ***coord_sys_pointer*)

   This routine returns the pointer to the coordinate system on which the attribute, pointed to by the given attribute pointer, is defined. The function is currently named as **coordsys_ptr**(). This name should be renamed to **AT_rtcsptr**() in the near future so that consistent naming convention may be used.

- ☐ **attribute_pointer**: ATTRIBUTE *

  /* *pointer to the attribute*

- ☐ *coord_sys_pointer*: COORDINATE_SYS **

  /* *pointer to the coordinate system*

7. void AT_rtcsinfo(COORDINATE_SYS ****coord_sys_pointer***,

   char ****coordinate_sys_id***, long **coord_system_type*,

   long **coord_system_dimension*, float ***coord_origin*,

   struct comp_distrib ***function_relation*,

   COORDINATE_SYS ***base_coord_system_pointer*)

This routine returns the information of the coordinate system pointed to by the coordinate system pointer.

- ☐ ***coord_sys_pointer***: COORDINATE_SYS **

  /* *Pointer to the coordinate system. If the pointer is set to NULL as input,* ***coordinate_sys_id*** *is used as the target of retrieval, in which case the pointer to the coordinate system specified in* ***coordinate_sys_id*** *is returned here.*

- ☐ ***coordinate_sys_id***: char **

  /* *A character string identification of the coordinate system. If the pointer is set to NULL as input,* ***coord_sys_pointer*** *is used as the target of retrieval, in which case the character string identification of the coordinate system pointed to by* ***coordinate_sys_id*** *is returned here. If both* ***coord_sys_pointer*** *and* ***coordinate_sys_id*** *are non-empty, the coordinate system pointed to by* ***coord_sys_pointer*** *is used and the character string stored in* ***coordinate_sys_id*** *is over written with the id of the coordinate system pointed to by* ***coord_sys_pointer***.

- ☐ *coord_system_type*: long *

  /* *1–Linear (both Cartesian and non-orthogonal) , 2–curvilinear, 3–model entity (tangent/ normal)*

- ☐ *coord_system_dimension*: long *

  /* *dimension of the coordinate system*

- ☐ *coord_origin*: float *coordinate_origin[*coord_system_dimension*]

  /* *coordinates of the origin of the coordinate system with respect to the base coordinate system*

- ☐ *function_relation*: struct comp_distrib *function_relation[*coord_system_dimension*]

/ The function relations, which define the coordinate system with respect to the coordinate system pointed to by the base_coord_system_pointer, written as arrays of character strings with an array size equal to the dimension of the coordinate system.*

☐ *base_coord_system_pointer*: COORDINATE_SYS **

/ pointer to the base coordinate system*

8. void AT_rtsyminfo(long **retrieval_method**, ATTRIBUTE ***attribute_pointer**,

    long ****symmetry_type***, long ****info_array*,

    long **num_of_sym_retrieved*, long **num_of_info_pieces*)

This routine has not yet been implemented. When it is, it will return the symmetry information which satisfies the condition set in RETRIEVAL_METHOD.

☐ **retrieval_method**: long

/ 1: retrieve all symmetries of the given symmetry type*

2: retrieve the next symmetry of the given symmetry type

3: retrieve all symmetries

4: retrieve the next symmetry

☐ **attribute_pointer**: ATTRIBUTE *

/ pointer to the attribute*

☐ ***symmetry_type***: long *

/ Type are: 1–symmetric, 2–totally symmetric, 3–antisymmetric, 4–totally antisymmetric, 5–plane symmetry, 6–line symmetry, 7–point symmetry, 8–plane-line symmetry, 9–anisotropic, 10–orthotropic, 11–transversely isotropic, 12–isotropic.*

When the option of getting the next symmetry is set, input of NULL = get the first symmetry, character string = get the next available symmetry after this one, output of NULL = no more symmetries to retrieve.

- ☐ *info_array*: long ***

  / The size of the 3–D array is [num_of_sym_retrieved] × [num_of_info_pieces] × [2]. The num_of_info_pieces is different depending on what the symmetry type is. For symmetry types 1 – 4, the num_of_info_pieces corresponds to the number of symmetric indices, i.e., the first row of [num_of_sym_retrieved] × [num_of_info_pieces] contains indices for the left side of the symmetry and the second row of [num_of_sym_retrieved] × [num_of_info_pieces] contains the indices for the right side of the symmetry. For symmetries 5, 6, and 8, the first row contains the angle of the symmetry and the second contains the coordinate axis/plane number that corresponds to the particular symmetry.*

- ☐ *num_of_sym_retrieved*: long *

  / When the* **retrieval_method** *is set to get the next symmetry, the number returned here is 1. When the* **retrieval_method** *is set to get all, the number returned here indicates how many are returned.*

- ☐ *num_of_info_pieces*: long *

  / The num_of_info_pieces is different depending on what the symmetry type is. For symmetry types 1 – 4, the num_of_info_pieces corresponds to the number of symmetric indices. For symmetry types 5 and 6, the num_of_info_pieces = 1. For symmetry type 8, the value of num_of_info_pieces = 2.*

9. void AT_rtdstcomp (long **retrieval_method**, ATTRIBUTE ***attribute_pointer**,
double **mult_value**, COORDINATE_SYS ***coord_sys_pointer**,
char ****variable_character_array**,
double ***variable_value_array**, long **number_of_var**,
long ****number_of _components***, long **components_defined*,
double **component_value_array*)

This routine returns the final values (in the given coordinate system and with all the relevant multipliers taken into account) of the distribution components of the attribute pointed to by the attribute pointer. All the variables required for evaluation of the distribution are given as input argument in the variable arrays.

☐ **retrieval_method**: long

/* *1: retrieve the value of the distribution of all of the components*

2: retrieve the value of the distribution of the next component

☐ **attribute_pointer**: ATTRIBUTE *

/* *pointer to the attribute*

☐ **mult_value**: double

/* *This is the number with which all the value of the distribution of all components are multiplied.*

☐ **coord_sys_pointer**: COORDINATE_SYS *

/* *A pointer to the coordinate system in which the component distribution is to be calculated. If a NULL is given as input, this routine uses the global coordinate system.*

☐ **variable_character_array**: char **

/ This array contains all the variables needed to qualify the value of the component. For example, if the distribution is spatially dependent, the coordinate variables are listed here with their corresponding values stored in the next array,* **variable_value_array.**

- ☐ **variable_value_array**: double *

  / This array contains the values of all the variables needed to qualify the value of the component distributions. The entries should correspond to the above* **variable_character_array.**

- ☐ **number_of_var**: long

  / The number of variables passed in in the* **variable_character_array** *and* **variable_value_array.**

- ☐ ***number_of_components***: long *

  / When* **retrieval_method** *signals to get all components, this integer indicates the total number of components passed. When the* **retrieval_method** *is set to get the distribution of the next component, this integer indicates which component is to be accessed, starting with 0. An increment of the number is returned if this is not the last one. If this is the last component, a 0 is returned.*

- ☐ *components_defined*: long *

  / When the* **retrieval_method** *is set to get all components, this integer, in binary form, indicates which component is specified and which is not, eg, 11001 means only first, second, and fifth components are specified. When the* **retrieval_method** *is set to get only the distribution of the next component, this integer indicates whether the returned component is specified or not (0 for no*

and 1 for yes). This flag takes care of the ambiguity of whether the components retrieved were not specified or simply have values of zero.

☐ *component_value_array*: double *

/* *When the* **retrieval_method** *signals to get all components, this array contains the values of all the components. Each array element must be checked against the variable, "components_defined." When the* **retrieval_method** *is set to get only the distribution of the next component, this array has a size of 1.*

4.2.4 Attribute Manipulation Operators

The attribute manipulation operators are used to make changes to the definition of the attributes that had been specified using the creation operators. These operators are especially useful as tools for a graphic user interface that allows the user to modify existing attribute definitions. The manipulation functions can be categorized as copy, modify, disassociate, and delete. Only three such operators have been implemented to date. They are described in this section. Again, the naming convention is:

- normal font () = operator name
- *italic*() = operator that returns a value
- **bold** = input
- *italic* = output
- ***bold-italic*** = the input might be modified by the output value

1. void AT_delorgn (ATTRIBUTE ***attribute_pointer**)

   This routine disassociates the organizational specification from the attribute. Because the organizational elements might be used else where, they are not deleted and allocated memory is not freed in this routine. This is done in another deletion routine.

- ☐ **attribute_pointer**: ATTRIBUTE *

  / The pointer to the attribute whose organization specification were to be disassociated must be given as input here.*

2. void AT_delreln (ATTRIBUTE ***attribute_pointer**)

   This routine disassociates all relational specifications from the attribute.

- ☐ **attribute_pointer**: ATTRIBUTE *

  / The pointer to the attribute whose relational specification were to be disassociated must be given as input here.*

3. *ATTRIBUTE *AT_cpyatt* (char ***old_name**, char ***new_name**)

   This routine copies all the components of a given attribute into a new attribute of a new name as given and returns the pointer to the newly created attribute.

- ☐ *AT_cpyatt*(): ATTRIBUTE *

  / The pointer to the newly created attribute is returned here.*

- ☐ **old_name**: char *

  / The name of the attribute that is to be copied*

- ☐ **new_name**: char *

  / The name of the new attribute that is to inherit all the specifications of the attribute identified as* **old_name**.

5 REPAS INTEGRATION

As introduced in the first chapter of this thesis, the REPAS project consists of five major analysis modules. Together, the five modules analyze the thermal, thermomechanical, and electromagnetic properties of a given MCM. Each of the five modules employs a unique analysis scheme. The technical challenge of this project is that all five analyses must be seamlessly integrated. The integration effort entails satisfying all input and output information for, and data transfer between, the client analyses. In all, two physical scales of analysis (global averaged layer-wise representation of MCM and detailed representation of particular sections of the interconnect) and three different analysis methodologies (global variational approximation approach, fast random walk technique, and adaptive finite element analysis) are involved.

A set of criteria guided the REPAS integration effort. The analysis models must be be derived from a single physical definition, and the analysis model idealization and domain discretization must be automated. Furthermore, such integration requires that the analyses can efficiently communicate with each other without any information loss or duplication. The information must be consistently defined and must be easily accessible. The set of information required for the analyses must also be flexible in that new information required can be easily integrated into the analysis process.

The SCOREC Attribute Manager (SAM) detailed in the first half of this thesis is designed to be the framework that provides for such data coordination for the client analyses. The previous chapters describe the design, implementation, and application interface operators of this manager. Recall Figure 2, which is reproduced here as Figure 28, what has been discussed thus far are the necessary components shown on the right

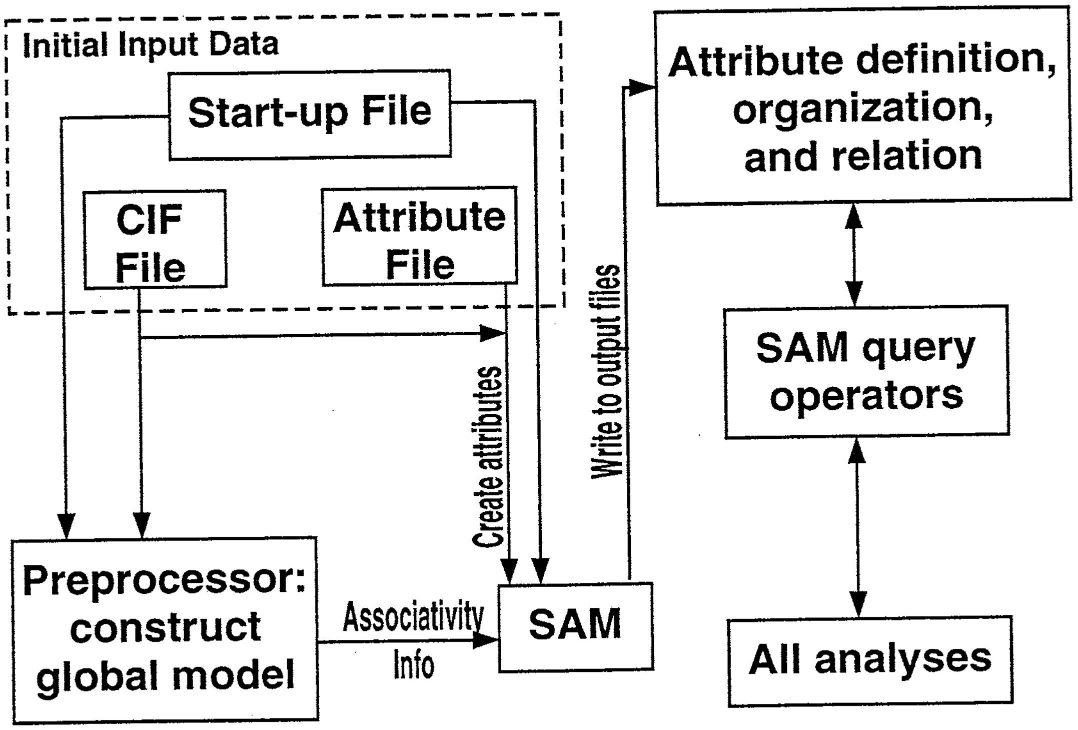


Figure 28 Schematic of of SAM as the Heart of Integration for the REPAS Project

side of the figure. An important component of this data management yet described is the preprocessing step that interprets the physical definition of the MCM and constructs the data models for the client analyses. The remaining portions of this thesis are dedicated to detailing the components on the left side of Figure 28. The next chapter describes the requirements on the formats of the three input files. Then, the three chapters following (chapters 7–9) present the preprocessing schemes used to set-up for the data models. Chapter 7 looks at the functional aspects of the preprocessing step from the viewpoint of the users. Chapter 8 focuses on the approach used to interpret the physical definition of the MCM from the input files. A physical model is built from the given physical definitions of the MCM. This physical model is the basis upon which the global layer model is derived. Chapter 9 describes the derivation methodologies as well as the data structure used. Although the physical model, including the attributes defined, is also used as a basis to build the two local models, these are not described here. For detailed information of the model building methodologies of the local models, please consult the documentation for the REPAS project [4][5].

6 ANALYSIS DATA REQUIREMENTS

Before any analyses can be performed, the physical definition of the interconnect must be preprocessed to construct appropriate discrete analysis models for the client analyses. The MCM physical definition is defined in three files, which must be provided at the start of the analysis process. The three files are: 1) a start-up file that specifies the problem name, local windows, and parameters required by the global and local analyses, 2) a CIF file containing layerwise description of the MCM to be analyzed, and 3) a corresponding attribute file that contains processing information such as layer thickness, material properties, and initial and boundary conditions. This chapter details the format of each of the three input files required for the analyses.

6.1 Start-up File Specifications

At the start of an analysis, the preprocessor looks into a file in the current directory named *epii.model*, to obtain start-up information. This information include the CIF model name, window coordinates for the local analyses, temperature and position resolutions, and reference temperatures. This section details the information required in this file. Figure 29 shows a sample start-up file.

The first line of the start-up file specifies the name of the model to be read. This name is used as a base name for all subsequent input/output files in the analysis process. Given a model name of *25chip*, for example, the corresponding CIF file must be named *25chip.cif* and the attribute file must be *25chip.sup*. Any existing output files from previous runs that have the same model name (in this case, *25chip*) are read/overwritten.

```
25chip                          model name
1e-8                            conversion factor of CIF input file to unit of
meters
(-3367100, -382250, 58000)      lower corner of local window
(-3353200, -366050, 69000)      upper corner of local window
0.1                             temperature resolution
1e-8                            spatial resolution
20                              reference temperature of MCM in °C
120                             process temperature in °C
0                               gt_mterm
0                               gt_nterm
0                               gt_pterm
0                               gt_qterm
```

Figure 29 Sample Start-up File

For the sake of clarity, it is recommended that a separate directory be created for each new analysis problem to avoid any unnecessary confusions.

The second line specifies the conversion factor to convert the units used in the given CIF file to unit of meters. All input references to the given interconnect dimensions are in terms of the unit specified in the CIF file, which is not necessarily in units of meters. For example, many CIF files have MCM designs specified in units of hundreds of a micron. The conversion factor must then be 10^{-8}. Because the distance and length information needed by the analyses are always in meters, the conversion factor supplied is used to do such conversion.

The third and fourth lines specify, respectively, the lower and upper corners of the local window. Although originally planned for the ability to specify multiple local windows, this code is currently limited to only one window. The rectangular parallelepiped defined by the lower left and upper right corner points are used as the analysis domain for the two local analyses. Everything in the MCM as specified in the CIF and attribute files inside this rectangular parallelepiped is used to build the local models. Note that this local window must be specified in the same units as the CIF file. Per the above example, the local window must be specified in units of hundreds of a micron.

The next two lines of *epii.model* contain information needed by the local thermal conduction analysis. The fifth line is a percentage for the temperature resolution. Taking the difference between the upper and lower bounds of the temperature spread, the temperature resolution is calculated as the difference times the percentage specified in the start-up file. The local thermal analysis result will have an error of one standard of deviation equal to the temperature resolution. The sixth line in the start-up file specifies the spatial resolution, which is the smallest size dimension that the local thermal analysis can resolve.

The next two lines of *epii.model* are temperature references required for the global and local thermal elastic stress analyses. The first of the two is the averaged normal operating temperature of the surrounding. The second is the processing temperature of the MCM during fabrication. This is the temperature at which the interconnect is considered to be stress free. Residual stress is expected after cool down.

The remaining four terms are used by the global thermal analysis to control the wave lengths of the temperature function in the interconnect (gt_mterm and gt_nterm) and in the chips (gt_pterm and gt_qterm). These terms are used mainly for trouble shooting purposes and should all be set to zero by the user.

6.2 CIF File Specifications

A representative picture of a typical MCM cross-section is shown in Figure 30 [12][14][7]. The MCM consists of a number of chips set on a block of interconnect

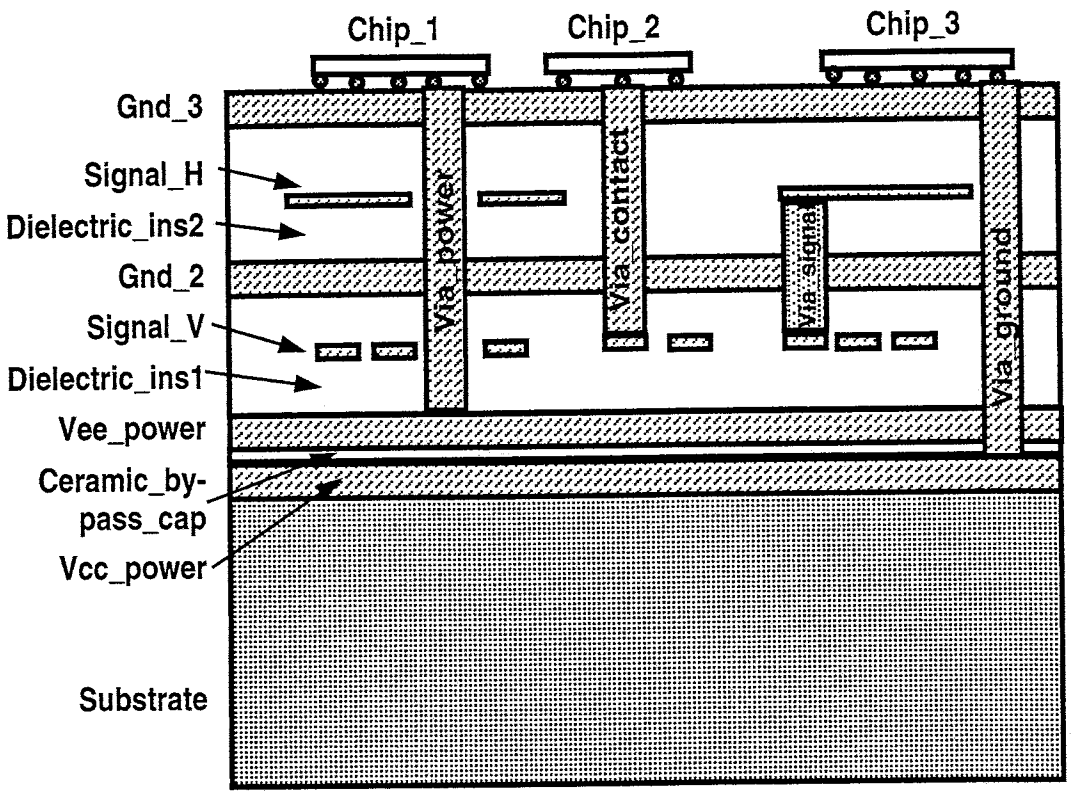


Figure 30 Typical MCM Cross-section

consisting of signal, ground, power, and dielectric planes. A common method to describe such layouts is the Caltech Intermediate Format (CIF), and CIF was chosen as the input standard of the MCM specification. Basically the CIF file describing the MCM design contains 2–D geometric information about the layers for the fabrication process (referred here as CIF layers). Because CIF is such a general data specification format, some requirements and naming conventions, in addition to those of standard CIF, are required to be included in the CIF file for correct extraction of information about the MCM design. The following is a list of assumptions on the information provided in the CIF file which is used to drive the generation of the analysis models for the global and local analyses:

1. The CIF file is always named in the form, *problemName.cif*, where *.cif* must be added as an extension to the problem name, *problemName.*

2. The direction of the thickness corresponds to the z-direction of a global Cartesian coordinate system. All the layers are described in an x-y plane. In the current REPAS software, all the parts in the MCM must align with the axes of a global Cartesian coordinate system, i.e., there are no slanted edges or surfaces.
3. A list of the layers of an MCM recognized by the analysis preprocessor in a CIF file is: **substrate, ground, power, signal, via, dielectric, solder_bumps**, and **chip**. The names of these layers are used as naming conventions in the CIF file. As such, the names of each of the layers specified in the CIF data file (herein called CIF layers) must begin with one of the above keywords. For example, in the case where there are two signal layers in the MCM, one can identify them as signal_1 and signal_2 or signal_x and signal_y. Note that the unique identifications, "1," "2," "x," and "y," are appended to the keyword with an underscore "_" This is the convention for all the other keywords as well.
4. The CIF preprocessor is case insensitive, i.e., two layer names identical in every way except for upper and lower casing are considered to be the same layer.
5. Layers of the same material but locate at different heights (i.e., having different z_min and z_max) must have unique names. For example, if there are two ground layers in the MCM, they must be specified twice in the CIF file with two different names.
6. The material properties and the z-min and z-max of all the layers of an MCM must be specified in a corresponding attribute file.
7. The CIF file directly obtained from the router package contains only those layers for which masks are to be made. In addition to these CIF layers, it is assumed that 2–D geometric information of *every* layer (including the substrate and solder bumps

as well as full metal layers, such as the ground and power planes) are explicitly specified in the CIF file.

8. The centers, lengths, and widths of the chips are to be specified in dummy[1] CIF layers (as boxes) named "chip_*," where * is the unique name of the particular chip. Each chip needs to be specified as a separate CIF layer. Each chip must be specified as one CIF layer.

9. The vias are specified as CIF layers named "via_*," where * is the unique name of the particular via layer. Some of the names used might be "thermal," "signal," or "ground" to distinguish between the different types of vias specified.

10. The centers and nominal diameters of the solder bumps must be specified in a dummy CIF layer with the name "solder_bumps" or "solder_bumps_*," where * is the unique name of the layer. The nominal height of a solder bump also needs to be specified in the attribute file under a keyword corresponding to the name in the CIF file. The solder bump layer in the CIF file maybe specified with boxes or circles, just like that of all other layers.

11. Provision is made for an MCM with dielectric layers[2] with different material properties. The thickness and material properties of the insulation need to be specified in the attribute file.

---

[1] Conforming to electronic packaging naming convention, this layer is called a *dummy* layer because it is not explicitly needed for the manufacturing process. However, it is required for the complete definition of the MCM. The same is true for the solder bump layers.

[2] Unless otherwise specified, a dielectric layer is defined as a rectangular slab of material. If a physical layer contains any of the dielectric material, the x-y dimension of the substrate will be used for the size of the dielectric layer, which should be the size of the physical layer as well. Also, the z-dimensions (thickness) of the dielectric layer (defined in the attribute file) needs to span this particular physical layer.

12. Only dielectric and vias may span the thickness of more than one physical layer. The dielectrics are considered as "background" materials of the MCM, i.e., the complement of any physical layer will be the background material, which spans the length and width of that physical layer. The default background material is the first dielectric layer specified in the CIF file.
13. Nothing is in the substrate and that it is *always* the first layer from the bottom up. This excludes those MCM designs that have ground and power planes embedded in the substrate. This is a current limitation. The x-y dimension of the substrate is used as the x-y-dimension of the MCM.
14. The signals and via layers must be specified before the dielectric, metal, and substrate layers. This is required for correct processing of information for the local thermal analysis.

The layout as described in Figure 30 is a valid MCM description according to the assumptions specified above. Figure 31 shows an example of a partial listing of an actual CIF file for a 25 chip MCM design. According to CIF rules [13], each layer is specified with an **L** followed by the layer name. In this case, notice the layer names follow the naming convention as set forth in assumption 3 above. Following the layer names are the boxes that define the layout configuration of that particular layer.

6.3 Attribute File Specifications

The CIF file is limited to the specification of only 2–D MCM data. This information is not sufficient to describe the analysis domains required for the global and local analyses. Additional information (such as thickness, material properties and initial conditions) need to be specified. These additional information must be specified in a file called

```
L SIGNAL_H; B 4205000 1000 -88000 -543000; B 4205000 1000 -88000 -545000 ; ...
L SIGNAL_V; B 1000 705000 2017000 -190000; B 1000 705000 2015000 -190000 ; ...
L VIA_CONTACT; B 1000 1000 -2793000 653000; B 1000 1000 -2791000 651000 ; ...
L VIA_POWER; B 5000 5000 -962570 666870; B 5000 5000 -962570 561870 ; ...
L VIA_GROUND; B 5000 5000 -962570 681870;
                B 5000 5000 -962570 576870 ; ...
L VIA_THERMAL; B 5000 5000 -1878500 2166500;
                B 5000 5000 -2077000 2164500; ...
L CHIP_1 ; B 1000000 1000000 -600000 -1800000 ;
L CHIP_2 ; B 1000000 1000000 600000 -1800000 ;
L CHIP_3 ; B 1000000 1000000 1800000 -1800000 ;
L CHIP_4 ; B 1000000 1000000 3000000 -1800000 ;
L CHIP_5 ; B 1000000 1000000 -600000 -600000 ;
L CHIP_6 ; B 1000000 1000000 600000 -600000 ;
L CHIP_7 ; B 1000000 1000000 3000000 -600000 ;
L CHIP_8 ; B 1000000 1000000 4200000 -600000 ;
L CHIP_9 ; B 1000000 1000000 -3000000 600000 ;
L CHIP_10 ; B 1000000 1000000 -1800000 -600000 ;
L CHIP_11 ; B 1000000 1000000 1800000 -600000 ;
L CHIP_12 ; B 1000000 1000000 -4200000 -600000 ;
L CHIP_13 ; B 1000000 1000000 -4200000 600000 ;
L CHIP_14 ; B 1000000 1000000 -4200000 1800000 ;
L CHIP_15 ; B 1000000 1000000 -3000000 -600000 ;
L CHIP_16 ; B 1000000 1000000 -3000000 -1800000 ;
L CHIP_17 ; B 1000000 1000000 -3000000 1800000 ;
L CHIP_18 ; B 1000000 1000000 -1800000 1800000 ;
L CHIP_19 ; B 1000000 1000000 -5400000 600000 ;
L CHIP_20 ; B 1000000 1000000 -1800000 600000 ;
L CHIP_21 ; B 1000000 1000000 -600000 600000 ;
L CHIP_22 ; B 1000000 1000000 600000 600000 ;
L CHIP_23 ; B 1000000 1000000 1800000 600000 ;
L CHIP_24 ; B 1000000 1000000 3000000 600000 ;
L CHIP_25 ; B 1000000 1000000 -600000 1800000 ;
L SUBSTRATE ; B 11000000 5400000 -600000 0 ;
L VDD_1 ; B 11000000 5400000 -600000 0 ; ...
L VDD_2 ; B 11000000 5400000 -600000 0 ; ...
L GND ; B 11000000 5400000 -600000 0 ; ...
L DIELECTRIC_INS1 ; B 11000000 5400000 -600000 0 ;
L DIELECTRIC_INS2 ; B 11000000 5400000 -600000 0 ;
```

Figure 31 Partial CIF Listing of 25 Chip MCM Design

the attribute input file. The attribute file will always be in the form *problemName.sup*, where *problemName* is the name of the problem to be analyzed. An example of an actual attribute file for a 25 chip MCM analysis is shown in Appendix D. The attribute file uses a basic block structure; each block contains information of one CIF layer. Conversely, each layer specified in the CIF file must have a corresponding layer block in the attribute file. Each layer information block is grouped further into sub-blocks of physical dimensions, material properties, layout information, and boundary condition. Specific keywords are used to identify the block as well as information within the sub-blocks.

A summary of the type of information required are:

1. the thickness dimension and material properties for all the layers,
2. material properties, and layout information of solder bumps, specified under keyword of **solder_bumps**, and substrate pad information, specified under keyword of **pins**,
3. convective heat transfer coefficients of each chip and the substrate, specified as boundary condition under the respective keywords of **chip_*** and **substrate**, where "_*" is any character string that uniquely identifies the chip — if there were more than one chip,
4. power density output of each chip specified as boundary condition, and
5. minimum wire width and wire pitch specified as layout information for each signal layer.

The attribute file is designed to be keyword driven, with a relatively flexible format. The specification format and options are illustrated here by ways of templates. Unless otherwise specified, the templates are intended to be used for an actual attribute as is written. The keywords are all shown **bold-typed**. <u>Those bold_typed words as input</u>

<u>parameters are required to be written as given</u> (e.g., "**E** = " for the modulus of elasticity that defines a stiffness tensor needs to be written explicitly in the attribute file). The optional input are placed inside (a pair of parenthesis). Each "*" in the first column followed by a word marks off a new unit. All lines beginning with a "#" in the first column are comments and do not represent input. The <*italicized words inside angled brackets*> should be replaced with appropriate numerical values at the time of creation of the attribute file. All real numerical values may be written in E format. For example, 10^{-8} may be specified as 10e8. Unless otherwise specified, each sub-block and keyword may be specified in any order. Additional options for input are specified inside a set of (open and closed parentheses). All comments will be written in helvetica font with a "#" in the first column of the line. Those comments requiring detailed descriptions are specified as ".....NOTE (*reference*)," with the corresponding reference found towards the end of this section. All values are in metric units. Needless to say, incorrect inputs or omission of input will result in incorrect outputs. Depending on the type of error, the first indication of incorrect input may be from warning messages from the preprocessing/model building routines. It is vital that the user take care to provide a correct input file.

```
# This data block is to specify information of the substrate pads under the
substrate
*pins
#.....NOTE m1 (Please see the end of this section for detailed explanation
#    of all such notes)
    material_properties:
#..............NOTE m2
          type: <material type>
#..............NOTE m3
          stiffness: ( E = )<modulus of elasticity>, ( nu = ) <Poisson's ratio>
#
     layout_info:
#..............NOTE l1
```

```
          npins:  <number_of_pins>
#
      physical_dimensions:
#..............NOTE p1

            pin_size:  (<height>, <diameter>) or (<height>, <width>, <length>)
#



# This data block is to specify information of a chip.  All chips are
# to be specified in the same manner.  If the MCM has more than one chip,
# the user must append "_*" to the keyword chip, where "*" is a
# unique specification of the chip.  Please refer to item 3 of the previous
# section for a more detailed description.
*chip_*
      physical_dimensions:
            thickness:  ( zmin = ) <minimum z of layer>,
                        ( zmax = ) <maximum z of layer>
      material_properties:
#..............NOTE m2
            type:  <material type>
#..............NOTE m4
            thermal_cond:  <thermal conduction coefficient(s)>
#..............NOTE m5
            thermal_expand:  <thermal expansion coefficient(s)>
#..............NOTE m3
             stiffness:  <stiffness coefficients and/or independent variables>
#
#.....NOTE b1
      boundary_condition:
#..............NOTE b2
             power_density:  <power density value>
#..............NOTE b3
             heat_trans:  <boundary condition flag>, <first coefficient<,
                              <second coefficient>



# This data block is to specify information of a substrate layer.
*substrate
```

```
    physical_dimensions:
          thickness: ( zmin = ) <minimum z of layer>,
                     ( zmax = ) <maximum z of layer>
    material_properties:
#...............NOTE m2
         type: <material type>
#...............NOTE m4
         thermal_cond: <thermal conduction coefficient(s)>
#...............NOTE m5
         thermal_expand: <thermal expansion coefficient(s)>
#...............NOTE m3
          stiffness: <stiffness coefficients and/or independent variables>
#
#.....NOTE b1
    boundary_condition:
#...............NOTE b3
          heat_trans: <boundary condition flag>, <first coefficient>,
                          <second coefficient>

# This data block specifies the information of a layer of power via. The other
# via layers (such as ground vias, thermal vias, and contact vias) will be
# specified in exactly the same way, substituting the string "power" with
# the appropriate string (such as "ground", "thermal", and "contact").
*via_power:
    physical_dimensions:
          thickness: ( zmin = ) <minimum z of layer>,
                     ( zmax = ) <maximum z of layer>
    material_properties:
#...............NOTE m2
         type: <material type>
#...............NOTE m4
         thermal_cond: <thermal conduction coefficient(s)>
#...............NOTE m5
         thermal_expand: <thermal expansion coefficient(s)>
#...............NOTE m3
          stiffness: <stiffness coefficients and/or independent variables>


# This data block specifies the information of a layer of ground plane. MCMs
# with more than one ground planes must append "_*" to the keyword, where
# "*" gives unique identification to the particular ground plane.
```

```
*ground:
    physical_dimensions:
          thickness: ( zmin = ) <minimum z of layer>,
                     ( zmax = ) <maximum z of layer>
    material_properties:
#..............NOTE m2
         type: <material type>
#..............NOTE m4
         thermal_cond: <thermal conduction coefficient(s)>
#..............NOTE m5
         thermal_expand: <thermal expansion coefficient(s)>
#..............NOTE m3
          stiffness: <stiffness coefficients and/or independent variables>

# This data block specifies the information of a layer of power plane. MCMs
# with more than one power planes must append "_*" to the keyword, where
# "*" gives unique identification to the particular power plane.
*power:
    physical_dimensions:
          thickness: ( zmin = ) <minimum z of layer>,
                     ( zmax = ) <maximum z of layer>
    material_properties:
#..............NOTE m2
         type: <material type>
#..............NOTE m4
         thermal_cond: <thermal conduction coefficient(s)>
#..............NOTE m5
         thermal_expand: <thermal expansion coefficient(s)>
#..............NOTE m3
          stiffness: <stiffness coefficients and/or independent variables>

# This data block specifies the information of a signal layer. MCMs
# with more than one signal layer must replace "_x" with "_*", where
# "*" gives unique identification to the particular signal layer.
*signal_x:
    physical_dimensions:
          thickness: ( zmin = ) <minimum z of layer>,
                     ( zmax = ) <maximum z of layer>
    material_properties:
#..............NOTE m2
         type: <material type>
#..............NOTE m4
```

```
          thermal_cond: <thermal conduction coefficient(s)>
#...............NOTE m5
          thermal_expand: <thermal expansion coefficient(s)>
#...............NOTE m3
           stiffness: <stiffness coefficients and/or independent variables>
     layout_info:
#...............NOTE l2
           min_wire_width: <the minimum wire width for the interconnect>
           min_wire_pitch: <the minimum wire pitch for the interconnect>


#
# NOTE b1 ...
#     For the global heat conduction analysis, thermal boundary conditions
#     are required and enforced exactly on the top of each chip and bottom
#     of the substrate.
#
# NOTE b2 ...
#     The power density generated by the chip must be specified in units of
#     $\frac{W}{m^2}$ , where W is watt and m is meter
#
# NOTE b3
#     Essentially, the method of heat dissipation and the amount of heat
#     dissipated from the top of the chips and through the bottom of the
#     substrate are specified here. The three terms required as input,
#     specifying in order are:
#           1) heat transfer flag: 1, if convective heat transfer
#                                  2, if constant temperature
#                                  3, if zero flux
#           2) first coefficient: 0, if constant temperature
#                      <non-zero number> as convective heat transfer coeff.,
#                      if boundary condition is convective heat transfer
#           3) second coefficient: <reference temperature>, if the b.c. is
#                                      convective heat transfer
#                                  <temperature value>, if constant
#                                   temperature
#     For example, an input of heat_trans: 1, 10, 70 specifies that
#     convective heat transfer is used, with the convective heat transfer
#     coefficient being 10 units, and the reference temperature is 70 units.
#     Notice these three terms may be deliminated by spaces or by comas.
#
# NOTE m1 ...
#    sub-block of material properties. The colon after the material_properties
#    keyword is optional. The spacing and indentation are also optional and
#    variable. At present for the material properties of pins, only the stiffness
#    is needed for the analyses.
#
# NOTE m2 ...
#    Within this material properties sub-block, all the material properties
#    specified after the material type is specified inherits this type — until
#    a new material type is specified. The material type needs to be defined
#    each time a new material sub-block is declared.

#    The options of material types are:
#      LIM : Linear Iosotropic Material
#      LOM : Linear Orthotropic Material
#      LIEM : Linear Iosotropic Elastic Material
#      LOEM : Linear Orthotropic Elastic Material
#      LAEM : Linear Anisotropic Elastic Material
#      IPM : Isotropic Plastic Material, with no temperature dependencies
#      IPMT : Isotropic Plastic Material, with temperature dependencies
#
# NOTE m3 ...
#    Before specifying the stiffness coefficients, be sure that the material type
#    is correct. One usually uses LIEM and LOEM instead of LIM and LOM.
#    The two components specifying stiffness, modulus of elasticity and
#    Poisson's ratio, must be specified in order. They must be delimited
#    with a comma. For orthotropic materials, the stiffness should be
#    specified in terms E and ν in the form of E1=E1, E2=E2, E3=E3,
#    nu12=ν12, nu23=ν23....
#    For anisotropic materials, the user may directly specify the coefficients
#    C11, C12, C13, ..., C21, ..., Cij, ..., C66 Note that the
#     order of specification for the coefficients must be as shown; they may
#     be deliminated by spaces or commas. Also, there must not be any
#     carriage returns in between any coefficients.

#
# NOTE m4 ...
#    Because of the limitation of the analyses, only isotropic and orthotropic
#     materials are supported. Thus, the thermal conduction coefficient input
#    is <κ> for isotropic materials and <κ11, κ22, κ33> for orthotropic
#    materials.
#
```

```
# NOTE m5
#    Allowable thermal expansion coefficients are:
#         <$\alpha$> for isotropic materials
#         <$\alpha_{11}$, $\alpha_{22}$, $\alpha_{33}$> for orthotropic materials
#         <$\alpha_{11}$, $\alpha_{12}$, $\alpha_{13}$, $\alpha_{22}$, $\alpha_{23}$, $\alpha_{33}$> for anisotropic materials.
#
# NOTE l1 ...
#    This is only for pins and is specified under npins; it is the total
#    number of pads found under the substrate
#
# NOTE l2 ...
#     For each signal layer, the minimum wire width, defined to be the
#     minimum allowable width of a typical wire, must be specified.
#     Also, the minimum wire pitch, defined to be the minimum distance
#     between the center of any one wire and the center of its adjacent
#     wire, must be specified.
#
# NOTE p1 ...
#    The pin_size is where the dimensions of the pins are specified. For
#    circular pads, only the nominal height and nominal diameter of the
#    pads need to be specified. For rectangular pads, height, width, and
#    length (all averaged values) must be specified.
```

7 DATA PROCESSING (a) — REPASpre

Assuming the three input files are correctly set-up, the next step to run a complete thermal/thermal mechanical analysis is to preprocess the given information. The preprocessing routine, **REPASpre**, composes of three major parts: attribute data-base preparation, physical model building, and global idealized model building. Each of the three parts is discussed in the subsequent three chapters; this chapter discusses the requirements to correctly invoke the preprocessor. To facilitate discussion, all of the analysis modules dependent on the data processed by REPASpre, such as the global and local thermal analyses, are referred to as *client analyses* hereafter.

7.1 REPASpre Overall Functional Description

REPASpre is a data preprocessing module that reads the data from the input files, interprets the data, builds working models with this data, associates the data with the model, and stores the processed information with the model information for use by the client analyses. This entire process from start to finish is transparent to the user. The user only needs to provide three input files, and then three output files are produced when REPASpre finishes. The basic flow of the module is as follows: after obtaining the initial start-up information from *epii.model*, REPASpre reads from the CIF file to obtain the CIF layer names. As each layer is read, all relevant data for that layer is read in from the CIF file and the attribute file. REPASpre then creates a set of attributes from this data. A layer of the physical model is built from this information, with the attributes of this CIF layer associated to this layer of the physical model. When all information is read and the physical model complete, the averaged layer information is extracted from the physical model to build the global idealized model. Another set of attributes are created from this averaged information and are associated to the appropriate layer in the idealized global model. Finally, all the attributes created in the process are written out to three output files to be used for input for each of the client analyses.

7.2 Function Capabilities

Given the required input files, the entire data preprocessing process is automatic, which includes CIF and attribute file parsing, attribute creation, organization, and association, and physical and idealized model building. The capabilities and limitations of each of the preprocessing modules are detailed in this section.

The input format is expandable and is relatively flexible. The file specification format was detailed extensively in Chapter 6. The only restriction is that the input formats for the three input files be conformed strictly to the format as specified in Chapter 6. Even a misspelling of a keyword can result in an infinite loop. If there is an update of the source code in the future, this deficiency should be eliminated. For now, all information must be specified explicitly. For example, for two layers that are made of the same material, the material properties of each layer must be specified separately. To reiterate, the correctness of the resulting model depends heavily on the strict adhesion to the assumptions listed in Chapter 6.

At the moment, only boxes and round flashes are recognized for the description of CIF layers in a CIF file. Rectangles, polygons, and wire paths are ignored. These other options can be added in the future as the need arises.

REPASpre can process as many CIF layers as the system memory allows. Each CIF layer may have up to one hundred material property attributes. The stiffness coefficients of one material are considered to be one attribute. The same limit applies to the boundary condition, physical dimension, and layout information attributes. Generally, a level 3 data block in the attribute file (see section 8.1, **Physical Model Building Approach**) is considered to be one attribute. So long as the assumptions of the CIF and attribute files as described in Chapter 6 are satisfied, REPASpre can process any MCM configuration with any number of signal layers and vias. The global idealized model building routine assumes that other than vias and dielectrics, no other CIF layers of a different height overlap one another in the z-direction.

7.3 Input Requirements

This section details the input requirements for invoking REPASpre. Three input files must be present in the current working directory before starting REPASpre: *epii.model*, *modelName.cif*, and *modelName.sup*, where *modelName* is the base name of the CIF file to be read. If the start-up script were not used, any of the three SAM data files from previous runs must first be removed from the current directory. The three SAM data files are: *modelName.lib*, *modelName.org*, and *modelName.rel*. Otherwise, the SAM data-base will be corrupted, and error messages will be echoed to the screen.

Also, the full path to the REPAS home directory must be furnished to the environment variable, **REPAS_HOME**, if the start-up script were used. Otherwise, the environment variable, **ATTLIB**, must be set at the prompt:

```
> setenv ATTLIB $REPAS_HOME/sam/lib
```

where *$REPAS_HOME* is the full path to the home directory of the REPAS code.

7.4 Output Files

REPASpre produces three output files: *modelName.lib*, *modelName.org*, *modelName.rel*. These three files are data files for the attribute manager, SAM. The *modelName.lib* contains a library of numerical definitions for all the attributes defined. The *modelName.org* contains the organizational structure in which these attributes are defined. The *modelName.rel* contains all the associations of the attributes to the relevant geometric and idealized models. For a detailed discussion of the formats of these output files, please refer to Chapter 4 of the document on the SCOREC Attribute Manager. These three files are required to be in the same directory where the analyses are run, as they are the inputs for each of the client analysis.

7.5 Executing

To invoke REPASpre, type at the prompt:

```
> $REPAS_HOME/bin/REPASpre
```

7.6 Error Diagnostics

This section details some of the possible diagnostics for error messages received during execution of REPASpre. Warning and error messages related to SAM are echoed to the error file *modelName.ATerr*. Otherwise, all other REPASpre errors are echoed to the screen. If REPASpre aborts abruptly with no messages shown on the screen, be sure to check *modelName.ATerr* for any error messages. Table 1 lists diagnostics of possible run-time errors.

Error Message	**Probable Cause**
ERROR: no model name read.	No *epii.model* in the current directory, or there are no input files corresponding to the name read from *epii.model.*
ERROR: can't find input file 'attrib.keys' in path '(null)/attrib.keys'	Environment variable **ATTLIB** not set to *$EPII_HOME/sam/lib*, or library files in *$EPII_HOME/sam/lib* were moved.
Should only be one attribute returned, but have 2	SAM data files *modelName.lib*, *modelName.org*, and *modelName.rel* were not removed before starting EPIIpreproc, or duplicate definition of layer in CIF or attribute file.
Warning messages and Notes...	General information for user.

Table 1 REPASpre Diagnostics

8 DATA PROCESSING (b) — PHYSICAL MODEL BUILDING

The building of models for client analyses requires a knowledge of the all the information associated with the domains to be analyzed. The first model to be built is the physical model, upon which all other domains are derived. To maintain consistency in the information used to build the analysis domains for the global and local procedures, each client analysis domain must be derived from the physical model domain. The physical model is an abstract description of the layers (CIF layers) as given in the CIF file. The domain information is obtained from the CIF file supplemented with information from the attribute file. Because this physical model is the domain based on which all other domains for the client analyses are built, it is used solely as and must be a source of accurate physical representation of the MCM as specified in the input files.

The overall functional description of REPASpre has already been detailed in Section 7.1, and it is not repeated here. Please refer to the previous chapter for a review on the overall functional description of REPASpre. As the focus of the previous chapter is from the stand-point of the user, this chapter and the next will focus at main portions of REPASpre from the stand-point of the programmer. This chapter is devoted to detailing the approach in the building of and the design of the data structure used for the physical model.

8.1 REPAS Preprocessing Overall Scheme

Before discussing the physical model building process, it is necessary to describe where this physical model building fit in the overall scheme of REPASpre. All data is preprocessed through one driver, which, in turns, calls different modules to build up the information required for the various analyses. This driver first sets up the data structure for the creation and storage of attributes for the various model domains. This set-up utilizes the operators provided by the SCOREC Attribute Manager (SAM). The CIF file is then parsed once to obtain the names of each CIF layer. The cumulative x-y cross-sectional areas of each CIF layer is also tracked during this first parsing step. As each new CIF layer is encountered while parsing, a set of routines is called to build the physical model from the corresponding supplement information specified in the attribute file. The routines that build the analysis model for the global analyses are then invoked to complete the preprocessing scheme. The detailed geometric and thermal models for the local analyses are built after the preprocessing step with the knowledge obtained from the physical model. Pseudo–Code 17 shows the algorithmic flow of the driver for preprocessing the input data.

Several points about the main driver is highlighted here. The driver consists of essentially three major parts: data-base preparation, physical model building, and global idealized model building. Because SAM is the sole means of storing and passing domain information to client analyses, it is an integral part of the model building process. Therefore, SAM is set-up before any information is processed — thus enabling the interface operators of SAM be used in the model building process to define and store the domain information. Then the CIF file is parsed from top to bottom to obtain the CIF layer information for the physical model domain. Two keywords in the CIF file are of particular interest to the parser in this first pass — "L" and "B" standing for *layer* and *box*, respectively. For the moment, this first pass ignores any definition with rectangles or polygons; all CIF components, except solder_bumps, must be specified in

```
GTpreproc()
{
  open_input_files();
  /* get the model name from "epii.model" */
  get_model_name();
  SAM_data_base_setup()

  /* set up the organizational framework for the
   * five analyses. */
  SAM_organization_setup();

  /* get scaling factor of the CIF file specification
   * from "epii.model" */
  get_scaling_factor();

  /* start reading the CIF file for the layers */
  /* have a peek at the next character in CIF file
   * to look for keywords. */
  Peek_at_the_next_input_character;
  while (NOT_End_Of_File)
  {
     skip_all_blanks();
     switch(next_input_character)
     {
       case EOF: goto done;
       case 'L': CIF_Read_Layer(CIF_layer_number);
       case 'B': CIF_Read_Box(CIF_layer_number,scalling factor)
       case 'E': CIF_Parse_End();

       /* for now, ignore polygon or rectangle
        * specifications in the CIF file */
       default: Ignore_As_Comments();
     }
     Skip_To_Semicolon();
  }
  done:
  build_global_model;
  CIF_Read_Additional_Info();
  store_attribute_data_base();
}
```

Pseudo-Code 17 REPAS preprocessing

box form. Solder-bumps are parsed in the same pass. When a box in a particular layer is encountered, the x-y cross-sectional area of the box is calculated based on the given box length and width. This area is added to the accumulative x-y cross-sectional area of the CIF layer in the physical model. As the need arises, extensions to flashes and paths may be added simply as more parsing options. After the CIF file has been completely parsed, the idealized global model is then built. (This building process for the idealized model is discussed in the next chapter.) As is detailed in the next chapter, another pass through the CIF file is made at this time to obtain information of the solder bumps and vias under each chip. The attributes created are added to the global model. Finally, all the attributes created, organized, and associated are then stored in three output files to be used for each of the five client analyses.

8.2 Physical Model Building Approach

The basic approach in building the physical model is first to read in each layer as defined in the CIF file. When a layer is read, all relevant information of the current layer (from the CIF file and the attribute file) is read in and appropriate attributes created and organized. This information is then associated with the current layer. Each subsequent layer from the CIF file is read and the appropriate attributes created — until all CIF layers are processed. The additional layer of pins not specified in CIF are then processed in the same way. The pseudo-code for building the physical model is shown in Pseudo–Code 18. This basic approach is relatively simple and straight forward. The challenge of building this physical model, rather, is in the implementation of this approach — in obtaining the necessary information from input source, in accurately defining, efficiently organizing, and storing the complex set of data in a manner usable for client analyses,

```
/* As parsing CIF file from top to bottom, when encounter
 * a CIF layer definition, get the CIF layer name */
while (more CIF layer available)
{
  Get_Next_CIF_Layer_Name();
  if (CIF_Layer == new)
  {
     /* allocate space for a new CIF layer */
     Create_CIF_Layer();

     Assign_Number(CIF_Layer);

     /* goto the correct attribute file location and
      * get all attributes belonging to this CIF layer */
     CIFsup(CIF_Layer);

  else
  {
     /* if CIF layer had previously been read, get the
      * corresponding layer number */
     Get_Layer_Number(CIF_Layer);
  }
}
```

Pseudo-Code 18 Physical Model Building

and in providing a structure for retrieval and access of the domain information stored. The discussion in this section, then, is focused on some of the important implementation details of the physical model building process.

The requirements as specified in Chapter 6 allow for the building for the physical model in a structured manner. The routines that parse the supplemental information of the attribute file take full advantage of the attribute file format as illustrated in Figure 32. All the attributes of a CIF layer are grouped into one large data block. To aid in discussion, this grouping is called *level 1* division. The level 1 data blocks can be divided into level 2 data blocks. A close examination of the attribute file specification as described

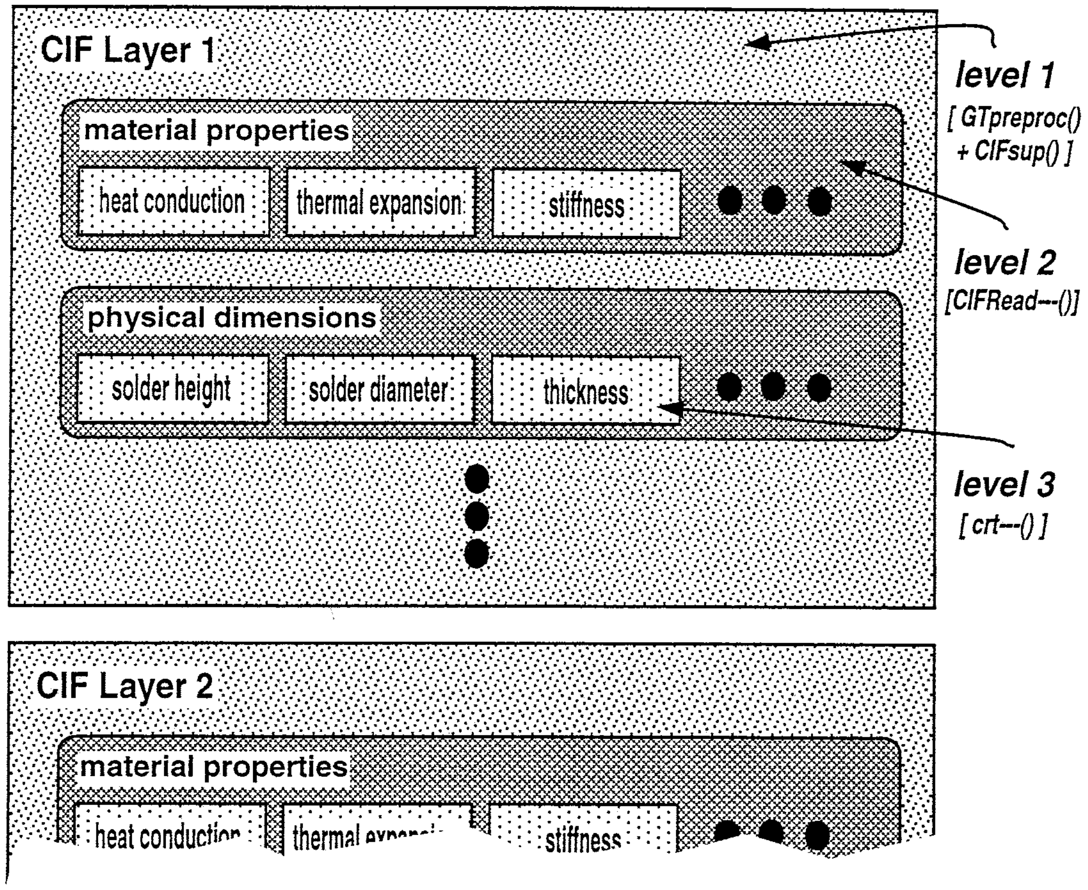


Figure 32 Attribute File Parsing Routine Format

in section 6.3 identifies four types of level 2 data blocks: material properties, physical dimensions, boundary conditions, and layout information. Each of the level 2 data blocks are divided further into level 3 data blocks that contain numerical values of the attribute. Some examples of level 3 data blocks are heat conduction and stiffness coefficients for the material properties (level 2) data block.

Just as the attribute file format is organized into 3 level data blocks, so the parsing routines are organized in like manner. At the top level is an overall driver named CIF-sup(), which calls four level 2 modules of CIFReadMatl(), CIFReadPhys(), CIFReadBC(), and CIFReadLayout() for the four respective level 2 data blocks. The pseudo-code for the level 2 modules are shown in Pseudo–Code 19. These modules search for and locate in the attribute file the target level 2 data block and obtain the keywords for the level 3 data blocks. According to the keyword (such as *stiffness*), the appropriate level 3 modules are invoked to parse the level 3 data blocks. The level 3 modules all named with the convention **crt—()**. The level 3 parsing routines read in the numerical information from the attribute file and create the appropriate attributes. The SAM operators as described in chapter 4 are utilized for the creation of proper attributes. As can be seen, the programming structure is designed to be easily expandable. New routines (most frequently crt—() routines) may be inserted into the current structure as new data blocks are required for the client analyses. All of the parsing modules assume that the rules for the attribute file specification as described in section 6.3 are strictly followed, and the integrity of the attributes created is strongly dependent upon the strict adhesion to the rules.

```
/* check the SAM organization structure to see if this portion
 * of the data for the current CIF layer had been processed
 * already. The keyword that identifies the level 1 data block
 * is used. Exit if yes. */
if (keyword processed == TRUE)
  exit;
else
{
  /* use the layer name to set-up and build into the data
   * structure of SAM. */
  Attribute_Create_Organization(CIF_Layer);

  /* parse the attribute file for the relevant information */
  while (within the CIF layer (level 1) data block)
  {
     /* skip those data level 2 data blocks that are not the
      * the same as that of the target level 2 data block. */
     Skip_Level2_Data_Block();
```

Pseudo-Code 19 Attribute Processing Modules (Continued) . . .

```
        /* skip the comment lines */
        Skip_Comment();

        /* if the level 2 data block is equivalent to that of another
         * CIF layer, get the attributes from that level 2 data block */
        if (level 2 data block == another level 2 data block)
          Grab_Level2_Data_Block(keyword);
        else
        {
          /* else, parse the current level 2 data block */
          switch ( level 3 data block keyword)
          {
            /* according to keyword, call level 3 routines to parse
             * the level 3 data block */
            crt—();
            crt--();
            crt--();
          }
        }
    }
  }
```

Pseudo-Code 19 Attribute Processing Modules

8.3 Physical Model Data Structure

The data structure for storing the physical model information needs to accurately reflect information of the CIF layers and to be easily searched and retrieved by the subsequent model building routines. The data structure used is shown in Pseudo–Code 20. All the attributes created for a CIF layer are stored in a linked list. The handle to this list is stored with the CIF layer. The CIF layer may be identified by name or by the vertical position of the layer (z-min and z-max). Two indices (both are one-dimensional integer arrays) utilize the CIF layer name and the z-mins. One index array points to the CIF layer in alphabetical order, according to the name of the CIF layer. The other index array points to the CIF layer in ascending numerical order, according to the z-min of the CIF layer. Instead of manipulating the main physical model data structure (which is at best very cumbersome and error prone), all manipulations and sortings are done through

```
/* The data structure to hold the CIF layers as was read in from the CIF
 * and attribute files. This is used as a reference for all the other
 * indices sorted in various ways. */
typedef struct attlist {          /* type define for a linked list of */
  struct attlist *next;           /* attributes */
  void *attPtr;
} attList;

typedef struct layerlist {
  char *layerName;               /* CIF layer name */
  double z_min, z_max;           /* z-min and z-max of the CIF layer */
  double area;                   /* total cumulative area of the layer */
  int phyLayer;                  /* index to the owning physical layer */
                               /* negative numbers are chips */
  attList *attList_ptr;          /* linked list of attributes applied */
                               /* on this layer */
} layerList;
```

Pseudo-Code 20 Data Structure Definition for Physical Model the index arrays. These arrays are maintained as the physical model is built so as to provide fast access to a particular CIF layer.

Two other pieces of information are stored: the cumulative area of the CIF layer and the index to the owner physical layer (from the global idealized model). These are needed to build the global idealized model. The parsing routine allocates memory for a new structure in the array of layerList every time a new CIF layer is encountered while parsing the CIF file. Table 2 shows an example of the information built from a sample 25–chip MCM design. The CIF layers are listed in ascending numerical order according to the z-mins of the layer. Notice the remaining chips have the same information as those layers shown. Notice also when two layers have the same z-min with different z-max's the order between the two layers is arbitrary.

CIF name	**z-min**	**z-max**	**Area**	**Attributes**
substrate	0	0.00060	0.00594	*pointer*→
gnd_1	0.00060	0.000625	0.00594	*pointer*→
via_ground	0.000625	0.0007222	2.75e-07	*pointer*→
vdd	0.000625	0.000650	0.00594	*pointer*→
dielectric_ins1	0.000650	0.000680	0.00594	*pointer*→
via_power	0.000650	0.0007222	2.95e-07	*pointer*→
signal_h	0.000662	0.000668	0.000181922	*pointer*→
via_contact	0.000668	0.000698	3.294e-07	*pointer*→
gnd_2	0.000680	0.000686	0.00594	*pointer*→
dielectric_ins2	0.000686	0.000716	0.00594	*pointer*→
signal_v	0.000698	0.000704	7.90442e-05	*pointer*→
gnd_3	0.000716	0.000722	0.00594	*pointer*→
chip_10	0.0007222	0.0007972	0.0001	*pointer*→
chip_9	0.0007222	0.0007972	0.0001	*pointer*→
chip_2	0.0007222	0.0007972	0.0001	*pointer*→
:	:	:	:	:

Table 2 Physical Model Information Stored in a 25–Chi MCM Design

9 DATA PROCESSING (c) — GLOBAL IDEALIZED MODEL BUILDING

The global analysis domain is modeled from a layerwise idealization of the actual MCM. This domain is the same for both the global thermal and thermal mechanical client analyses. The information of the actual MCM is obtained solely from the physical model created in the previous preprocessing step. If the physical model of the MCM were as shown on the left of Figure 33, then the global idealized model of the physical model is as shown on the right of that model in the same figure. With the exception of those layers containing the interconnect signals, all other layers are modeled as homogeneous global layers. In this preprocessing step, the layers with the interconnect signals are idealized as 2–material heterogeneous layers, consisting of dielectrics and signals. The corresponding volume fractions of the dielectrics and signals are calculated and later

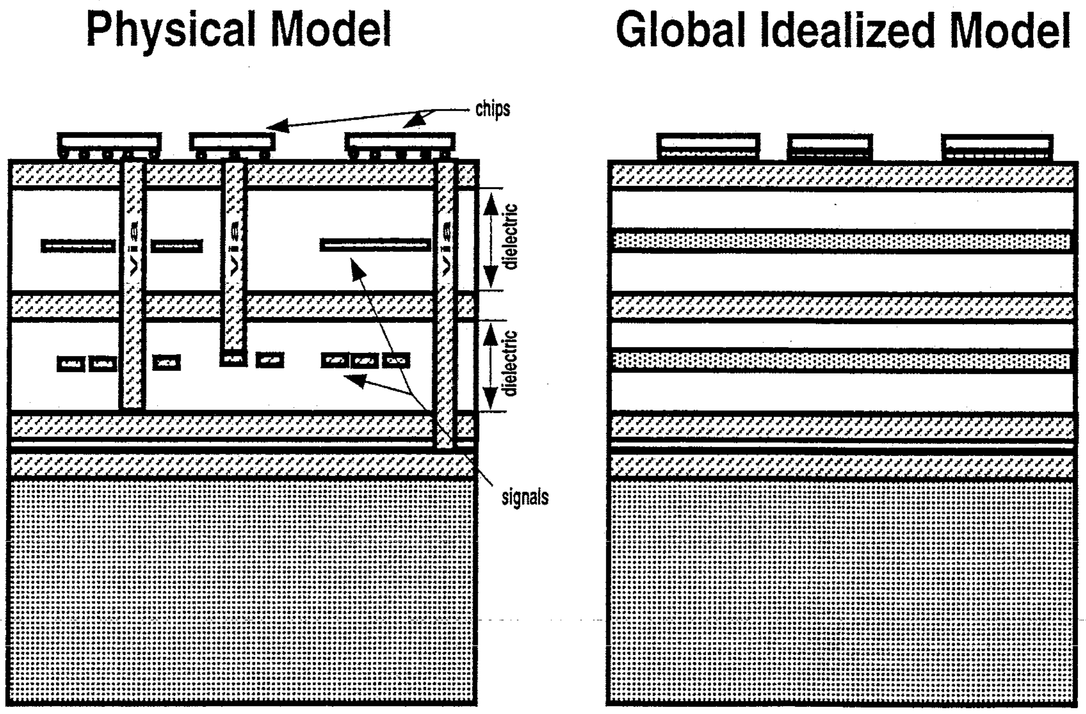


Figure 33 Illustration of the Global Idealized Model

passed as attributes to the global analyses to calculate the effective material properties of the signal layers. In the scope of the project, the averaging of material properties is performed by the client analyses. Therefore, this set of global model building routines only need to provide to the client analyses information of the different material layers (member layers) contained in each global layer and the volume fraction of each member layer. Volume fractions are used because the distribution of the different materials are not required for the global analyses. This chapter details the approach in the building of this global idealized model from the physical model.

9.1 Global Model Building Approach

As depicted in Figure 33 and discussed above, the model that the global client analyses want is an "averaged-layerwise" representation of the physical model [15][21]. Whereas the physical model contains vias, signals, and dielectric layers interpenetrating each other, these features must be decomposed and reconsolidated to create a model that the global client analyses recognize. The general strategy of building the idealized model from the physical model requires two major operations (Figure 34): step 1) decompose the physical model into appropriate layers and step 2) consolidate the decomposed layers into global idealized layers. The decomposition requires full understanding of how all the layers given in the physical model fit together. From the knowledge of the physical model, the layers are broken up into idealized model layers. Since one idealized layer may contain parts of many CIF layers, all of the decomposed pieces need to be consolidated into the appropriate idealized layers. As a final step to complete the model for the global client analyses, the global analysis domain needs to inherit all the appropriate attributes from the domain as described by the physical model. The global idealized model is complete when all idealized layers are consolidated and the domain information fully defined (step 3).

As one can see from step 2 of Figure 34, the decomposition and consolidation steps are actually iterations of steps because they are carried out one idealized layer at a time. When all the idealized layers are consolidated, the global idealized model will be completed with the appropriate domain information. The algorithm that implements the global building strategy is as follows:

1. The CIF layers are sorted with respect to their positions in the z-direction in ascending order.
2. Each CIF layer is examined, with the layers needing further processing sifted out. The remaining CIF layers are then inserted into the global model.
3. The dielectrics layers, which were sifted out in the previous step, are then inserted into the appropriate global layers.
4. Once the global layers are built, the CIF file is parsed to find the area of the vias going from each chip to each of the global layers. This information is used in global thermal analysis to determine the amount of heat carried by the vias from the chip to the different layers of the MCM.
5. The CIF file is parsed again to find the number of solder bumps and the averaged area and the nominal height of the solder bumps under each chip.

This algorithm is illustrated in detail in Pseudo-Code 21 and some of the important points are discussed following the pseudo-code.

```
/* keep an index list of the CIF layers of the physical model
 * sorted by z-min.*/
```

Pseudo-Code 21 Global Model Building Algorithm (Continued) . . .

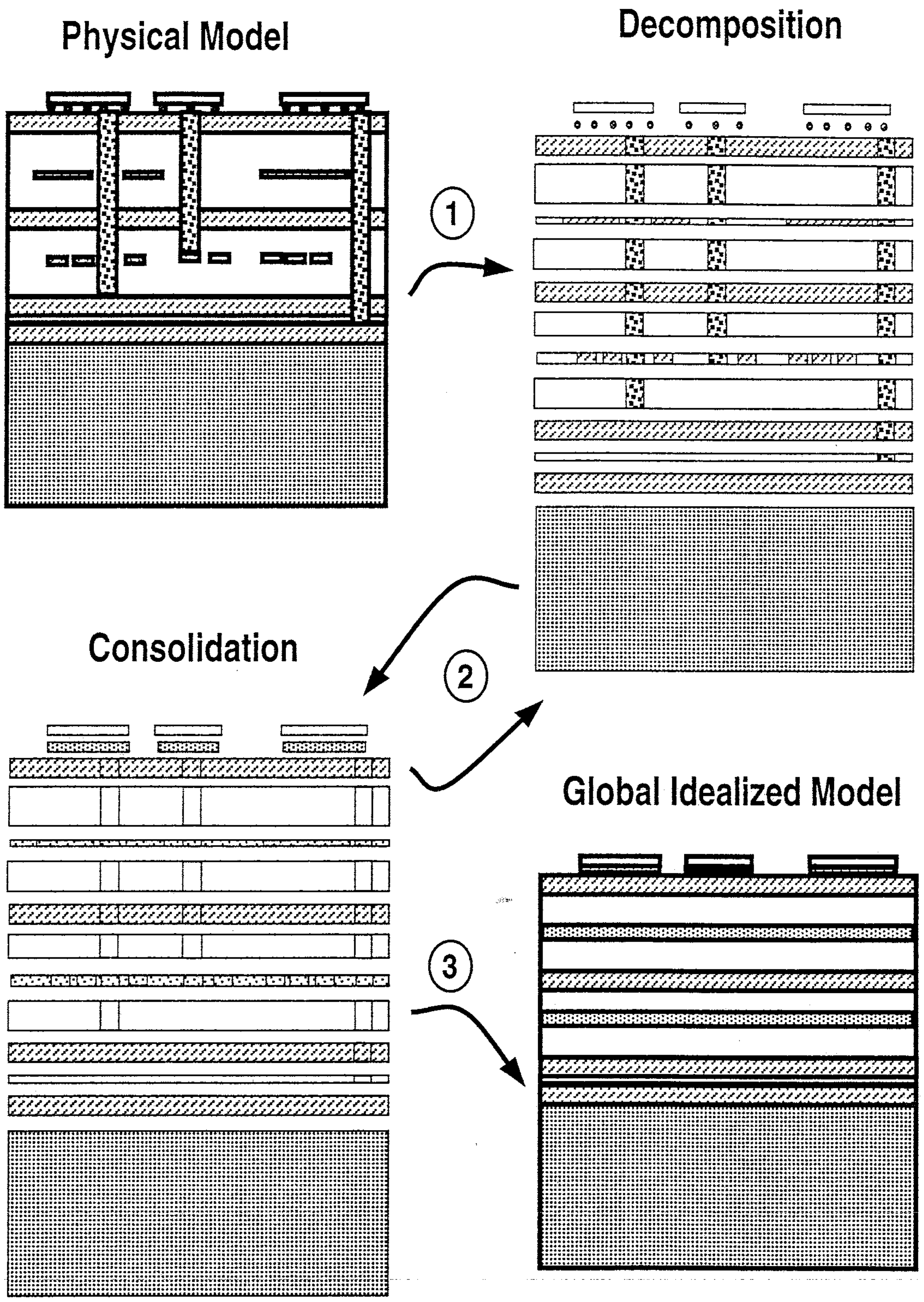


Figure 34 Global Idealized Model Building Approach

```
Sort_CIF_Layers_By_Z_min

/* initialize the counters of chips, vias, dielectric, physical, global,
 * and current working global layer(s) */
Initialize_Count
/* go through all the layers of the physical model from the bottom up,
 * beginning with first (bottom) CIF layer, using the sorted index
 * list */
while (layer != Total_Number_Of_CIF_Layers)
{
  Extract_Keyword_From_CIF_Layer_Name

  /* based on the keyword, sift out all the chips, substrate, via,
   * and dielectric layers */
  switch( keyword )
  {
    case "substrate":
    {
      /* From the assumption in item 13 of section 5.2, the substrate is
       * always physically the bottom-most layer of the MCM. Therefore,
       * the substrate should be the first CIF layer encountered. This
       * layer set to be the first (bottom) layer of the global model */
     Set_Global_Layer
     /* set the back pointer on the CIF layer to this first idealized
      * layer */
     Set_Back_Pointer
     Increment_Global_Layer_Count
    }
   case "chip":
   {
     /* keep count of number and keep track of chips */
     Store_Chip_Index
     Increment_Chip_Count
     /* set the back pointer of the CIF layer to negative of the chip */
     Set_Back_Pointer
   }
   case "via":
   {
     /* keep count of the # of and keep track of the vias encountered */
```

Pseudo-Code 21 Global Model Building Algorithm (Continued) . . .

```
  Store_Via_Index
  Increment_Via_Count
}
case "dielectric":
{
  /* keep count of the number of and keep track of the dielectric
   * layers encountered */
  Store_Dielectric_Layer_Index
  Increment_Dielectric_Layer_Count
}
default:
{
  /* The remaining CIF layers should fit directly into the global
   * layer without further processing */
  /* Check and set the current working global layer */
  If ( Current_CIF_Layer_Higher_Than_Previous_Global_Layer )
  {
    Increment_Current_Working_Global_Layer
    If ( Gap_Between_Current_CIF_Layer_And_Previous_Global_Layer )
      Increment_Current_Working_Global_Layer
  }
  /* check for overlap */
  If ( Current_CIF_Layer_NOT_Overlap_Previous_Global_Layer )
  {
    /* increment number of CIF layers this global layer contains */
    Increment_Member_layers

    /* add the index of the CIF layer to the current global layer */
    Set_Global_Layer

    /* set the back ptr of the CIF layer to the current global layer */
    Set_Back_Pointer
  }
  else
  {
    /* two layers overlap, which violates the assumptions that other
     * than via and diselecetric layers, no other CIF layers may have
     * partial overlaps in the z-direction. Issue warning. */
    Echo_Warning
```

Pseudo-Code 21 Global Model Building Algorithm (Continued) . . .

```
          }
          /* break up and insert each dielectric layer into the global
           * idealized model structure */
          Foreach ( Dielectric_Layer )
             Insert_Layer

          /* Now that the global analysis domain is constructed, associate the
           * appropriate attributes to each idealized layer to fully define
           * the domain */
          Foreach ( Global_Layer )
          {
            /* loop through member CIF layers and retrieve the owning
             * attributes */
            Foreach ( Member_CIF_Layer )
            {
              Get_Attributes
              /* loop through each attribute and associate it with the current
               * current global layer */
              Foreach ( attribute )
                Associate_Attribute_With_Global_Layer
            }
          }
          /* Do the same for the chips */
          Foreach ( chip )
          {
            Get_Attributes
            /* loop through each attribute and associate it with the current
             * current chip */
            Foreach ( attribute )
              Associate_Attribute_With_Chip
           }
        }
      }
    }
```

Pseudo-Code 21 Global Model Building Algorithm

Several important points about the algorithm are mentioned in this section. The success of the decomposition and consolidation steps depends heavily on the strict adhesion of the CIF and attribute file inputs to the assumptions as detailed in Chapter 6

Analysis Data Requirements. The strategy in building the global model is to build one global layer at a time, starting with the bottom layer. When the bottom layer is finished, the current working layer is incremented to the next global layer. An index array, with z-min of each CIF layer sorted in ascending order, is used to reduce the number of searches needed to find all the CIF layers belonging to a particular global layer. Table 2 of section 8.3 shows the CIF layers of the physical model sorted in this order of the vertical position. Notice the first layer (i.e., the bottom-most layer) must necessarily be the substrate according to assumption 13 of the **CIF File Specifications** (Section 6.2). Error in the global model building process will occur if this assumption is not met, since the substrate is used as the reference point for all subsequent layers.

In addition to the substrate, three other types of CIF layers from the physical model are sifted out for further processing: chips, vias, and dielectric layers. As the chips are sifted out, they are stored and numbered according to the order of encounter. Recall from assumption 12 of the **CIF File Specifications** (section 6.2) that the via and dielectric layers are the only two layers that may span the height of more than one CIF layer (in the z-direction). Therefore, these two layers need to be sifted out and be broken into the appropriate global layers. The decomposition step (Step 1) in Figure 34 illustrates the breaking of the via and the dielectric layers.

All other types of CIF layers should fit directly into the global layer without any further processing. Notice also from Table 2 that if the vias, chips, and dielectric layer are taken out, of the remaining layers the z-max of one layer should never be greater than the z-min of the next CIF layer. The CIF layer vertical positions, z-min and z-max, are used to place the CIF layers into the appropriate global layer. This placement is always

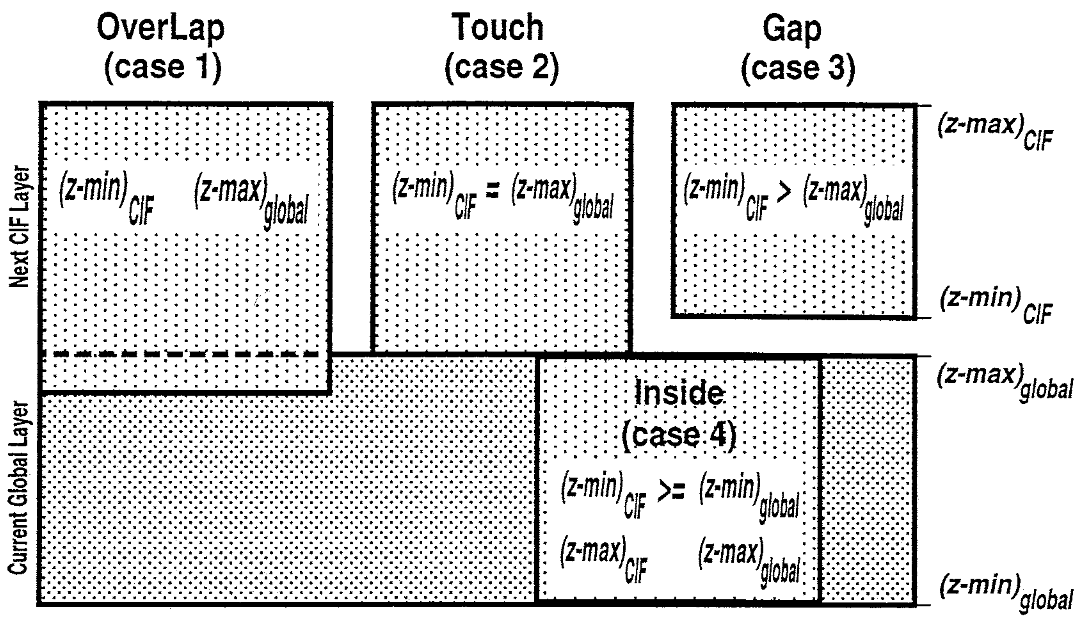


Figure 35 Criteria of Layer Comparison

relative to the current working global layer. As was mentioned above, the substrate is necessarily the first CIF layer inserted into the global structure. This is the only known position at the start of global model building. After the substrate is inserted into the first global layer, the vertical position (z-min and z-max) of the next CIF layer is compared to that of the current working global layer, which is the vertical position of the substrate. Since one global layer may contain many CIF layers, this comparison determines whether the next available CIF layer belongs to the current global layer or that the current global layer must be incremented to the next global layer. Figure 35 illustrates the criteria used to determine the state of a CIF layer relative to that of the current global layer. Four cases are checked.

Case 1. $(z\text{-}min)_{CIF} < (z\text{-}max)_{global}$ According to the assumptions set forth in section 6.2 **CIF File Specifications**, this case is invalid. Only the via and dielectric layers may span the height of (overlap) more than one layer, and the via and dielectric layers had been sifted out previous to this decomposition step. A warning message to check this CIF layer is issued, and this CIF layer is ignored.

Case 2. $(z\text{-}min)_{CIF} = (z\text{-}max)_{global}$ The next CIF layer is directly above the current global layer. Therefore, the current working global layer is incremented to the next global layer.

Case 3. $(z\text{-}min)_{CIF} > (z\text{-}max)_{global}$ A gap implies that a background material is defined between the next CIF layer and the current global layer. According to the CIF file specification assumption 12 of section 6.2, the background material is a dielectric layer — which was sifted out previous to the decomposition step. This gap, then, is filled with the dielectric, which is a new global layer. Therefore, the working global layer needs to be incremented twice to make room for the dielectric layer to be inserted at a later time.

Case 4. $(z\text{-}min)_{CIF} >= (z\text{-}min)_{global}$ and $(z\text{-}max)_{CIF} <= (z\text{-}max)_{global}$ The forth case is when the next CIF layer is part of the current global layer. For this case, the vertical position (z-min and z-max) of the current global layer must bound those of the CIF layer. If only one of the two conditions are satisfied, an invalid CIF layer specification warning is issued and the CIF layer is ignored. This is the same reasoning as two overlapping layers of Case 1. Since the CIF layer is neither a via nor a dielectric layer, it cannot span the length of more than one global layer.

At the conclusion of the comparison, the placement of the next CIF layer is determined and the CIF layer is consolidated into the current global layer. This placement process continues until all the remaining CIF layers are consolidated into the global structure. Finally, the layer that were sifted out earlier are now ready to be decomposed and consolidated into the global model.

The chips do not need any further processing. Also, only selected information on the vias need to be processed. The vias are important to the global analyses in that they are some of the major heat carrying agents from the chips and distribute the heat to the different layers of the MCM. Therefore, the total x-y cross-sectional area of the vias extending from each chip to each layer of the MCM is tabulated. This tabulated information is made into attributes and made available to the global client analyses. In proportion to the tabulated areas, the distribution of the amount of heat of each chip carried into the MCM is determined by the client analyses. Other than this information, the global analyses assume that the contribution of the vias to the averaged material properties of the global idealized layers is insignificant. Therefore vias do not need to be incorporated into the global model (only the x-y cross-sectional attributes are needed). Therefore, for the decomposition step, only the dielectric layers need to be decomposed and consolidated into the global structure. The vias can be inserted easily into the global model at a later time if they need to be taken into consideration. The solder bumps are also tabulated for each chip and given to the global client analyses the same way as the vias. They carry portions of heat to the very top layer of the MCM.

At this point, the basic structure of the global model is defined except for the dielectric layers. The insertion strategy is that for each of the dielectric layers, the global model is scanned from the bottom and up. For the sake of discussion, a gap between two current global layers is also considered to be a global layer. For each global layer that satisfies the insertion criteria, as listed below, a new layer that is a "derivative" of the current dielectric layer is created. This new layer inherits all of the attributes of the dielectric layer and spans the height of the global layer. The insertion criteria are:

1. The dielectric layer must span the complete height of the global layer, and
2. The total area of the global layer is less than that of the substrate.

The second criterion is based on assumption 13 of the **CIF File Specifications** (section 6.2) that the x-y dimension of the substrate is used as the x-y dimension of the MCM. The dielectric layers fill in gaps between two layers as well as fill into global layers in which the total cumulative x-y cross-sectional area of all the member layers is less than that of the MCM. This is consistent with the actual manufacturing process because the signals are usually etched out of the dielectric block. The new CIF layers are then consolidated to the global model. Thus the final decomposition-consolidation step is completed.

Finally, the members of each global layer must inherit the appropriate attributes from the physical model. The strategy here is to loop through all the global layers. The attributes of all the member layers (obtained from the physical model) of each global layer are retrieved. Each attribute is then associated with the global model layer. The association is done through SAM interface operators. The global model is now complete.

9.2 Global Model Data Structure

Two major points govern the structure used to store the global model information: 1) Since all the global model information is derived from the physical model, information that do not change should not be duplicated. 2) Furthermore, each global layer is made up of one or more of the CIF layers defined in the physical model. The number of CIF

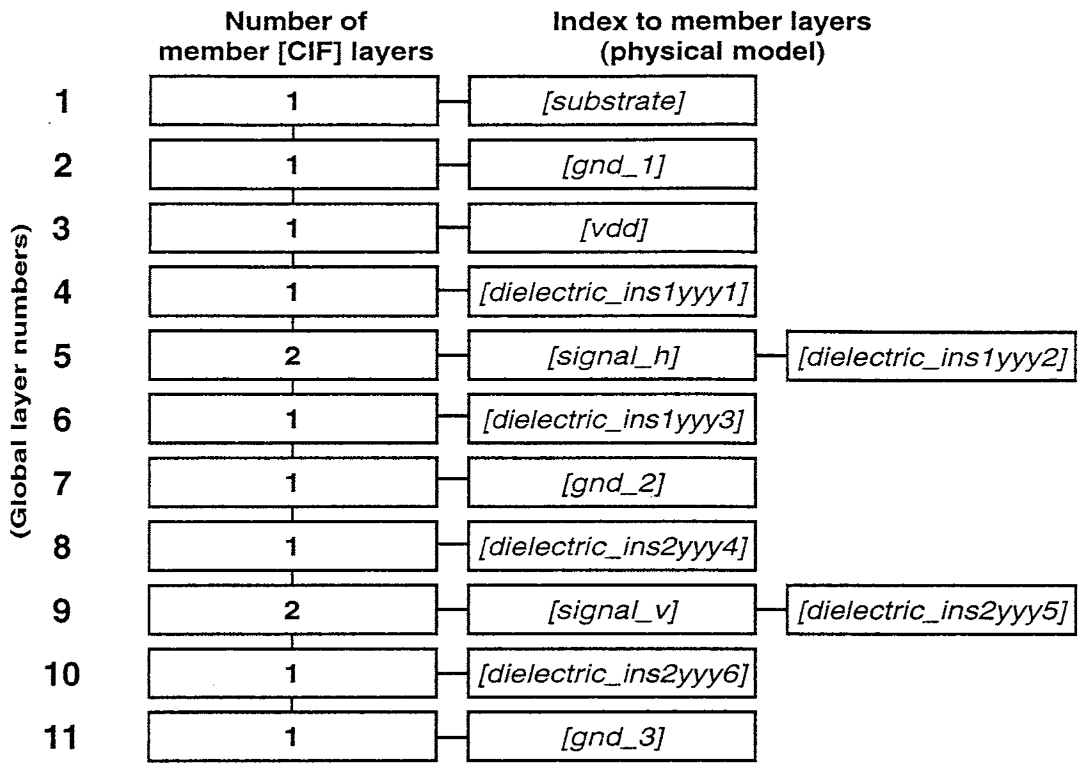


Figure 36 Data Structure of the Global Idealized Model

layers contained in one global layer may be different from layer to layer. A data structure used that satisfies these two requirements is a variable length two dimension index array. Figure 36 is a representation of the global idealized model of the 25–chip MCM design as previously mentioned. This global model is extracted from the information of the physical model, which is shown in Table 2. For comparison purposes, this table is repeated here, as Table 3, along side a tabulation of the global model information (Table 4). For this example model, there are a total of 11 global layers, extracted out of 12 CIF layers, excluding chips. Notice none of the vias appear in the global layer. Also, because the signal layers are embedded in the dielectric layers, each of the two dielectric layers are broken into three global layers. This separation produces a middle global layer consisting of signals and dielectrics, and a global layer of dielectrics on the top and bottom of this middle global layer. Within a global layer, the first position of the

CIF name	z-min ($\times 10^{-4}$)	z-max ($\times 10^{-4}$)	Area ($\times 10^{-4}$)	Attributes
substrate	0	6.0	59.4	*pointer*→
gnd_1	6.0	6.25	59.4	*pointer*→
via_ground	6.25	7.222	2.75e-03	*pointer*→
vdd	6.25	6.50	59.4	*pointer*→
dielectric_ins1	6.50	6.80	59.4	*pointer*→
via_power	6.50	7.222	2.95e-03	*pointer*→
signal_h	6.62	6.68	1.81922	*pointer*→
via_contact	6.68	6.98	3.294e-03	*pointer*→
gnd_2	6.80	6.86	59.4	*pointer*→
dielectric_ins2	6.86	7.16	59.4	*pointer*→
signal_v	6.98	7.04	0.790442	*pointer*→
gnd_3	7.16	7.22	59.4	*pointer*→
chip_10	7222	7.972	1.0	*pointer*→
chip_9	7.222	7.972	1.0	*pointer*→
chip_2	7.222	7.972	1.0	*pointer*→
:	:	:	:	:

Table 3 Physical Model Information Stored in a 25–Chip MCM Design

global index array indicates the number of member layers the global layer contains. The subsequent positions contain the indices to the CIF layers from the physical model. In this way, none of the information specified in the physical model is duplicated. This structure is also flexible enough to allow for any number of global layers as well any number of member layers within each global layer.

Global Layer	CIF name	z-min ($\times 10^{-4}$)	z-max ($\times 10^{-4}$)	Area ($\times 10^{-4}$)	Attributes
1	substrate	0	6.0	59.4	*pointer*$\rightarrow$
2	gnd_1	6.0	6.25	59.4	*pointer*$\rightarrow$
3	vdd	6.25	6.50	59.4	*pointer*$\rightarrow$
4	dielectric_ins1yyy1	6.50	6.62	59.4	*pointer*$\rightarrow$
5	signal_h	6.62	6.68	1.81922	*pointer*$\rightarrow$
	dielectric_ins1yyy2	6.62	6.68	57.58078	*pointer*$\rightarrow$
6	dielectric_ins1yyy3	6.8	6.80	59.4	*pointer*$\rightarrow$
7	gnd_2	6.80	6.86	59.4	*pointer*$\rightarrow$
8	dielectric_ins2yyy4	6.86	6.98	59.4	*pointer*$\rightarrow$
9	dielectric_ins2yyy5	6.98	7.04	58.60956	*pointer*$\rightarrow$
	signal_v	6.98	7.04	0.790442	*pointer*$\rightarrow$
10	dielectric_ins2yyy6	7.04	7.16	59.4	*pointer*$\rightarrow$
11	gnd_3	7.16	7.22	59.4	*pointer*$\rightarrow$

Table 4 Global Model Extracted From the Physical Model

10 SUMMARY AND FUTURE WORK

The first part of this thesis described a generalized framework that provides an environment in which analyses may define, organize, and use analysis attributes that are associated with appropriate model entities. This framework is the foundational structure for the SCOREC Attribute Manager (SAM) that also provides for convenient interactions between multiple analysis modules. Three major aspects of SAM has been discussed in detail: physical information, organizational information, and relational information.

Since the SAM information manager is a key element for the REPAS project, it must closely work with the model building efforts. The final five chapters discussed the approach in providing the modelling information for the global/local - thermal/thermomechanical analyses. The format of the input files were first presented, followed by a presentation of the entire preprocessing module from the point of view of the user. The remaining chapters presented the approach used in building the physical and global models. The physical model contains critical information for all four client analyses. In the process of building this physical model, SAM operators are called to create attributes from the three input files that were discussed in Chapter 6. These attributes are then associated with the physical model entities. Another model for the two global thermal/thermomechanical analyses is then built out of this physical model. Again, the corresponding attributes are created and associated with the global model. In building these two models, most of the required attributes for the client analyses are defined. Although the model building process for the two local analyses were not discussed here, the preprocessing step provides the foundation upon which the two local analyses models are built. So too, the preprocessing step creates the pertinent analysis attributes, organize them in a manner appropriate for the five client analyses, and associate the attributes to the correct analysis model. At the end of the preprocessing step, SAM is ready to return all the enquiries from the client analyses, thus fulfilling the role of an information manager.

As the attribute manager is a first effort to develop a generalized framework, this work can only provide a foundation for future refinements and evolutions to bring this underlying structure to its fullest potential. The remaining portions of this section describes some of the areas upon which the author feels can be improved.

10.1 General Structure Issues of SAM

The general structure of SAM that has been developed thus far provides a good foundation upon which all the other components of the manager can be built. As much as possible, the routines are written to be modular, extendable, and replaceable. Even so, there is no doubt that the data structure in general can be cleaned up and made more efficient, specifically in the following areas:

1. As SAM is being used more frequently, it is becoming apparent that more operators must be developed to meet the analysis needs. Work must be done to assess any missing functionalities that are needed for the current needs. Some operators not yet implemented are described in Section 4.2.
2. Consistency in naming conventions for routines and operators — This is needed for purposes of maintenance and readability. Maintenance becomes an important issue especially when the functionalities and features of SAM are expanded. A uniform naming convention contribute towards ease of maintenance as well as ease of use by applications. Towards the end of this work, a naming convention was beginning to take shape. For example, operators callable by C routines begin with

"**AT_**" whereas operators callable by FORTRAN routines begin with "**at_**". The next two letters following the "AT_" identifies the *action* of the operator. For example, creation operators use "**cr**", "**rt**" for retrieval operators, "**cp**" for operators that copy or duplicate information, and "**del**" for deletion operators. A good portion of the operators that are developed use this convention, while those developed earlier on in the development stage still have inconsistent names.

3. Inter-language Operators — Although increasingly more applications are being written in C and C++ programming languages, a large percentage of applications still use FORTRAN. Thus far, all of the interface operators are written in C and cater to be called from C routines. Although C-wrapper routines have been written for a number of the major interface operators, there is need for a complete set of C-wrapper routines for the entire suite of SAM operators. A wrapper routine is a function written in C with the arguments written specifically for interface between FORTRAN and C functions. The argument usage varies from one machine platform to another. The discussion of these differences is beyond the scope of this thesis. For more information regarding this, please consult the operating system users' manuals.

4. Garbage Collection and Memory Management — Currently, there are only two deletion routines. However, there is not a garbage collection function that properly frees the memories from the structures these deletion operators discard. This is becoming an increasingly important issue as the applications do more dynamic definition and allocations of new attributes and distributions — especially when a graphic user interface is in place for SAM.

10.2 Symmetry

Currently, only a limited number of symmetries can be represented. These symmetries (as listed in Section 3.1.2) are: 1-symmetric, 2-totally symmetric, 3-antisymmetric, 4-totally antisymmetric, 5-plane symmetry, 6-line symmetry, 7-point symmetry, 8-plane-line symmetry, 9-anisotropic, 10-orthotropic, 11-transversely isotropic, 12-isotropic. These symmetries are initially designed to describe the material types. There is another type of symmetries that has yet to be considered. One such type of symmetry is with respect to the setup of the analysis problem. For example, a model being solved might have some type of geometric symmetry or that the load distribution might be symmetric with respect to some axes. What is designed cannot adequately handle such situations. Representation of another type of symmetry in terms of "periodicity" is also lacking. In order to accommodate these other types of symmetries, the design must be broadened and/or completely redesigned.

With the current design, to describe the material tensor of an linear isotropic elastic material requires the user to input all 21 of the tensor components — even if more than half of those are zeroes. It is desirable to develop some kind of material templates and operators to handle material definitions. Some initial work has been started for the REPAS project. However, what is developed is very limited. The operator basically recognizes a keyword of LIEM (for Linear Isotropic Elastic Material) and automatically creates and fills in the tensor components. The components may be defined by a set of independent variables, for example, E and ν for modulus of elasticity and Poisson's ratio, respectively. Again, the material template work is incomplete. The most desirable is that a complete material handler module be designed, with operators to define, store, retrieve, and manipulate the material information needed for analysis.

10.3 Coordinate System

Currently, the capability to handle coordinate systems are in place. However, manipulation operators need to be developed. One such operator might take distributions from one coordinate system and transforms it to another. Another operator might take evaluated distribution results from one coordinate system and compares them to results in another coordinate system.

Also, as specified at the end of Section 3.1.5, operators need to be developed to allow for definitions of coordinate systems local to geometric model entities. For example, a uniform pressure distribution on a curved surface can be defined best with respect to the normal of the surface. This would require point-wise evaluation of the surface in its geometric space. As such this requires a close interface with the geometric model used. The set of operators must be able to define a unique coordinate system with available enquiry routines from the geometric modeler. Since the coordinate system will most likely be defined in terms of normal and tangents, logistic need to be worked out for the operators to define consistent directions of the tangents at every point on the surface.

10.4 Distribution

Distribution handling capabilities is one of the most important building blocks of SAM. For the most part, it is also one of the most well developed features of SAM. Even so, several enhancements are needed to make SAM a more robust manager.

1. The built-in function handling allows passing information from one procedure to another through SAM. A side benefit of the design is that the built-in function

that a distribution dependents on can perform a set of operations at the time of evaluation. For example, the built-in function may in turn call other procedures to generate/update some data with a specific updated piece of data returned. Thus not only can the built-in functions pass relevant data, they can be used to perform certain actions when evaluated. Despite these features, the information passed through the built-in function is currently limited to returning just a 1–D array of numbers. This may need to be expanded to accommodate multiple dimension arrays and/or tables of date — which may include character strings and or C structures.

2. At the time of evaluation, the calling routine must provide the values of all the undefined variables of the distribution. This requires that the calling routine knows exactly what it is asking for. The evaluation will fail if just one variable is undefined. An operator is needed to identify these undefined variables a priori to the calling routine. Furthermore, this operator (or a separate operator) should be able to identify whether the distribution in question is a numerical constant, a uniform distribution, or nonuniform distribution and pass this information to the calling routine.

3. Provision is made for distributions to be dependent on the value of the tensor components of another attribute. There is currently no general way to identify the specific tensor component of an attribute. Therefore, a tensorial component specification convention needs to be developed, along with appropriate enhancements to the parser and related routines, before the tensor component dependency can be used for distributions.

4. For those attributes not definable as a tensor (for example, composite layer orientations), the attribute components can be defined using a distribution list. A distribution

list, as detailed in Section 3.1.4, can be used to store distribution definitions local to the attribute. The current structure of the list is comparable to a 1–D array of distributions. For those applications that require, say, a 2–D table of values, the list becomes difficult to use. An expansion to this list functionality will definitely add to the robustness of SAM.

5. Finally, all distributions are assumed to be operating from one single unit of measurement, since this unit of distributions are not specified anywhere. The distribution data structure needs to set aside space to define the units of measurement of the distributions. Corresponding operators need to be enhanced with checking units, and new operators need to be developed to convert distributions from one unit to that of another.

10.5 Organization Issues of SAM

The SAM organization structure allows the application to put the potentially large number of attributes into manageable and reasonable orders. A suite of operators are developed for dynamic creation of such organization structure (Section 4.2). These same operators are used to read the user defined structure from I/O files, as described in Section 4.1.2. As such, a graphic user interface will make this definition process more interactive and increase the ease of use of SAM. The user interface will use the same interface operators as described in Chapter 4 to create, manipulate, store, and access the attribute information. Not only will such a user interface greatly enhance the usability of SAM, it will also allow for enforcing the rules of the manager and ensure correct specification of attributes.

10.6 Relational Issues of SAM

As detailed in the discussion on the design of the relational aspects of attributes (Section 2.3), a complete treatment of the subject requires extensive integration with geometric modelling systems. An initial list of geometric modelling functionalities required for augmentation is detailed in Appendix A. The functionalities can be separated into three categories: 1) return information about specified geometric and topological entities, 2) define and create necessary geometric and topological entities, and 3) remove, transform, and in general manipulate existing geometric and topological entities. As such, each one of these functionalities is dependent on the particular modelling system being used. Some work must be done to determine and ensure that the listed set of functionalities are sufficient for the needs of SAM.

Secondly, two issues regarding the augmentation process need to be addressed: 1) rules must be developed to keep track of the attributes and associated auxiliary geometries from creation through the evolution of the augmented models, and 2) rules and operators must be developed to handle the augmentation logistics of what information to retain, what to delete, and how to switch from one augmentation level to another. As can be seen from the discussion from Section 2.3, these two issues are not trivial to solve. Only a basic association of attributes to model entities has been implemented. Much effort is still needed to develop fully the relational capabilities into a complete functional module.

Appendix A Augmentation Operators

Listed in this appendix is an initial compilation of the operators needed for the idealization and augmentation processes. These operators are separated into categories of 1) interrogation of existing geometric and topological data base, 2) definition of geometric entities, 3) manipulation of geometric/topological entities.

Where there are existing operators, the operators are described using the following naming convention:

- ☐ **BOLD-CAP** = routine name
- ☐ *italics* = input parameter
- ☐ **bold** = output parameter
- ☐ ***bold-italics*** = input and output parameter

A.1 Interrogation of the Existing Geometric and Topological Data Base

Listed below is a set of operators to query the geometric modeler for topological and geometric information.

1. Operator that calculates the mass of a given region.
2. Operator that calculates the moment of inertia of a given region.
3. Operator that calculates the mid-planes of a region.
4. **CKENFC** (*face_index*, *entity_index*, *entity_type*, **adjacency_flag**)

   Check if entity, *entity_index*, of type, *entity_type*, is topologically adjacent to the specified face, *face_index*.

5. **FACUSE** (*face_index*, **region_on_face_use1**, **region_on_face_use2**)

   Returns the indices or pointers to the two topological regions, **region_on_face_use1**, **region_on_face_use2**, sharing the face, *face_index*.

6. **FCEVRT** (*face_index*, ***face_vertex***)

   Returns the next vertex, ***face_vertex***, of the face, *face_index*, given the current vertex, ***face_vertex***

7. **GETENT** (*entity_type*, ***entity_index***)

   Returns the next topological entity, ***entity_index***, of type, *entity_type*, for the model.

8. **SUBENT** (*entity_type*, *entity_index*, ***lower_order_entity***)

   Returns the lower order adjacency, ***lower_order_entity***, for the given topological entity, *entity_index*, of type, *entity_type*.

9. **GTFCEG** (*edge_index*, ***face_index***)

   Returns the next face, ***face_index***, of the given current edge, *edge_index*.

10. **CLSFPT** (*face_index*, *point_to_project*, *seed_point*, *seed_parameter*,

    projected_point, **parameter_of_projected_point**, **error_flag**)

    Finds the closest point on the face, *face_index*, to a point off the face, *point_to_project*.

11. **DISFCE** (*face_index*, *point_to_project*, *seed_point*, *seed_parameter*,

    distance_to_face, **error_flag**)

    Finds the closest distance to the face, *face_index*, from a point off the face, *point_to_project*.

12. **FCEPRM** (*face_index*, *point_on_face*, *parameter_of_point_on_entity_index*,

    entity_type, *entity_index*, ***parameter_on_face***)

Calculates parameter values, ***parameter_on_face***, on a face, *face_index*, of a given point, *point_on_face*, which has the parameter value, *parameter_of_point_ on_entity_index*, on entity, *entity_index*, of type*entity_type.*

13. **FCENOR** (*face_index*, *point_on_face*, *parameter_of_point_on_entity_index*, *entity_type*, *entity_index*, **normal_to_face**)

    Calculates the normal vector, **normal_to_face**, to a face, *face_index*, at a given point, *point_on_face*, which has the parameter value, *parameter_of_point_on_entity_index*, on entity, *entity_index*, of type *entity_type.*

14. FDINOT (*spatial_point*, *max_number_of_regions*, **Number_of_regions_found**, **list_of_regions_founds**)

    Finds all regions about the point *spatial_point*

15. **FREEVT** (*vertex_index*, **free_status**)

    Indicates whether the vertex, *vertex_index*, is constrained at a single location, or merely constrained to a loci of points.

16. **GPOINT** (*entity_type*, *entity_index*, **spatial_point**, *parameter_value*)

    Calculates the spatial coordinates, **spatial_point**, on an entity, *entity _index*, of type *entity_type* which corresponds to the given parameter value(s) *parameter_value.*

17. **MODTOL** (**modeler_tolerance**)

    Returns the tolerance, **modeler_tolerance**, used by the modeler.

18. **ENTPRM** (*entity_index*, *entity_type*, *spatial_point*, *parameter_value*, **begin_parameter**, **end_parameter**)

    Returns the lowest, **begin_parameter**, and highest **end_parameter**, parameter values of a given model entity, *entity_index*, of type, *entity_type.*

19. **SMTPRM** (*face_index*, *spatial_point_1*, *parameter_spatial_point_1*, *spatial_point_2*, *parameter_spatial_point_2*, *line_parameter_value*, **spatial_point**, **parameter_value_spatial_point**, **error_flag**)

    Returns a point, **spatial_point**, on a given face, *face_index*, which is at parameter value, *line_parameter_value*, on a parametric line segment which is bounded by *spatial_point_1*, and *spatial_point_2*.

20. **INTEPL** (*edge_index*, *begin_point*, *begin_parameter*, *end_point*, *end_parameter*, *spatial_point_on_plane*,*normal_vector_plane*,*max_number_of_intersections*, **spatial_intersection_points**, **parameter_intersection_points**, **type_of_found_intersections**, **number_of_intersections_found**)

    Returns the intersection points, **spatial_intersection_points**, of a given model edge T_i^1, *edge_index*, with a plane defined by a point on the face, *spatial_point_on_plane*, and a normal vector to the face, *normal_vector_plane*. The type of intersection points must be also returned. The number of returned intersection points must be limited to *max_number_of_intersections.*

21. **INTFCE** (*face_index*, *begin_point*, *end_point*, *max_number_of_intersections*, **number_of_intersections_found**, **type_of_intersections_found**, **spatial_intersection_points**, **parameter_intersection_points**)

    Returns the intersection points, **spatial_intersection_points**, of a given model face, *face_index*, with a line segment defined by the two points *begin_point* and *end_point.* Type of intersection points must be also returned. The number of returned intersection points must be limited to *max_number_of_intersections*

A.2 Definition of Geometric Entities

Regardless of which geometric modeler is used, the following is a list of the basic capabilities the geometric modeler needed to be able to support and have appropriate interface operators for applications such as SAM to use.

1. create point
2. create curves
3. create surface patches
4. create solids
5. create primitives (box, triangle, torus, etc)
6. create "features"(fillets, holes, etc)

A.3 Manipulation of Geometric/Topological Entities

Listed below is a summary of functions that operators needed to support for the manipulation of geometric/topological entities.

1. REMOVE entity (point, curve, patch, solid, "feature")
2. DISASSOCIATE entity from the model (*such as when creating auxiliary geometries, they should be "disassociated" from the original model.*)
3. INTERSECT an entity with another entity
4. SUBTRACT an entity from another entity of equal or higher order (*basically an intersection and then removal of the intersection*)
5. UNION of an entity with another entity

6. SPLIT an entity with another entity (*intersect the two entities and separate the target entity at the intersection.*) An example of splitting an edge with another edge is shown in Figure 37 below.

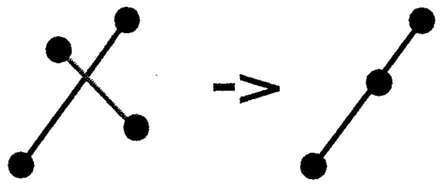


Figure 37 An Edge Split at Intersection From a Second Edge

7. IMPRINT an entity on another entity of equal or higher order. An example of an edge imprinting on another edge is shown in Figure 38 below. The actions done are

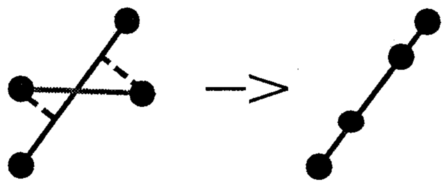


Figure 38 An Edge IMPRINTed by a Second Edge

basically to project the working entity onto the target entity with respect to a certain direction and SPLIT the target entity at the projection points.

8. MERGE two redundant entities into one single entity.

Appendix B Summary of SAM Application Interface Operators

List in this appendix is a summary of the SAM application interface operators. The following naming convention is used:

- ☐ normal-font() = routine name
- ☐ *italic*() = routine that has a return value of the given type
- ☐ **bold** = input parameter
- ☐ *italic* = output parameter
- ☐ ***bold-italics*** = input and output parameter

B.1 Attribute Manager Set-up/Wrap-up

The required environment variable to use with SAM operators is

```
setenv ATT_HOME "<the full path to the attribute directory>"
```

Those routines using any of the interface operators must include the following header in the calling routine:

$ATT_HOME/attribute/include/header.h

1. void AT_setup(char ***filename**)

   This routine allocates and initializes the variables needed by the attribute manager. This operator also checks to see if *filename.lib*, *filename.org*, and *filename.rel* exist. If so, the contents of the files are read into the attribute manager. If only new attributes are intended to be created, care must be taken to first remove these files before calling AT_setup() to avoid duplication of attributes. The present restriction is that if one of the three files is present, then all three must be present.

2. void AT_stop(char ***filename**)

   This routine writes to file everything that the attribute manager has in memory, including the attribute definitions, organization structure, and attribute associations. The files to which these attributes are written are *filename.lib*, *filename.org*, and *filename.rel*. Any files with the same names are overwritten.

B.2 Physical Creation Operators

1. *ATTRIBUTE *AT_cratt* (char ***attribute_id**)

   This routine creates an attribute with the given id. The pointer to the newly created attribute is returned.

2. *MULTIPLIER *AT_crmult* (char ***multiplier_id**, double **multiplier_value**)

   This routine creates a multiplier with the given id and value. The pointer to the newly created multiplier is returned.

3. void tensor_order (long **retr_store_flag**, ATTRIBUTE ***attribute_pointer**,
   long **tensor_order*)

   This routine retrieves/stores the tensor order information, dependent on what **retr_store_flag** is.

4. void coord_sys_ptr (long **retr_create_flag**, ATTRIBUTE ***attribute_pointer**,
   COORDINATE_SYS **coord_sys_ptr*)

   This routine retrieves the pointer to the coordinate system used by the given attribute, pointed to by attribute_pointer. If the command is to create, a pointer is assigned to a new coordinate system and returned to the calling routine. The calling routine is responsible to call **csys_info**() and **csys_func**() to complete the definition of the new coordinate system.

5. void csys_func (long **retr_store_flag**, COORDINATE_SYS ***coord_sys_pointer**,

   DISTRIBUTION ***func_relation*, long *func_position*,

   char ***func_string*)

   This routine retrieves or stores the pointers to the current function that defines the coordinate system with respect to the given base coordinate system. If the *retrieve* option is selected, the function returned is the next available function or a NULL if the end of the function list is reached. If the *store* option is selected, the next coordinate system function is stored to the data structure.

6. void csys_info (long **retr_store_flag**, COORDINATE_SYS ***coord_sys_pointer*,

   long **coord_sys_type*, float ***coord_origin*,

   long **coord_sys_dimension*, char ***coord_sys_id*,

   COORDINATE_SYS ***base_coord_sys_pointer*,

   DISTRIBUTION ****func_ptr*)

   Given the particular pointer to the coordinate system, this routine returns/stores from/into the attribute data base the coordinate system type, the origin coordinates of the current coordinate system, the dimension of the coordinate system, the coordinate system ID, the pointer to the base coordinate system, and the pointer to an array of distribution functions of coordinate axes

7. void symmetry_ptr (long **retr_store_flag**, ATTRIBUTE ***attribute_pointer**,

   SYMMETRY ***symmetry_pointer*)

   This routine retreives the pointer to the symmetry information or stores the information into the data base. Note, this routine must be called first (before any calls to the other symmetry related creation operators) since memory for each new symmetry is created here.

8. void symmetry_type (long **retr_store_flag**, SYMMETRY **symmetry_pointer*,

   long **symmetry_type*)

   This routine stores/retreives the symmetry type to/from the data base.

9. void symm_info (long **retr_store_flag**, SYMMETRY **sym_pointer,* long *lrflag*,

   SYMMETRY_INFO **lr_sym_pointer*, float **lr_sym_info*)

   This routine stores/retrieves the information for a single symmetry into/from the data base. The implemented routine has not been tested.

10. void dist_comp (long **retr_store_flag**, ATTRIBUTE ***attribute_pointer**,

    struct comp_distrib ***distrib_comp_pointer*)

    This routine gets the pointer to the distribution of the next tensor component.

11. *DISTRIBUTION *parser* (long **input_flag**, void ***input_source**,

    ATTRIBUTE ***attribute_pointer**, long **store_flag**)

    Given an equation in character string form, this routine parses the equation and returns the parsed equation as a distribution.

12. void AT_mktencomp (char ***input_equation**, ATTRIBUTE ***attribute_pointer**)

    This routine parses the given equation and attaches the resulting distribution to a new tensor component of the given attribute. No provision is made to distinguish which component of the tensor is being created. The order that **AT_rtdstcomp()** returns the tensor components is the order in which the input equations are given in this routine.

13. void AT_mklist (ATTRIBUTE ***attribute_pointer**, char ****equations**, int **num_eqns**)

    Given an array of equations, this routine parses each equation and attaches the parsed distribution to a list, which is attached to the attribute given in attribute_pointer.

14. void AT_mktmplat (ATTRIBUTE ***attribute_pointer**, char ****equations**,

    int **num_eqns**)

    Given a pointer to an attribute and a set of defining equations, this routine creates and assigns distributions, tensor order, and symmetries using a previously defined template. The appropriate template is selected based on the specific label and attribute type on the given attribute. This implies that the appropriate labels (and coordinate system) and type are assigned prior to the calling of this routine. What has been implemented as of this writting is only a set of stiffnesses for the following materials: **LOEM** for Linear Orthotropic Elastic Material and **LIEM** for Linear Isotropic Elastic Material. To use this template operator, the first label of the attribute must be one of the two keywords just mentioned. Also, the attribute type must be **stiffness**.

B.3 Organizational Creation Operators

1. *SET *AT_crset* (char ***set_id**)

   This routine creates a set with the given id. The pointer to the newly created set is returned.

2. *GROUP *AT_crgrp* (char ***group_id**)

   This routine creates a group with the given id. The pointer to the newly created group is returned.

3. *CASE *AT_crcas* (char ***case_id**)

   This routine creates a case with the given id. The pointer to the newly created case is returned.

4. void AT_bultorgn (char ***parent_id**, ORGNCODE **parent_type**, char ***child_id**, ORGNCODE **child_type**, char ****multitplier_id**, double ***multiplier_values**, int **num_mults**)

   This routine builds a segment of the organizational structure with the given parent, child, and the multipliers to the child.

B.4 Relational Creation Operators

void AT_astopat (ATTRIBUTE ***attribute_pointer**, long **model_entity_number**, char ***model_entity_string**, long **model_entity_type**, char ***model_name**)

This function associates a given model entity with the attribute to which the given pointer points. It takes as input a pointer to the attribute, the entity identification in numerical or string form, the entity type, and the model identification.

B.5 Attribute Retrieval Operator

void AT_rtatts (long **retrieval_method**, char ***case**, char ***group**, char ***set**, char ***geom_model_name**, long **model_entity_type**, long **model_entity_integer**, char ***model_entity_string**, char ***attribute_type**, long **array_size**, ATTRIBUTE *****attribute_pointer***, double **mult_value*, long **number_retrieved*)

This routine returns pointers to attributes which satisfy the organizational and relational condition specified in the input. Any information specified in the input (arguments third through tenth) are used to narrow down the selection of available attributes. The third, fourth, and fifth arguments are used to restrict the search to the given branches in the organizational structure. The sixth through tenth arguments are used to pin-point the desired attributes in the already shrunken selection of qualified attributes.

At the time of writing, this routine has not been tested. However, another routine named **attptr_C()** had been used extensively for attribute retrieval. **AT_rtatts()** and **attptr_C()** are essentially the same with one major difference: **attptr_C()** returns (geom_model_name, model_entity_type, model_entity_integer, model_entity_string, and attribute_type) in addition to (attribute_pointer and mult_value). This added feature is desirable if not for one important flaw in the return logic. When an attribute is associated with more than one model entity, only one model entity is returned. The same flaw occurs for both the geom_model_name and model_entity_type. Because of this flaw, another operator, **AT_rtatts()**, is developed that returns only the attribute_pointers and mult_value.

B.6 Organization Retrieval Operators

1. void AT_rtnxorgn (ORGNCODE **node_type**, void *****node_pointer***)

   This routine returns the next organizational element in the organizational structure with the given organizational element type, such as a case, group, set, or attribute.

2. void AT_rtnxchild (void ***node_pointer**, ORGNCODE **node_type**,

   void *****child_pointer***, ORGNCODE *child_type*)

This routine returns the next child pointer and child type of a given node pointer and node type.

3. *ATTRIBUTE *AT_rtatt* (char ***attribute_id**)

   This routine retrieves the pointer to the attribute that has the same id as given.

4. *SET *AT_rtset* (char ***set_id**)

   This routine retrieves the pointer to the set that has the same id as given.

5. *GROUP *AT_rtgrp* (char ***group_id**)

   This routine retrieves the pointer to the group that has the same id as given.

6. *CASE *AT_rtcas* (char ***case_id**)

   This routine retrieves the pointer to the case that has the same id as given.

7. void AT_rtmprod (void ***node_pointer**, ORGNCODE **node_type**,
   void ***child_pointer**, ORGNCODE **child_type**,
   double **multiplier_value*)

   This routine had not been implemented yet. When it is implemented, this routine will return the product of all the multipliers between two given connected nodes in the organizational structure. Hierarchically, the node must be of a type higher than that of the child.

8. void AT_rtmults(void ***node_pointer**, ORGNCODE **node_type**,
   void ***child_pointer**, ORGNCODE **child_ type**,
   long ****number_of_multipliers***, char ***multiplier_id*,
   double **multiplier_value*)

   This routine had not been implemented yet. When it is implemented, this routine will return the identification and value of all the multipliers between two given connected nodes in the organizational structure. Hierarchically, the node must be of a type higher than that of the child.

9. void AT_rtnodeid (void ***node_pointer**, ORGNCODE **node_type**, char ***node_id*)

   This routine returns the identification of a node pointed to by the given node pointer of the given node type. For example, if node = attribute (**node_type** = ATATT), **node_pointer** = ATTRIBUTE_POINTER.

B.7 Attribute Physical Information Retrieval Operators

1. *char *AT_rtatid* (ATTRIBUTE ***attribute_pointer**)

   This routine retrieves the id of the attribute pointed to by the given attribute pointer.

2. void AT_rtlabels (ATTRIBUTE ***attribute_pointer**, long ****number_of_labels***,
   char ***attribute_labels*)

   This routine returns all of the labels of the attribute pointed to by the given attribute pointer.

3. *char *AT_rtnxatlab* (ATTRIBUTE ***attribute_pointer**,
   Generic_List_Cells ****label_pointer***)

   This routine retrieves the next label of the given attribute. The id of the attribute is excluded from the definition of a "label."

4. void AT_rtatype (ATTRIBUTE ***attribute_pointer**, char ***attribute_type*)

   This routine retrieves the attribute type of the given attribute.

5. void AT_rtenord (ATTRIBUTE ***attribute_pointer**, long **tensor_order*)

   This routine returns the tensor order of the attribute pointed to by the given attribute pointer. The name being used now is **tensord_C()**. This name should be changed in the near future to **AT_rtenord**() so that consistent naming convention may be used.

6. void AT_rtcsptr(ATTRIBUTE ***attribute_pointer**,

   COORDINATE_SYS ***coord_sys_pointer*)

   This routine returns the pointer to the coordinate system on which the attribute, pointed to by the given attribute pointer, is defined. The function is currently named as **coordsys_ptr**(). This name should be renamed to **AT_rtcsptr**() in the near future so that consistent naming convention may be used.

7. void AT_rtcsinfo(COORDINATE_SYS ****coord_sys_pointer***,

   char ****coordinate_sys_id***, long **coord_system_type*,

   long **coord_system_dimension*, float ***coord_origin*,

   struct comp_distrib ***function_relation*,

   COORDINATE_SYS ***base_coord_system_pointer*)

   This routine returns the information of the coordinate system pointed to by the coordinate system pointer.

8. void AT_rtsyminfo(long **retrieval_method**, ATTRIBUTE ***attribute_pointer**,

   long ***symmetry_type***, long ****info_array*,

   long **num_of_sym_retrieved*, long **num_of_info_pieces*)

   This routine has not yet been implemented. When it is, it will return the symmetry information which satisfies the condition set in RETRIEVAL_METHOD.

9. void AT_rtdstcomp (long **retrieval_method**, ATTRIBUTE ***attribute_pointer**,

   double **mult_value**, COORDINATE_SYS ***coord_sys_pointer**,

   char ****variable_character_array**,

double ***variable_value_array**, long **number_of_var**,

long ****number_of _components***, long **components_defined*,

double **component_value_array*)

This routine returns the final values (in the given coordinate system and with all the relevant multipliers taken into account) of the distribution components of the attribute pointed to by the attribute pointer. All the variables required for evaluation of the distribution are given as input argument in the variable arrays.

B.8 Attribute Manipulation Operators

1. void AT_delorgn (ATTRIBUTE ***attribute_pointer**)

   This routine disassociates the organizational specification from the attribute. Because the organizational elements might be used else where, they are not deleted and allocated memory is not freed in this routine. This is done in another deletion routine.

2. void AT_delreln (ATTRIBUTE ***attribute_pointer**)

   This routine disassociates all relational specifications from the attribute.

3. *ATTRIBUTE *AT_cpyatt* (char ***old_name**, char ***new_name**)

   This routine copies all the components of a given attribute into a new attribute of a new name as given and returns the pointer to the newly created attribute.

Appendix C Example REPAS Output Files

This appendix presents some excerpts of sample output files from the REPAS project. The sample is taken from a 25 chip MCM design. These three output files correspond to the three input files needed by the SCOREC Attribute Manager and are used as input files for SAM at the start of each of the four global/local client analyses. The first file presented, *25chip.lib*, contains a partial listing of the physical information of the attributes defined for a typical run of the 25–chip example. The second file, *25chip.org*, contains a partial description of the organization structure used for the example run. Finally, the third file, *25chip.rel*, contains a partial listing of the relational information associating the defined attributes to the appropriate model entities.

C.1 25chip.lib

```
*distribution
ZERO = 0

*Multiplier
     label:  sFactor
     value:  1e-08

*Multiplier
     label:  Mdefault
     value:  1

*Coordinate System
     id:  global
     dimension:  3
     type:  1
     origin:  (0, 0, 0)
     function:  $1=1
     function:  $2=1
     function:  $3=1

*npins@pins
 labels:
 type:  npins
 tensor_order:  0
 symmetry:
 distribution:
  component:  168

*stiffness@pins
 labels:  LIEM
 type:  stiffness_pins
 tensor_order:  0
 list:  E = 1.3e11, nu
= 0.34
 symmetry:
```

```
 distribution:
  component:

*pin_size@pins
 labels:
 type:  pin_size
 tensor_order:  -1
 list:  500000, 40000,
40000

*stiffness@solder_bumps
 labels:  LIEM
 type:  stiffness
 tensor_order:  4
 list:  E = 3e10, nu =
0.2
 coord_sys:  global
 symmetry:
 distribution:
  component:  E*(1-
nu)/(1-nu-2*nu^2)
  component:  E*nu/(1-
nu-2*nu^2)
  component:  E*nu/(1-
nu-2*nu^2)
  component:  ZERO
  component:  ZERO
  component:  ZERO
  component:  E*(1-
nu)/(1-nu-2*nu^2)
  component:  E*nu/(1-
nu-2*nu^2)
  component:  ZERO
  component:  ZERO
  component:  ZERO
```

```
  component:  E*(1-
nu)/(1-nu-2*nu^2)
  component:  ZERO
  component:  ZERO
  component:  ZERO
  component:
E/(2*(1+nu))
  component:  ZERO
  component:  ZERO
  component:
E/(2*(1+nu))
  component:  ZERO
  component:
E/(2*(1+nu))

*thermal_expand@solder_bumps
 labels:  LIM
 type:  thermal_expand
 tensor_order:  2
 coord_sys:  global
 symmetry:
 distribution:
  component:  1.0e-5

*thermal_cond@solder_bumps
 labels:  LIM
 type:  thermal_cond
 tensor_order:  2
 coord_sys:  global
 symmetry:
 distribution:
  component:  36

*sol_diam@solder_bumps
 labels:
```

```
 type: sol_diam
 tensor_order: 0
 symmetry:
 distribution:
  component: 14000

*sol_height@solder_bumps
 labels:
 type: sol_height
 tensor_order: 0
 symmetry:
 distribution:
  component: 8500

*GT_amp
 labels:
 type: temp_expand
 tensor_order: 0
 symmetry:
 distribution:
  component:
GT_amp($1,$2,$3)

*GS_disp
 labels:
 type: displacement
 tensor_order: 1
 coord_sys: global
 symmetry:
 distribution:
  component:
GS_disp($1,$2,$3)
  component:
GS_disp($1,$2,$3)
  component:
GS_disp($1,$2,$3)

*LT_temp
 labels:
 type: temperature
 tensor_order: 0
 symmetry:
 distribution:
  component:
LT_temp($1,$2,$3)

*GT_temp
 labels:
 type: temperature
 tensor_order: 0
 symmetry:
 distribution:
  component:
GT_temp($1,$2,$3)


*majorDir@layer1
 labels:
 type: maj_dir
 tensor_order: 0
 symmetry:
 distribution:
  component: 0

*geomInfo@layer1
 labels:
 type: averages
 tensor_order: -1
 list: 0, 0, 0

*numOfPhyLayers
 labels:
 type: nlayers
 tensor_order: 0
 symmetry:
 distribution:
  component: 11

*numberOfChips
 labels:
 type: nchips
 tensor_order: 0
 symmetry:
 distribution:
  component: 25

*thickness@DIELECTRIC_INS1YYY2
 labels:
 type: thickness
 tensor_order: -1
 list: 66200, 66800

*thickness@DIELECTRIC_INS1YYY1
 labels:
 type: thickness
 tensor_order: -1
 list: 65000, 66200

*thermal_cond@DIELECTRIC_INS1
 labels: LIM
 type: thermal_cond
 tensor_order: 2
 coord_sys: global
 symmetry:
 distribution:
  component: 0.2

*thickness@DIELECTRIC_INS1
 labels:
 type: thickness
 tensor_order: -1
 list: zmin = 65000,
zmax = 68000

*stiffness@GND_1
 labels: LIEM
 type: stiffness
 tensor_order: 4
 list: E = 1.3e11, nu
= 0.34
 coord_sys: global
 symmetry:
 distribution:
  component: E*(1-
nu)/(1-nu-2*nu^2)
  component: E*nu/(1-
nu-2*nu^2)
  component: E*nu/(1-
nu-2*nu^2)
  component: ZERO
  component: ZERO
  component: ZERO
  component: E*(1-
nu)/(1-nu-2*nu^2)
  component: E*nu/(1-
nu-2*nu^2)
  component: ZERO
  component: ZERO
  component: ZERO
  component: E*(1-
nu)/(1-nu-2*nu^2)
  component: ZERO
  component: ZERO
  component: ZERO
  component:
E/(2*(1+nu))
  component: ZERO
  component: ZERO
  component:
E/(2*(1+nu))
  component: ZERO
  component:
E/(2*(1+nu))

*thermal_expand@GND_1
 labels: LIM
 type: thermal_expand
 tensor_order: 2
 coord_sys: global
 symmetry:
 distribution:
  component: 1.7e-5

*thermal_cond@GND_1
 labels: LIM
 type: thermal_cond
 tensor_order: 2
 coord_sys: global
 symmetry:
 distribution:
  component: 398

*thickness@GND_1
 labels:
 type: thickness
 tensor_order: -1
 list: zmin = 60000,
zmax = 62500

*offset@gs
 labels:
 type: org_offset
 tensor_order: 1
 coord_sys: global
 symmetry:
 distribution:
  component: -0.006
  component: 0
  component: 0

*offset@gt
 labels:
 type: org_offset
 tensor_order: 1
 coord_sys: global
 symmetry:
 distribution:
  component: -0.061
  component: -0.027
  component: 0

*layer_width
 labels:
 type: layer_width
 tensor_order: 0
 symmetry:
 distribution:
  component: 0.054

*layer_length
 labels:
 type: layer_length
 tensor_order: 0
 symmetry:
 distribution:
  component: 0.11

*heat_transfer@SUBSTRATE
 labels:
 type: heat_trans
 tensor_order: -1
 list: 1, 1300, 25

*stiffness@SUBSTRATE
 labels: LIEM
 type: stiffness
 tensor_order: 4
 list: E=1.92e11, nu =
0.22
 coord_sys: global
 symmetry:
 distribution:
  component: E*(1-
nu)/(1-nu-2*nu^2)
  component: E*nu/(1-
nu-2*nu^2)
  component: E*nu/(1-
nu-2*nu^2)
  component: ZERO
  component: ZERO
  component: ZERO
  component: E*(1-
nu)/(1-nu-2*nu^2)
  component: E*nu/(1-
nu-2*nu^2)
  component: ZERO
  component: ZERO
  component: ZERO
  component: E*(1-
nu)/(1-nu-2*nu^2)
  component: ZERO
  component: ZERO
  component: ZERO
  component:
E/(2*(1+nu))
  component: ZERO
  component: ZERO
  component:
E/(2*(1+nu))
  component: ZERO
  component:
E/(2*(1+nu))

*thermal_expand@SUBSTRATE
 labels: LIM
 type: thermal_expand
 tensor_order: 2
 coord_sys: global
 symmetry:
 distribution:
  component: 3.14e-6

*thermal_cond@SUBSTRATE
 labels: LIM
 type: thermal_cond
 tensor_order: 2
 coord_sys: global
 symmetry:
 distribution:
  component: 84

*thickness@SUBSTRATE
 labels:
 type: thickness
 tensor_order: -1
 list: zmin = 0, zmax
= 60000

*center@CHIP_25
 labels:
 type: chip_center
 tensor_order: 1
 coord_sys: global
 symmetry:
 distribution:
  component: -0.006
  component: 0.018
  component: 7.222e-12

*chip_length@CHIP_25
 labels:
 type: chip_length
 tensor_order: 0
 symmetry:
 distribution:
  component: 0.01

*chip_width@CHIP_25
 labels:
 type: chip_width
 tensor_order: 0
 symmetry:
 distribution:
  component: 0.01

*nsold_bmp@CHIP_25
 labels:
 type: nsold_bmp
 tensor_order: 0
 symmetry:
 distribution:
  component: 0

*pwr_via_area@CHIP_25
 labels:
 type: pwr_via_area
 tensor_order: 0
 symmetry:
 distribution:
  component: 5e8

*sig_via_area@CHIP_25
 labels:
 type: sig_via_area
 tensor_order: 0
 symmetry:
 distribution:
  component: 1.25e9

*gnd_via_area@CHIP_25
 labels:
 type: gnd_via_area
 tensor_order: 0
 symmetry:
 distribution:
  component: 5e8

*therm_via_area@CHIP_25
 labels:
 type: therm_via_area
 tensor_order: 0
 symmetry:
 distribution:
  component: 0

*nth_vias@CHIP_25
 labels:
 type: nth_vias
 tensor_order: 0
 symmetry:
 distribution:
  component: 0

*heat_transfer@CHIP_25
 labels:
 type: heat_trans
 tensor_order: -1
 list: 1, 10, 70

*power_density@CHIP_25
 labels:
 type: power_density
 tensor_order: 0
 symmetry:
 distribution:
  component: 5e4

*stiffness@CHIP_25
 labels: LIEM
 type: stiffness
 tensor_order: 4
 list: E=8.47e10, nu =
0.22
 coord_sys: global
 symmetry:
 distribution:
  component: E*(1-
nu)/(1-nu-2*nu^2)
  component: E*nu/(1-
nu-2*nu^2)
  component: E*nu/(1-
nu-2*nu^2)
  component: ZERO
  component: ZERO
  component: ZERO
  component: E*(1-
nu)/(1-nu-2*nu^2)
  component: E*nu/(1-
nu-2*nu^2)
  component: ZERO
  component: ZERO
  component: ZERO
  component: E*(1-
nu)/(1-nu-2*nu^2)
  component: ZERO
  component: ZERO
  component: ZERO
  component:
E/(2*(1+nu))
  component: ZERO
  component: ZERO
  component:
E/(2*(1+nu))
  component: ZERO
  component:
E/(2*(1+nu))

*thermal_expand@CHIP_25
 labels: LIM
 type: thermal_expand
 tensor_order: 2
 coord_sys: global
 symmetry:
 distribution:
  component: 5.7e-6

*thermal_cond@CHIP_25
 labels: LIM
 type: thermal_cond
 tensor_order: 2
 coord_sys: global
 symmetry:
 distribution:
  component: 58

*thickness@CHIP_25
 labels:
 type: thickness
 tensor_order: -1
 list: zmin = 72220,
zmax = 79720

*thermal_expand@VIA_CONTACT
 labels: LIM
 type: thermal_expand
 tensor_order: 2
 coord_sys: global
 symmetry:
 distribution:
  component: 1.7e-5

*thermal_cond@VIA_CONTACT
 labels: LIM
 type: thermal_cond
 tensor_order: 2
 coord_sys: global
 symmetry:
 distribution:
  component: 398

*thickness@VIA_CONTACT
 labels:
 type: thickness
 tensor_order: -1
 list: zmin = 66800,
zmax = 69800

*min_wire_pitch@SIGNAL_V
 labels:
 type: min_wire_pitch
 tensor_order: 0
 symmetry:
 distribution:
  component: 40

*min_wire_width@SIGNAL_V
 labels:
 type: min_wire_width
 tensor_order: 0
 symmetry:
 distribution:
  component: 20

*stiffness@SIGNAL_V
 labels: LIEM
 type: stiffness
 tensor_order: 4
 list: E = 1.3e11, nu
= 0.34
 coord_sys: global
 symmetry:
 distribution:
  component: E*(1-
nu)/(1-nu-2*nu^2)
  component: E*nu/(1-
nu-2*nu^2)
  component: E*nu/(1-
nu-2*nu^2)
  component: ZERO
  component: ZERO
  component: ZERO
  component: E*(1-
nu)/(1-nu-2*nu^2)
  component: E*nu/(1-
nu-2*nu^2)
  component: ZERO
  component: ZERO
  component: ZERO
  component: E*(1-
nu)/(1-nu-2*nu^2)
  component: ZERO
  component: ZERO
  component: ZERO
  component:
E/(2*(1+nu))
  component: ZERO
  component: ZERO
  component:
E/(2*(1+nu))
  component: ZERO
  component:
E/(2*(1+nu))

*thermal_expand@SIGNAL_V
 labels: LIM
 type: thermal_expand
 tensor_order: 2
 coord_sys: global
 symmetry:
 distribution:
  component: 1.7e-5
```

```
*thermal_cond@SIGNAL_V
 labels: LIM
 type: thermal_cond
 tensor_order: 2
 coord_sys: global
 symmetry:
 distribution:
  component: 398

*thickness@SIGNAL_V
 labels:
 type: thickness
 tensor_order: -1
 list: zmin = 69800,
zmax = 70400

*min_wire_pitch@SIGNAL_H
 labels:
 type: min_wire_pitch
 tensor_order: 0
 symmetry:
 distribution:
  component: 40

*min_wire_width@SIGNAL_H
 labels:
 type: min_wire_width
 tensor_order: 0
 symmetry:
 distribution:
  component: 20

*stiffness@SIGNAL_H
 labels: LIEM
 type: stiffness
 tensor_order: 4
 list: E = 1.3e11, nu
= 0.34
 coord_sys: global
 symmetry:
 distribution:
  component: E*(1-
nu)/(1-nu-2*nu^2)
  component: E*nu/(1-
nu-2*nu^2)
  component: E*nu/(1-
nu-2*nu^2)
  component: ZERO
  component: ZERO
  component: ZERO
  component: E*(1-
nu)/(1-nu-2*nu^2)
  component: E*nu/(1-
nu-2*nu^2)
  component: ZERO
  component: ZERO
  component: ZERO
  component: E*(1-
nu)/(1-nu-2*nu^2)
  component: ZERO
  component: ZERO
  component: ZERO
  component:
E/(2*(1+nu))
  component: ZERO
  component: ZERO
  component:
E/(2*(1+nu))
  component: ZERO
  component:
E/(2*(1+nu))

*thermal_expand@SIGNAL_H
 labels: LIM
 type: thermal_expand
 tensor_order: 2
```

```
 coord_sys: global
 symmetry:
 distribution:
  component: 1.7e-5

*thermal_cond@SIGNAL_H
 labels: LIM
 type: thermal_cond
 tensor_order: 2
 coord_sys: global
 symmetry:
 distribution:
  component: 398

*thickness@SIGNAL_H
 labels:
 type: thickness
 tensor_order: -1
 list: zmin = 66200,
zmax = 66800

*elem_type
 labels: SolidTet10
 type: element
 tensor_order: -1

*initial_stress
 labels:
 type: initial_stress
 tensor_order: 2
 coord_sys: global
 symmetry:
 distribution:
  component: 10
  component: 10
  component: 10
  component: 0
  component: 0
  component: 0

*initial_temp
 labels:
 type: initial_temp
 tensor_order: 0
 symmetry:
 distribution:
  component: 20

*position_resolution
 labels:
 type: position_res
 tensor_order: 0
 symmetry:
 distribution:
  component: 1e-8

*temperature_resolution
 labels:
 type: temp_res
 tensor_order: 0
 symmetry:
 distribution:
  component: 0.1

*local_window
 labels:
 type: local_wind
 tensor_order: -1
 list: -3367100,
-382250, 58000, -
3353200, -366050, 69000
```

C.2 25chip.org

```
*Case: local_stress
   attribute:
elem_type
   group: ini-
tial_condition@ls
   group: tempera-
ture_distribution@ls
   group: bound-
ary_condition@ls
   group: mate-
rial_properties@ls
   group: prob-
lem_definition@ls

*Case: local_thermal
   group: accu-
racy_constants@lt
   group: bound-
ary_condition@lt
   group: mate-
rial_properties@lt
   group: physi-
cal_dimensions@lt
   group: prob-
lem_definition@lt

*Case: global_stress
   group: ini-
tial_condition@gs
   group: bound-
ary_condition@gs
   group: lay-
out_info@gs
   group: mate-
rial_properties@gs
   group: physi-
cal_dimensions@gs

*Case: global_thermal
   group: bound-
ary_condition@gt
   group: lay-
out_info@gt
   group: mate-
rial_properties@gt
   group: physi-
cal_dimensions@gt

*Group: ini-
tial_condition@ls
   attribute: ini-
tial_stress
   attribute: ini-
tial_temp

*Group: tempera-
ture_distribution@ls
   attribute: LT_temp

*Group: bound-
ary_condition@ls
   attribute: GS_disp

*Group: accu-
racy_constants@lt
   attribute: posi-
tion_resolution
   attribute: tempera-
ture_resolution

*Group: bound-
ary_condition@lt
   set: bound-
ary_condition@pins
   set: bound-
ary_condition@solder_bumps
   attribute: GT_temp
   set: bound-
ary_condition@DIELECTRIC_INS2
   set: bound-
ary_condition@DIELECTRIC_INS1
   set: bound-
ary_condition@GND_3
   set: bound-
ary_condition@GND_2
   set: bound-
ary_condition@GND_1
   set: bound-
ary_condition@VDD
   set: bound-
ary_condition@SUBSTRATE
   set: bound-
ary_condition@CHIP_25
   set: bound-
ary_condition@CHIP_24
   set: bound-
ary_condition@CHIP_23
   set: bound-
ary_condition@CHIP_22
   set: bound-
ary_condition@CHIP_21
   set: bound-
ary_condition@CHIP_20
   set: bound-
ary_condition@CHIP_19
   set: bound-
ary_condition@CHIP_18
   set: bound-
ary_condition@CHIP_17
   set: bound-
ary_condition@CHIP_16
   set: bound-
ary_condition@CHIP_15
   set: bound-
ary_condition@CHIP_14
   set: bound-
ary_condition@CHIP_13
   set: bound-
ary_condition@CHIP_12
   set: bound-
ary_condition@CHIP_11
   set: bound-
ary_condition@CHIP_10
   set: bound-
ary_condition@CHIP_9
   set: bound-
ary_condition@CHIP_8
   set: bound-
ary_condition@CHIP_7
   set: bound-
ary_condition@CHIP_6
   set: bound-
ary_condition@CHIP_5
   set: bound-
ary_condition@CHIP_4
   set: bound-
ary_condition@CHIP_3
   set: bound-
ary_condition@CHIP_2
   set: bound-
ary_condition@CHIP_1
   set: bound-
ary_condition@VIA_GROUND
   set: bound-
ary_condition@VIA_POWER
   set: bound-
ary_condition@VIA_CONTACT
   set: bound-
ary_condition@SIGNAL_V
   set: bound-
ary_condition@SIGNAL_H

*Group: prob-
lem_definition@lt
   attribute: lo-
cal_window
         multiplier:
sFactor

*Group: ini-
tial_condition@gs
   attribute: ini-
tial_stress
   attribute: ini-
tial_temp

*Group: layout_info@gs
   set: lay-
out_info@pins
   set: lay-
out_info@solder_bumps
   attribute: ma-
jorDir@layer11
   attribute: ma-
jorDir@layer10
   attribute: ma-
jorDir@layer9
   attribute: ma-
jorDir@layer8
   attribute: ma-
jorDir@layer7
   attribute: ma-
jorDir@layer6
   attribute: ma-
jorDir@layer5
   attribute: ma-
jorDir@layer4
   attribute: ma-
jorDir@layer3
   attribute: ma-
jorDir@layer2
   attribute: ma-
jorDir@layer1
   attribute: numOf-
PhyLayers
   attribute: num-
berOfChips
   set: lay-
out_info@DIELECTRIC_INS2
   set: lay-
out_info@DIELECTRIC_INS1
   set: lay-
out_info@GND_3
   set: lay-
out_info@GND_2
   set: lay-
out_info@GND_1
   set: lay-
out_info@VDD
   attribute: off-
set@gs
   set: lay-
out_info@SUBSTRATE
   set: lay-
out_info@CHIP_25
   set: lay-
out_info@CHIP_24
```

```
   set: lay-
out_info@CHIP_23
   set: lay-
out_info@CHIP_22
   set: lay-
out_info@CHIP_21
   set: lay-
out_info@CHIP_20
   set: lay-
out_info@CHIP_19
   set: lay-
out_info@CHIP_18
   set: lay-
out_info@CHIP_17
   set: lay-
out_info@CHIP_16
   set: lay-
out_info@CHIP_15
   set: lay-
out_info@CHIP_14
   set: lay-
out_info@CHIP_13
   set: lay-
out_info@CHIP_12
   set: lay-
out_info@CHIP_11
   set: lay-
out_info@CHIP_10
   set: lay-
out_info@CHIP_9
   set: lay-
out_info@CHIP_8
   set: lay-
out_info@CHIP_7
   set: lay-
out_info@CHIP_6
   set: lay-
out_info@CHIP_5
   set: lay-
out_info@CHIP_4
   set: lay-
out_info@CHIP_3
   set: lay-
out_info@CHIP_2
   set: lay-
out_info@CHIP_1
   set: lay-
out_info@VIA_GROUND
   set: lay-
out_info@VIA_POWER
   set: lay-
out_info@VIA_CONTACT
   set: lay-
out_info@SIGNAL_V
   set: lay-
out_info@SIGNAL_H

*Group: physi-
cal_dimensions@gs
   set: physi-
cal_dimensions@pins
   set: physi-
cal_dimensions@solder_bumps
   attribute: geom-
Info@layer11
   attribute: geom-
Info@layer10
   attribute: geom-
Info@layer9
   attribute: geom-
Info@layer8
   attribute: geom-
Info@layer7
   attribute: geom-
Info@layer6
   attribute: geom-
Info@layer5
   attribute: geom-
Info@layer4
   attribute: geom-
Info@layer3
   attribute: geom-
Info@layer2
   attribute: geom-
Info@layer1
   attribute: thick-
ness@DIELECTRIC_INS2YYY6
         multiplier:
sFactor
   attribute: thick-
ness@DIELECTRIC_INS2YYY5
         multiplier:
sFactor
   attribute: thick-
ness@DIELECTRIC_INS2YYY4
         multiplier:
sFactor
   attribute: thick-
ness@DIELECTRIC_INS1YYY3
         multiplier:
sFactor
   attribute: thick-
ness@DIELECTRIC_INS1YYY2
         multiplier:
sFactor
   attribute: thick-
ness@DIELECTRIC_INS1YYY1
         multiplier:
sFactor
   set: physi-
cal_dimensions@DIELECTRIC_INS2
   set: physi-
cal_dimensions@DIELECTRIC_INS1
   set: physi-
cal_dimensions@GND_3
   set: physi-
cal_dimensions@GND_2
   set: physi-
cal_dimensions@GND_1
   set: physi-
cal_dimensions@VDD
   attribute:
layer_width
   attribute:
layer_length
   set: physi-
cal_dimensions@SUBSTRATE
   set: physi-
cal_dimensions@CHIP_25
   set: physi-
cal_dimensions@CHIP_24
   set: physi-
cal_dimensions@CHIP_23
   set: physi-
cal_dimensions@CHIP_22
   set: physi-
cal_dimensions@CHIP_21
   set: physi-
cal_dimensions@CHIP_20
   set: physi-
cal_dimensions@CHIP_19
   set: physi-
cal_dimensions@CHIP_18
   set: physi-
cal_dimensions@CHIP_17
   set: physi-
cal_dimensions@CHIP_16
   set: physi-
cal_dimensions@CHIP_15
   set: physi-
cal_dimensions@CHIP_14
   set: physi-
cal_dimensions@CHIP_13
   set: physi-
cal_dimensions@CHIP_12
   set: physi-
cal_dimensions@CHIP_11
   set: physi-
cal_dimensions@CHIP_10
   set: physi-
cal_dimensions@CHIP_9
   set: physi-
cal_dimensions@CHIP_8
   set: physi-
cal_dimensions@CHIP_7
   set: physi-
cal_dimensions@CHIP_6
   set: physi-
cal_dimensions@CHIP_5
   set: physi-
cal_dimensions@CHIP_4
   set: physi-
cal_dimensions@CHIP_3
   set: physi-
cal_dimensions@CHIP_2
   set: physi-
cal_dimensions@CHIP_1
   set: physi-
cal_dimensions@VIA_GROUND
   set: physi-
cal_dimensions@VIA_POWER
   set: physi-
cal_dimensions@VIA_CONTACT
   set: physi-
cal_dimensions@SIGNAL_V
   set: physi-
cal_dimensions@SIGNAL_H

*Group: layout_info@gt
   set: lay-
out_info@pins
   set: lay-
out_info@solder_bumps
   attribute: ma-
jorDir@layer11
   attribute: ma-
jorDir@layer10
   attribute: ma-
jorDir@layer9
   attribute: ma-
jorDir@layer8
   attribute: ma-
jorDir@layer7
   attribute: ma-
jorDir@layer6
   attribute: ma-
jorDir@layer5
   attribute: ma-
jorDir@layer4
   attribute: ma-
jorDir@layer3
   attribute: ma-
jorDir@layer2
   attribute: ma-
jorDir@layer1
   attribute: numOf-
PhyLayers
   attribute: num-
berOfChips
   set: lay-
out_info@DIELECTRIC_INS2
   set: lay-
out_info@DIELECTRIC_INS1
   set: lay-
out_info@GND_3
   set: lay-
out_info@GND_2
```

```
    set: lay-
out_info@GND_1
    set: lay-
out_info@VDD
    attribute: off-
set@gt
    set: lay-
out_info@SUBSTRATE
    set: lay-
out_info@CHIP_25
    set: lay-
out_info@CHIP_24
    set: lay-
out_info@CHIP_23
    set: lay-
out_info@CHIP_22
    set: lay-
out_info@CHIP_21
    set: lay-
out_info@CHIP_20
    set: lay-
out_info@CHIP_19
    set: lay-
out_info@CHIP_18
    set: lay-
out_info@CHIP_17
    set: lay-
out_info@CHIP_16
    set: lay-
out_info@CHIP_15
    set: lay-
out_info@CHIP_14
    set: lay-
out_info@CHIP_13
    set: lay-
out_info@CHIP_12
    set: lay-
out_info@CHIP_11
    set: lay-
out_info@CHIP_10
    set: lay-
out_info@CHIP_9
    set: lay-
out_info@CHIP_8
    set: lay-
out_info@CHIP_7
    set: lay-
out_info@CHIP_6
    set: lay-
out_info@CHIP_5
    set: lay-
out_info@CHIP_4
    set: lay-
out_info@CHIP_3
    set: lay-
out_info@CHIP_2
    set: lay-
out_info@CHIP_1
    set: lay-
out_info@VIA_GROUND
    set: lay-
out_info@VIA_POWER
    set: lay-
out_info@VIA_CONTACT
    set: lay-
out_info@SIGNAL_V
    set: lay-
out_info@SIGNAL_H

*Set: bound-
ary_condition@SUBSTRATE
    attribute:
heat_transfer@SUBSTRATE

*Set: bound-
ary_condition@CHIP_2
    attribute:
heat_transfer@CHIP_2
    attribute:
power_density@CHIP_2

*Set: bound-
ary_condition@CHIP_1
    attribute:
heat_transfer@CHIP_1
    attribute:
power_density@CHIP_1

*Set: mate-
rial_properties@pins
    attribute: stiff-
ness@pins

*Set: mate-
rial_properties@solder_bumps
    attribute: stiff-
ness@solder_bumps
    attribute: ther-
mal_expand@solder_bumps
    attribute: ther-
mal_cond@solder_bumps

*Set: mate-
rial_properties@DIELECTRIC_INS1
    attribute: stiff-
ness@DIELECTRIC_INS1
    attribute: ther-
mal_expand@DIELECTRIC_INS1
    attribute: ther-
mal_cond@DIELECTRIC_INS1

*Set: mate-
rial_properties@GND_1
    attribute: stiff-
ness@GND_1
    attribute: ther-
mal_expand@GND_1
    attribute: ther-
mal_cond@GND_1

*Set: mate-
rial_properties@VDD
    attribute: stiff-
ness@VDD
    attribute: ther-
mal_expand@VDD
    attribute: ther-
mal_cond@VDD

*Set: mate-
rial_properties@SUBSTRATE
    attribute: stiff-
ness@SUBSTRATE
    attribute: ther-
mal_expand@SUBSTRATE
    attribute: ther-
mal_cond@SUBSTRATE

*Set: mate-
rial_properties@CHIP_2
    attribute: stiff-
ness@CHIP_2
    attribute: ther-
mal_expand@CHIP_2
    attribute: ther-
mal_cond@CHIP_2

*Set: mate-
rial_properties@CHIP_1
    attribute: stiff-
ness@CHIP_1
    attribute: ther-
mal_expand@CHIP_1
    attribute: ther-
mal_cond@CHIP_1

*Set: mate-
rial_properties@VIA_GROUND
    attribute: stiff-
ness@VIA_GROUND
    attribute: ther-
mal_expand@VIA_GROUND
    attribute: ther-
mal_cond@VIA_GROUND

*Set: mate-
rial_properties@SIGNAL_V
    attribute: stiff-
ness@SIGNAL_V
    attribute: ther-
mal_expand@SIGNAL_V
    attribute: ther-
mal_cond@SIGNAL_V

*Set: mate-
rial_properties@SIGNAL_H
    attribute: stiff-
ness@SIGNAL_H
    attribute: ther-
mal_expand@SIGNAL_H
    attribute: ther-
mal_cond@SIGNAL_H

*Set: layout_info@pins
    attribute:
npins@pins

*Set: lay-
out_info@CHIP_1
    attribute: cen-
ter@CHIP_1
    attribute:
nsold_bmp@CHIP_1
    attribute:
pwr_via_area@CHIP_1
          multiplier:
sFactor
          multiplier:
sFactor
    attribute:
sig_via_area@CHIP_1
          multiplier:
sFactor
          multiplier:
sFactor
    attribute:
gnd_via_area@CHIP_1
          multiplier:
sFactor
          multiplier:
sFactor
    attribute:
therm_via_area@CHIP_1
          multiplier:
sFactor
          multiplier:
sFactor
    attribute:
nth_vias@CHIP_1

*Set: lay-
out_info@SIGNAL_V
    attribute:
min_wire_pitch@SIGNAL_V
          multiplier:
sFactor
```

```
   attribute:
min_wire_width@SIGNAL_V
       multiplier:
sFactor

*Set: lay-
out_info@SIGNAL_H
   attribute:
min_wire_pitch@SIGNAL_H
       multiplier:
sFactor
   attribute:
min_wire_width@SIGNAL_H
       multiplier:
sFactor

*Set: physi-
cal_dimensions@pins
   attribute:
pin_size@pins
       multiplier:
sFactor

*Set: physi-
cal_dimensions@solder_bumps
   attribute:
sol_diam@solder_bumps
       multiplier:
sFactor
   attribute:
sol_height@solder_bumps
       multiplier:
sFactor

*Set: physi-
cal_dimensions@DIELECTRIC_INS2
   attribute: thick-
ness@DIELECTRIC_INS2
       multiplier:
sFactor

*Set: physi-
cal_dimensions@DIELECTRIC_INS1
   attribute: thick-
ness@DIELECTRIC_INS1
       multiplier:
sFactor

*Set: physi-
cal_dimensions@GND_1
   attribute: thick-
ness@GND_1
       multiplier:
sFactor

*Set: physi-
cal_dimensions@VDD
   attribute: thick-
ness@VDD
       multiplier:
sFactor

*Set: physi-
cal_dimensions@SUBSTRATE
   attribute: thick-
ness@SUBSTRATE
       multiplier:
sFactor

*Set: physi-
cal_dimensions@CHIP_1
   attribute:
chip_length@CHIP_1
   attribute:
chip_width@CHIP_1
   attribute: thick-
ness@CHIP_1
       multiplier:
sFactor

*Set: physi-
cal_dimensions@VIA_GROUND
   attribute: thick-
ness@VIA_GROUND
       multiplier:
sFactor

*Set: physi-
cal_dimensions@VIA_POWER
   attribute: thick-
ness@VIA_POWER
       multiplier:
sFactor

*Set: physi-
cal_dimensions@VIA_CONTACT
   attribute: thick-
ness@VIA_CONTACT
       multiplier:
sFactor

*Set: physi-
cal_dimensions@SIGNAL_V
   attribute: thick-
ness@SIGNAL_V
       multiplier:
sFactor

*Set: physi-
cal_dimensions@SIGNAL_H
   attribute: thick-
ness@SIGNAL_H
       multiplier:
sFactor
```

C.3 25chip.rel

```
*model_name = PhysicalLayers

*attribute = npins@pins
    model_entity_number: 0
    model_entity_string: layer0
    model_entity_type: 3

*model_name = PhysicalLayers

*attribute = stiffness@pins
    model_entity_number: 0
    model_entity_string: layer0
    model_entity_type: 3

*model_name = PhysicalLayers

*attribute = pin_size@pins
    model_entity_number: 0
    model_entity_string: layer0
    model_entity_type: 3

*model_name = PhysicalLayers

*attribute = stiffness@solder_bumps
    model_entity_number: 0
    model_entity_string: layer0
    model_entity_type: 3

*model_name = PhysicalLayers

*attribute = ther-
mal_expand@solder_bumps
    model_entity_number: 0
    model_entity_string: layer0
    model_entity_type: 3

*model_name = PhysicalLayers

*attribute = ther-
mal_cond@solder_bumps
    model_entity_number: 0
    model_entity_string: layer0
    model_entity_type: 3

*model_name = PhysicalLayers

*attribute = sol_diam@solder_bumps
    model_entity_number: 0
    model_entity_string: layer0
    model_entity_type: 3


*model_name = PhysicalLayers

*attribute = GT_amp
```

```
    model_entity_number: 0
    model_entity_string: layer2
    model_entity_type: 3

*model_name = PhysicalLayers

*attribute = GT_amp
    model_entity_number: 0
    model_entity_string: layer1
    model_entity_type: 3


*model_name = PhysicalLayers

*attribute = power_density@CHIP_2
    model_entity_number: 0
    model_entity_string: chip0
    model_entity_type: 3

*model_name = PhysicalLayers

*attribute = stiffness@CHIP_2
    model_entity_number: 0
    model_entity_string: chip3
    model_entity_type: 3

*model_name = CIFlayer

*attribute = stiffness@CHIP_2
    model_entity_number: 0
    model_entity_string: CHIP_2
    model_entity_type: 3

*model_name = PhysicalLayers

*attribute = thermal_expand@CHIP_2
    model_entity_number: 0
    model_entity_string: chip3
    model_entity_type: 3

*model_name = CIFlayer

*attribute = thermal_expand@CHIP_2
    model_entity_number: 0
    model_entity_string: CHIP_2
    model_entity_type: 3

*model_name = PhysicalLayers

*attribute = thermal_cond@CHIP_2
    model_entity_number: 0
    model_entity_string: chip3
    model_entity_type: 3

*model_name = CIFlayer

*attribute = thermal_cond@CHIP_2
    model_entity_number: 0
    model_entity_string: CHIP_2
    model_entity_type: 3

*model_name = PhysicalLayers

*attribute = thickness@CHIP_2
    model_entity_number: 0
    model_entity_string: chip3
    model_entity_type: 3

*model_name = CIFlayer

*attribute = thickness@CHIP_2
    model_entity_number: 0
    model_entity_string: CHIP_2
    model_entity_type: 3

*model_name = PhysicalLayers

*attribute = center@CHIP_1
    model_entity_number: 0
    model_entity_string: chip7
    model_entity_type: 3

*model_name = CIFlayer

*attribute = center@CHIP_1
    model_entity_number: 0
    model_entity_string: CHIP_1
    model_entity_type: 3

*model_name = PhysicalLayers

*attribute = pwr_via_area@CHIP_1
    model_entity_number: 0
    model_entity_string: chip7
    model_entity_type: 3

*model_name = CIFlayer

*attribute = pwr_via_area@CHIP_1
    model_entity_number: 0
    model_entity_string: CHIP_1
    model_entity_type: 3

*model_name = PhysicalLayers

*attribute = pwr_via_area@CHIP_1
    model_entity_number: 0
    model_entity_string: chip0
    model_entity_type: 3

*model_name = PhysicalLayers

*attribute = sig_via_area@CHIP_1
    model_entity_number: 0
    model_entity_string: chip7
    model_entity_type: 3

*model_name = CIFlayer

*attribute = therm_via_area@CHIP_1
    model_entity_number: 0
    model_entity_string: CHIP_1
    model_entity_type: 3

*model_name = PhysicalLayers

*attribute = therm_via_area@CHIP_1
    model_entity_number: 0
    model_entity_string: chip0
    model_entity_type: 3

*model_name = PhysicalLayers

*attribute = nth_vias@CHIP_1
    model_entity_number: 0
    model_entity_string: chip7
    model_entity_type: 3

*model_name = CIFlayer

*attribute = nth_vias@CHIP_1
    model_entity_number: 0
    model_entity_string: CHIP_1
    model_entity_type: 3

*model_name = PhysicalLayers

*attribute = nth_vias@CHIP_1
    model_entity_number: 0
    model_entity_string: chip0
    model_entity_type: 3

*model_name = PhysicalLayers

*attribute = heat_transfer@CHIP_1
    model_entity_number: 0
    model_entity_string: chip7
    model_entity_type: 3

*model_name = CIFlayer

*attribute = heat_transfer@CHIP_1
    model_entity_number: 0
    model_entity_string: CHIP_1
    model_entity_type: 3

*model_name = PhysicalLayers

*attribute = heat_transfer@CHIP_1
    model_entity_number: 0
    model_entity_string: chip0
    model_entity_type: 3

*model_name = CIFlayer

*attribute = thermal_expand@CHIP_1
    model_entity_number: 0
    model_entity_string: CHIP_1
    model_entity_type: 3

*model_name = CIFlayer

*attribute = thickness@CHIP_1
    model_entity_number: 0
    model_entity_string: CHIP_1
    model_entity_type: 3

*model_name = 25chip

*attribute = initial_stress
    model_entity_number: 3394
    model_entity_type: 3

*model_name = 25chip

*attribute = initial_stress
    model_entity_number: 2643
    model_entity_type: 3

*model_name = 25chip

*attribute = initial_stress
    model_entity_number: 1973
    model_entity_type: 3

*model_name = 25chip

*attribute = initial_stress
    model_entity_number: 1384
    model_entity_type: 3

*model_name = PhysicalLayers

*attribute = initial_stress
    model_entity_number: 0
    model_entity_string: layer0
    model_entity_type: 3

*model_name = 25chip

*attribute = initial_stress
    model_entity_number: 19
    model_entity_string: layer0
    model_entity_type: 3

*model_name = 25chip

*attribute = initial_temp
    model_entity_number: 20706
    model_entity_type: 3

*model_name = 25chip

*attribute = initial_temp
    model_entity_number: 20065
    model_entity_type: 3
```

Appendix D Example REPAS Attribute File

This appendix presents a sample attribute file required as one of the three input files for the REPAS preprocessing. The file specification format is specified in Section 6.3. The sample file presented below is taken out of a run for a 25 chip MCM design and is presented in two columns.

```
*pins
  material_properties:
 type: LIEM
 stiffness: E = 1.3e11, nu = 0.34
  layout_info:
 npins: 168
  physical_dimensions:
        pin_size: 500000, 40000,
40000

*solder_bumps
  physical_dimensions:
 sol_height: 8500
 sol_diam: 14000

  material_properties:
 type: LIM
 thermal_cond: 36
 thermal_expand: 1.0e-5
 type: LIEM
 stiffness: E = 3e10, nu = 0.2

*chip_1
  physical_dimensions:
        thickness: zmin = 72220,
zmax = 79720
  material_properties:
 type: LIM
 thermal_cond: 58
 thermal_expand: 5.7e-6
 type: LIEM
 stiffness: E=8.47e10, nu = 0.22
  boundary_condition:
 power_density: 7e4
 heat_trans: 1, 10, 70
  layout_info:
 nsold_bmp: 0

*chip_2
  physical_dimensions:
        thickness: zmin = 72220,
zmax = 79720
  material_properties:
 type: LIM
 thermal_cond: 58
 thermal_expand: 5.7e-6
 type: LIEM
 stiffness: E=8.47e10, nu = 0.22
  boundary_condition:
 power_density: 7e4
 heat_trans: 1, 10, 70
  layout_info:
 nsold_bmp: 0

*chip_3
  physical_dimensions:
        thickness: zmin = 72220,
zmax = 79720
  material_properties:
 type: LIM
 thermal_cond: 58
 thermal_expand: 5.7e-6
 type: LIEM
 stiffness: E=8.47e10, nu = 0.22
  boundary_condition:
 power_density: 7e4
 heat_trans: 1, 10, 70
  layout_info:
 nsold_bmp: 0

*chip_4
  physical_dimensions:
        thickness: zmin = 72220,
zmax = 79720
  material_properties:
 type: LIM
 thermal_cond: 58
 thermal_expand: 5.7e-6
 type: LIEM
 stiffness: E=8.47e10, nu = 0.22
  boundary_condition:
 power_density: 7e4
 heat_trans: 1, 10, 70
  layout_info:
 nsold_bmp: 0

*chip_5
  physical_dimensions:
        thickness: zmin = 72220,
zmax = 79720
  material_properties:
 type: LIM
 thermal_cond: 58
 thermal_expand: 5.7e-6
 type: LIEM
 stiffness: E=8.47e10, nu = 0.22
  boundary_condition:
 power_density: 7e4
 heat_trans: 1, 10, 70
  layout_info:
```

```
 nsold_bmp: 0

*chip_6
  physical_dimensions:
        thickness: zmin = 72220,
zmax = 79720
  material_properties:
 type: LIM
 thermal_cond: 58
 thermal_expand: 5.7e-6
 type: LIEM
 stiffness: E=8.47e10, nu = 0.22
  boundary_condition:
 power_density: 7e4
 heat_trans: 1, 10, 70
  layout_info:
 nsold_bmp: 0

*chip_7
  physical_dimensions:
        thickness: zmin = 72220,
zmax = 79720
  material_properties:
 type: LIM
 thermal_cond: 58
 thermal_expand: 5.7e-6
 type: LIEM
 stiffness: E=8.47e10, nu = 0.22
  boundary_condition:
 power_density: 7e4
 heat_trans: 1, 10, 70
  layout_info:
 nsold_bmp: 0

*chip_8
  physical_dimensions:
        thickness: zmin = 72220,
zmax = 79720
  material_properties:
 type: LIM
 thermal_cond: 58
 thermal_expand: 5.7e-6
 type: LIEM
 stiffness: E=8.47e10, nu = 0.22
  boundary_condition:
 power_density: 7e4
 heat_trans: 1, 10, 70
  layout_info:
 nsold_bmp: 0

*chip_9
  physical_dimensions:
        thickness: zmin = 72220,
zmax = 79720
  material_properties:
 type: LIM
 thermal_cond: 58
 thermal_expand: 5.7e-6
 type: LIEM
 stiffness: E=8.47e10, nu = 0.22
  boundary_condition:
 power_density: 8e4
 heat_trans: 1, 10, 70
  layout_info:
 nsold_bmp: 0

*chip_10
  physical_dimensions:
        thickness: zmin = 72220,
zmax = 79720
  material_properties:
 type: LIM
 thermal_cond: 58
 thermal_expand: 5.7e-6
 type: LIEM
 stiffness: E=8.47e10, nu = 0.22
  boundary_condition:
 power_density: 8e4
 heat_trans: 1, 10, 70
  layout_info:
 nsold_bmp: 0

*chip_11
  physical_dimensions:
        thickness: zmin = 72220,
zmax = 79720
  material_properties:
 type: LIM
 thermal_cond: 58
 thermal_expand: 5.7e-6
 type: LIEM
 stiffness: E=8.47e10, nu = 0.22
  boundary_condition:
 power_density: 7e4
 heat_trans: 1, 10, 70
  layout_info:
 nsold_bmp: 0

*chip_12
  physical_dimensions:
        thickness: zmin = 72220,
zmax = 79720
  material_properties:
 type: LIM
 thermal_cond: 58
 thermal_expand: 5.7e-6
 type: LIEM
 stiffness: E=8.47e10, nu = 0.22
  boundary_condition:
 power_density: 7e4
 heat_trans: 1, 10, 70
  layout_info:
 nsold_bmp: 0

*chip_13
  physical_dimensions:
        thickness: zmin = 72220,
zmax = 79720
  material_properties:
 type: LIM
 thermal_cond: 58
 thermal_expand: 5.7e-6
 type: LIEM
 stiffness: E=8.47e10, nu = 0.22
  boundary_condition:
 power_density: 7e4
 heat_trans: 1, 10, 70
  layout_info:
 nsold_bmp: 0

*chip_14
  physical_dimensions:
        thickness: zmin = 72220,
zmax = 79720
```

```
 material_properties:
type: LIM
thermal_cond: 58
thermal_expand: 5.7e-6
type: LIEM
stiffness: E=8.47e10, nu = 0.22
 boundary_condition:
power_density: 7e4
heat_trans: 1, 10, 70
 layout_info:
nsold_bmp: 0

*chip_15
  physical_dimensions:
         thickness: zmin = 72220,
zmax = 79720
  material_properties:
 type: LIM
 thermal_cond: 58
 thermal_expand: 5.7e-6
 type: LIEM
 stiffness: E=8.47e10, nu = 0.22
  boundary_condition:
 power_density: 7e4
 heat_trans: 1, 10, 70
  layout_info:
 nsold_bmp: 0

*chip_16
  physical_dimensions:
         thickness: zmin = 72220,
zmax = 79720
  material_properties:
 type: LIM
 thermal_cond: 58
 thermal_expand: 5.7e-6
 type: LIEM
 stiffness: E=8.47e10, nu = 0.22
  boundary_condition:
 power_density: 7e4
 heat_trans: 1, 10, 70
  layout_info:
 nsold_bmp: 0

*chip_17
  physical_dimensions:
         thickness: zmin = 72220,
zmax = 79720
  material_properties:
 type: LIM
 thermal_cond: 58
 thermal_expand: 5.7e-6
 type: LIEM
 stiffness: E=8.47e10, nu = 0.22
  boundary_condition:
 power_density: 7e4
 heat_trans: 1, 10, 70
  layout_info:
 nsold_bmp: 0

*chip_18
  physical_dimensions:
         thickness: zmin = 72220,
zmax = 79720
  material_properties:
 type: LIM
 thermal_cond: 58
 thermal_expand: 5.7e-6
 type: LIEM
 stiffness: E=8.47e10, nu = 0.22
  boundary_condition:
 power_density: 7e4
 heat_trans: 1, 10, 70
  layout_info:
 nsold_bmp: 0

*chip_19
  physical_dimensions:
         thickness: zmin = 72220,
zmax = 79720
  material_properties:
 type: LIM
 thermal_cond: 58
 thermal_expand: 5.7e-6
 type: LIEM
 stiffness: E=8.47e10, nu = 0.22
  boundary_condition:
 power_density: 7e4
 heat_trans: 1, 10, 70
  layout_info:
 nsold_bmp: 0

*chip_20
  physical_dimensions:
         thickness: zmin = 72220,
zmax = 79720
  material_properties:
 type: LIM
 thermal_cond: 58
 thermal_expand: 5.7e-6
 type: LIEM
 stiffness: E=8.47e10, nu = 0.22
  boundary_condition:
 power_density: 11.5e4
 heat_trans: 1, 10, 70
  layout_info:
 nsold_bmp: 0

*chip_21
  physical_dimensions:
         thickness: zmin = 72220,
zmax = 79720
  material_properties:
 type: LIM
 thermal_cond: 58
 thermal_expand: 5.7e-6
 type: LIEM
 stiffness: E=8.47e10, nu = 0.22
  boundary_condition:
 power_density: 11e4
 heat_trans: 1, 10, 70
  layout_info:
 nsold_bmp: 0

*chip_22
  physical_dimensions:
         thickness: zmin = 72220,
zmax = 79720
  material_properties:
 type: LIM
 thermal_cond: 58
 thermal_expand: 5.7e-6
 type: LIEM
 stiffness: E=8.47e10, nu = 0.22
 boundary_condition:
power_density: 11e4
heat_trans: 1, 10, 70
 layout_info:
nsold_bmp: 0

*chip_23
 physical_dimensions:
        thickness: zmin = 72220,
zmax = 79720
 material_properties:
type: LIM
thermal_cond: 58
thermal_expand: 5.7e-6
type: LIEM
stiffness: E=8.47e10, nu = 0.22
 boundary_condition:
power_density: 11e4
heat_trans: 1, 10, 70
 layout_info:
nsold_bmp: 0

*chip_24
 physical_dimensions:
        thickness: zmin = 72220,
zmax = 79720
 material_properties:
type: LIM
thermal_cond: 58
thermal_expand: 5.7e-6
type: LIEM
stiffness: E=8.47e10, nu = 0.22
 boundary_condition:
power_density: 11e4
heat_trans: 1, 10, 70
 layout_info:
nsold_bmp: 0

*chip_25
 physical_dimensions:
        thickness: zmin = 72220,
zmax = 79720
 material_properties:
type: LIM
thermal_cond: 58
thermal_expand: 5.7e-6
type: LIEM
stiffness: E=8.47e10, nu = 0.22
 boundary_condition:
power_density: 5e4
heat_trans: 1, 10, 70
 layout_info:
nsold_bmp: 0

*dielectric_ins2
 physical_dimensions:
        thickness: zmin = 68600,
zmax = 71600
 material_properties:
type: LIM
thermal_cond: 0.2
thermal_expand: 3.5e-5
type: LIEM
stiffness: E = 2.5e9, nu = 0.45

*dielectric_ins1
 physical_dimensions:
        thickness: zmin = 65000,
zmax = 68000
 material_properties:
type: LIM
thermal_cond: 0.2
thermal_expand: 3.5e-5
type: LIEM
stiffness: E = 2.5e9, nu = 0.45

*gnd_1
 physical_dimensions:
        thickness: zmin = 60000,
zmax = 62500
 material_properties:
type: LIM
thermal_cond: 398
thermal_expand: 1.7e-5
type: LIEM
stiffness: E = 1.3e11, nu = 0.34

*gnd_2
 physical_dimensions:
        thickness: zmin = 68000,
zmax = 68600
 material_properties:
type: LIM
thermal_cond: 398
thermal_expand: 1.7e-5
type: LIEM
stiffness: E = 1.3e11, nu = 0.34

*gnd_3
 physical_dimensions:
        thickness: zmin = 71600,
zmax = 72200
 material_properties:
type: LIM
thermal_cond: 398
thermal_expand: 1.7e-5
type: LIEM
stiffness: E = 1.3e11, nu = 0.34
 boundary_condition:
heat_trans: 1, 10, 70

*vdd
 physical_dimensions:
        thickness: zmin = 62500,
zmax = 65000
 material_properties:
type: LIM
thermal_cond: 398
thermal_expand: 1.7e-5
type: LIEM
stiffness: E = 1.3e11, nu = 0.34

*substrate
 physical_dimensions:
        thickness: zmin = 0, zmax =
60000
 material_properties:
type: LIM
thermal_cond: 84
thermal_expand: 3.14e-6
type: LIEM
stiffness: E=1.92e11, nu = 0.22
 boundary_condition:
heat_trans: 1, 1300, 25
```

```
*via_ground
  physical_dimensions:
        thickness: zmin = 62500,
zmax = 72200
  material_properties:
 type: LIM
 thermal_cond: 398
 thermal_expand: 1.7e-5
 type: LIEM
 stiffness: E = 1.3e11, nu = 0.34

*via_power
  physical_dimensions:
        thickness: zmin = 65000,
zmax = 72200
  material_properties:
 type: LIM
 thermal_cond: 398
 thermal_expand: 1.7e-5
 type: LIEM
 stiffness: E = 1.3e11, nu = 0.34

*via_contact
  physical_dimensions:
        thickness: zmin = 66800,
zmax = 69800
  material_properties:
 type: LIM
 thermal_cond: 398
 thermal_expand: 1.7e-5
 type: LIEM
 stiffness: E = 1.3e11, nu = 0.34

*signal_v
  physical_dimensions:
        thickness: zmin = 69800,
zmax = 70400
  material_properties:
 type: LIM
 thermal_cond: 398
 thermal_expand: 1.7e-5
 type: LIEM
 stiffness: E = 1.3e11, nu = 0.34
  layout_info:
 min_wire_width: 20
 min_wire_pitch: 40

*signal_h
  physical_dimensions:
        thickness: zmin = 66200,
zmax = 66800
  material_properties:
 type: LIM
 thermal_cond: 398
 thermal_expand: 1.7e-5
 type: LIEM
 stiffness: E = 1.3e11, nu = 0.34
  layout_info:
 min_wire_width: 20
 min_wire_pitch: 40
```

Bibliography

[1] H. Anton. *Elementary Linear Algebra.* John Wiley & Sons, 4th edition, 1984.

[2] P. L. Baehmann, T. L. Sham, L. Y. Song, and M. S. Shephard. Thermal and thermo-mechanical analysis of multichip modules using adaptive finite element techniques. In D. Agonafer and R. L. Fulton, editors, *Computer Aided Design in Electronic Packaging*, volume EEP-3, pages 57–63, New York, NY, 1992. ASME.

[3] M. Beall. Scorec mesh database user's guide, version 2.2 - draft. Technical Report SCOREC Report # 26-1993, Rensselaer Polytechnic Institute, Troy, NY 12180-3590, January 1994.

[4] M. Beall, A. Garg, R. Garimella, R. Iverson, Y. L. Coz, B. Lwo, T. Sham, M. Shephard, L. Song, and V. Wong. *REPAS — Rensselaer Electronic Packaging Software (User's Manual).* Rensselaer Polytechnic Institute, Troy, NY 12180–3590, 1994.

[5] M. Beall, A. Garg, R. Garimella, R. Iverson, Y. L. Coz, B. Lwo, T. Sham, M. Shephard, L. Song, and V. Wong. *REPAS — Rensselaer Electronic Packaging Software (Programmer's Manual).* Rensselaer Polytechnic Institute, Troy, NY 12180–3590, 1994.

[6] I. Beju, E. Soos, and P. P. Teodorescu. *Euclidean Tensor Calculus with Applications.* Abacus Press, 1983.

[7] L. Berardinis. The packaging hurdle. *Machine Design*, 1991.

[8] H. D. Block. *Introduciton to Tensor Analysis.* Charles E. Merrill Books Inc., 1962.

[9] S. Johnson. *YACC — Yet Another Compiler-Compiler.* Bell Laboratories, Murray Hill, New Jersey.

[10] B. Kernighan and R.Pike. *The UNIX Programming Environment.* Prentice-Hall, Inc., Murray Hill, New Jersey, 1984.

[11] Y. L. Le Coz and R. B. Iverson. A stochastic algorithm for high speed capacitance extraction in integrated circuits. *Solid State Electronics*, 35:1005–1012, 1992.

[12] J. McDonald, H. Greub, R. Steinvorth, B. Donlan, and A. Bergendahl. Wafer scale interconnects for gaaa packaging — applications to risc architecture. *Computer Magazine*, 1987.

[13] C. Mead and L. Conway. *Introduction to VLSI Systems.* Addison-Wesley Publ. Co., Reading, MA, 1980.

[14] L. Schaper. Design of multichip modules. *IEEE*, Vol. 80, No. 12, 1992.

[15] T. Sham, H. Tiersten, P. Baehmann, L. Song, Y. Zhou, B. Lwo, Y. L. Coz, and M. Shephard. A global-local procedure for the heat conduction analysis of multichip modules. In P. A. Engel and W. T. Chen, editors, *Advances in Electronic Packaging 1993*, volume 2, pages 551–562, New York, NY, 1993. ASME.

[16] M. Shephard and P. Finnigan. Towards automatic model generation. In A. Noor and J. Oden, editors, *State-of-the-Art Surveys on Computational Mechanics*, pages 335–366, 1989.

[17] M. S. Shephard. Finite element modeling within an integrated geometric modeling environment: Part II - attribute specification, domain differences, and indirect element types. *Engng. with Computers*, 1:72–85, 1985.

[18] M. S. Shephard. The specification of physical attribute information for engineering analysis. *Engineering with Computers*, 4:145–155, 1988.

[19] M. S. Shephard, T.-L. Sham, L.-Y. Song, V. S. Wong, R. Garimella, H. F. Tiersten, B. Lwo, Y. LeCoz, and R. B. Iverson. Global/local analyses of multichip modules: Automated 3-d model construction and adaptive finite element analysis. In *Advances in Electronic Packaging 1993*, volume 1, pages 39–49. American Society of Mechanical Engineers, 1993.

[20] Sun Microsystems. *Programming Utilities for the Sun Workstation*, 1985.

[21] H. Tiersten, T. Sham, B. Lwo, Y. Zhou, L. Song, P. Baehmann, Y. Le Coz, and M. Shephard. A global-local procedure for the thermoelastic analysis of multichip modules. In P. A. Engel and W. T. Chen, editors, *Advances in Electronic Packaging 1993*, volume 1, pages 103–118, New York, NY, 1993. ASME.

[22] B. Walton. Automatic generation of three-dimensional all-hexahedral and mixed-eelement finite element meshes. Master's thesis, Scientific Computation Research Center, Rensselaer Polytechnic Institute, Troy, NY, 1989.

[23] K. J. Weiler. *Topological Structures for Geometric Modeling*. PhD thesis, Rensselaer Design Research Center, Rensselaer Polytechnic Institute, Troy, NY, May 1986.

[24] K. J. Weiler. The radial-edge structure: A topological representation for non-manifold geometric boundary representations. In M. J. Wozny, H. W. McLaughlin, and J. L. Encarnacao, editors, *Geometric Modeling for CAD Applications*, pages 3–36. North Holland, 1988.

[25] G. Wempner. *Mechanics of Solids with Applications to Thin Bodies*. McGraw-Hill Book Company, 1928.

[26] E. Young. *Vector and Tensor Analysis*. Marcel Dekker, Inc., 1978.