REPAS — Rensselaer Electronic Packaging Analysis Software

Programmer's Manual

M. Beall
A. Garg
R. Garimella
R.B. Iverson
Y.L. Le Coz
B.-J. Lwo
T.-L. Sham
M.S. Shephard
L.-Y. Song
V.S. Wong

Rensselaer Polytechnic Institute,
Troy, NY 12180–3590

April 28, 1994

Contents

1 Introduction

1.1 Analysis Components

The analysis of MCMs requires a system of carefully coordinated software to perform thermal, thermomechanical and electromagnetic analyses. The **Rensselaer Electronic Packaging Analysis Software (REPAS)** is a seamlessly integrated system of thermal and thermomechanical analysis procedures for MCMs, with an interface to an electromagnetic analysis capability.

Since MCMs are highly complex structures with a large number of individual components in terms of wires, vias, etc., it is not practical to fully represent each of these entities in a thermal or thermomechanical analysis of the entire interconnect. At the same time, the local geometric details of these entities are critical to the prediction of the conditions under which a failure will occur. To address these problems, the thermal and thermomechanical analysis procedures of REPAS employ a global/local solution methodology where a global analysis is first performed on an idealized model to provide the boundary condition information needed for a local analysis which includes the geometric details of all the entities in that region [27].

Both the global thermal and thermomechanical analysis are performed using a unique variational technique [25, 31] which operates on a layered representation of the interconnect. The properties for each layer of the interconnect are determined based on the volume fraction of the wires using an averaging procedure appropriate for that analysis procedure. Since there are differences in the techniques required to successfully implement the variational approach for the thermal and thermomechanical analysis, separate software modules have been developed for them.

The local heat conduction analysis in REPAS is performed using a highly efficient stochastic algorithm for solving Laplace's equation [19]. The algorithm is based on the floating random-walk method [4, 11]. Briefly put, the algorithm solves the Laplace equation on a scalable cubic domain, subject to arbitrary Dirichlet conditions. A boundary-integral solution is found, from which an integral for temperature at the domain center is obtained. This integral is expanded as an infinite sum, and probability rules that statistically evaluate the sum are deduced. These rules define the algorithm, yielding temperature at a specific point within the windowed local heat conduction domain. The Dirichlet boundary

conditions are obtained from the global heat conduction analysis over the window boundary.

REPAS performs local thermomechanical analysis using adaptive finite element techniques including the possibility of nonlinear material behavior. Since there are a number of effective commercial finite element analysis codes, this capability employs the ABAQUS [12] nonlinear finite element code as the basic analysis engine. User defined extensions have been added to ABAQUS through its user routine capability to support specification of a temperature distribution. The finite element mesh is generated using the Finite Octree automatic mesh generator [28] developed at Rensselaer. To ensure the accuracy of the finite element analysis, it is run adaptively using an error estimation procedure to predict the required mesh refinements which are then carried out using the functionalities of the automatic mesh generator.

To support the electromagnetic analysis needs, the MagiCAD [24] program is linked with this software system. It is used after routing to ensure that the critical nets are performing within the noise and time specifications.

1.2 Analysis Model Generation and Integration of Analyses

Each of the REPAS analysis procedures requires a discretization of the domain of the MCM with the appropriate analysis attributes associated with that discretization. The form of the discretization, and the techniques used to define it depends on the analysis procedure being used. For each analysis type the discretization process must begin with the information in the **CIF (Caltech Intermediate Format)** file. In some cases, the discretization is constructed directly from the CIF file, while in others the information in the CIF file is used to construct a more appropriate geometric definition which is then discretized. Since CIF can be used only for description of 2–D geometry, it is supplemented by additional geometry information in an attribute file for building of the discretizations.

REPAS contains different model building procedures to construct appropriate analysis models for the two thermal and two thermomechanical analyses [30]. The model building procedures provide the global analyses with a layered description of the MCM with averaged material properties for the interconnect layers, idealized solder bump information and chip layout on the top surface of the MCM. The local thermal analysis is provided with a discretization where the conductors within the region of the analysis are represented as rectangular prisms. For

local thermal stress analysis, information in the CIF and attribute files is used to construct a 3–D non-manifold geometric model of the local region using the Parasolid solid modeler and proper extensions to support the non-manifold representation. Finite Octree is then used to generate a mesh for this model. The mesh and analysis attributes together form the finite element analysis model for local thermal stress analysis.

The construction of consistent discretizations for the different analyses, and storage and retrieval of the various model and analysis data is enabled by a generalized attribute manager, **SAM (SCOREC Attribute Manager)**. The attribute manager allows for specification of analysis information, organization of such data in a hierarchal manner and association of this information with the different discretizations. The above functions also facilitate transfer of information between analyses.

1.3 Software system

REPAS is made up of a set of programs, in-house and commercial, and scripts seamlessly integrated to perform automated thermal and thermomechanical analysis of MCMs with the capability of performing electromagnetic analysis on MCM designs. Figure 1 shows the groups of software and their interactions.

The electromagnetic analysis, done by the MagiCAD software from Mayo Foundation [24], is a stand-alone program and draws only from the CIF file augmented with some user extensions. The rest of the software (consisting of the preprocessor, and the heat conduction and thermomechanical analyses) is run by a shell script. This script in turn calls other shell scripts or programs to perform the different analyses. The preprocessor is a stand-alone program which does the input processing and the global model building. The global heat conduction analysis is a single program executed by a simple script which performs some file checks before execution. The global thermal stress analysis consists of three programs, each program performing one step of the analysis. They are executed by a shell script similar to the global heat conduction analysis script. The local heat conduction analysis is performed by single program. The local thermal stress analysis consists of 8 programs to build the local geometric model and perform an adaptive finite element analysis. They are managed by a shell script which automatically executes the adaptive loop until the solution has converged.

In contrast to the other software components, SAM is a library of functions that all programs requiring model information not in the CIF file must call. The preprocessor invokes SAM, and stores model and attribute information in the SAM database. When the preprocessor finishes building all the required attributes, it

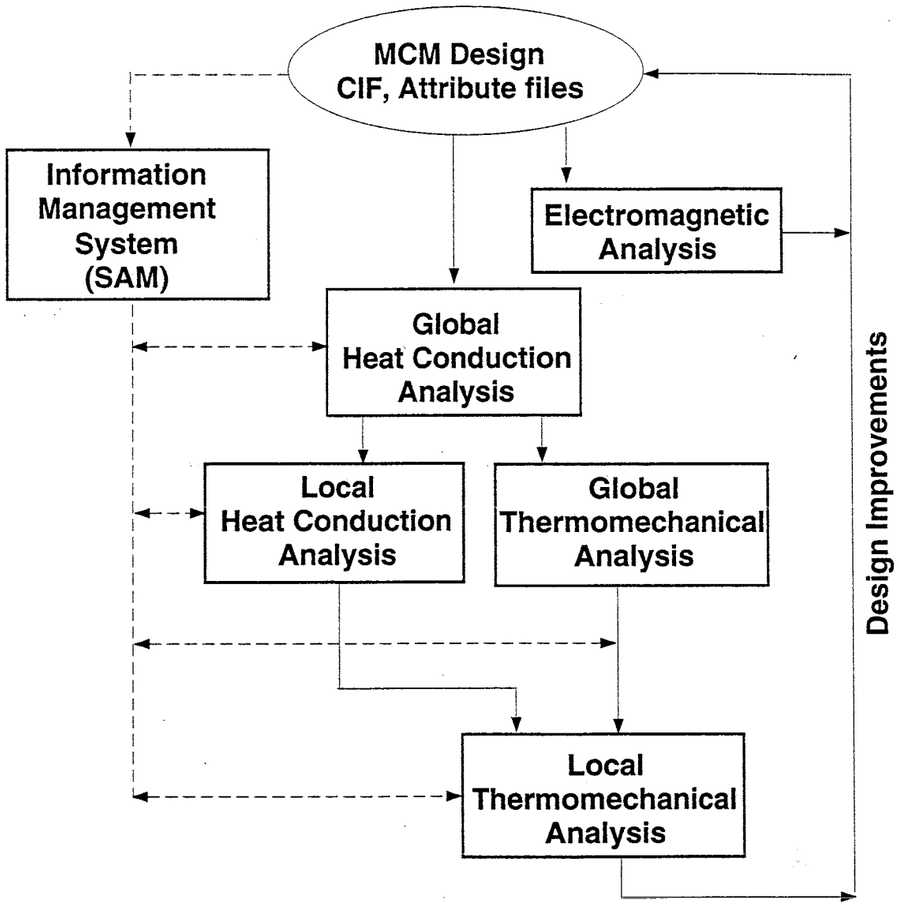


Figure 1. Overview of software functional groups and their interactions

asks SAM to write out its information onto disk. In each subsequent program that requires attribute information, the SAM database is setup from the disk files and queries made to SAM for the required information. Only local stress model building procedures modify attribute information contained in SAM apart from querying it.

In addition to the software described above, some of the individual analyses use software libraries (commercial, in-house and public domain) for their purposes. These are described in individual sections or referenced.

1.4 Note for Programmers

This manual is intended for programmers who would like to incorporate new capabilities into the REPAS software or make changes to its modules to tailor it to their own needs. Some familiarity with the concepts and use of REPAS software is assumed, although the programmer's information is supplemented by brief explanations of related concepts and references where details may be found.

Although, REPAS is made up of a large number of components, special care was taken in the design of the system, to allow for easy replaceability of components with equivalent modules. To this end, all software descriptions in this manual discuss the replaceability of that module and the changes required in the other modules. Often, the replacement of a procedure by an equivalent procedure affects its interface to only the preceding and subsequent modules in the execution sequence. Much of the REPAS code has also been designed for replaceability of individual operators or functions.

REPAS was developed by multiple groups concurrently working on different aspects of the project and involves a wide variety of software, some of which were developed afresh while others were used as is or adapted for this task. Due to this, source codes and scripts are written in a variety of languages, including FORTRAN, C and C++. Thus, in spite of the specific considerations given to the ease of the task of modifying REPAS, considerable familiarity with programming in the relevant languages and the technical details associated with the procedures is necessary for making major changes to the software. The programmer will be aided in this by the numerous citations throughout the text and by the other manuals accompanying this package.

The rest of this manual contains a description of the SCOREC Attribute Manager, the input processor or preprocessor, the global and local heat conduction

analyses, the global and local thermal stress, and the electromagnetic analysis. Each section contains (where applicable) a short description of the functionality of the component, description of important modules or subroutines, the input and output procedures, instructions for compiling the module and discussion of issues in replacing the module.

1.5 Conventions

- Filenames and path names are specified in **bold** letters.
- ***bold-italic*** portions of filenames must be substituted by an appropriate string, e.g., if one is working with a model **xyz**, the file referred to in the text as ***modelname*.cif** refers to the file **xyz.cif**.
- When described, variables of procedures are listed in *italics*.
- Pseudo-code is presented largely in `computer listing style`.
- Shell commands that must be executed are presented in `computer listing style`.
- **$NAME** refers to a path variable that is site specific. In particular, **$REPAS_HOME** is the top level directory in which the REPAS software resides.

2 SCOREC Attribute Manager — SAM

The SCOREC Attribute Manager (SAM) provides the framework for seamless integration of the analyses. This is done by providing a generalized mechanism for storing analysis attributes, maintaining associativities between attributes and models, and coordinating inter-analysis information exchange. The salient features of SAM are:

1. Tensorial specification of attributes independent of analysis type.
2. Mechanism for description of complex distributions (like non-uniform loads)
3. Ability to maintain associativity of attributes with any type of model (geometric or idealized).
4. Generalized organizational structure
5. Operator driven interface to the analyses

In the context of REPAS, SAM is used to build up attribute definitions, store them and answer analyses procedure queries about the analysis attributes. A preprocessor (Chapter 3) reads in the input files required for REPAS and builds up the attribute definitions, organizes them and associates them with the global idealized models.

Although SAM has a number of low level operators that can be used to retrieve this information, the task of retrieving information is made simpler for the analyses by two higher level operators, **AT_gtvalgbl** and **AT_gtvalocal**. **AT_gtvalgbl** is used by the two global analyses and **AT_gtvalocal**, by the two local analyses. The operators take as input the name of the analysis that is making the query, description of the type of information it is asking for (physical dimensions or material properties) and the layer number (or name) or spatial location for which this information is needed. Using input constructed from this information, these operators call an information retrieval operator of SAM and then output the retrieved data in a form that can be understood by the calling analysis.

Only the local stress analysis procedures modify the attribute database in addition to querying it. This is required since the attributes on the geometric model must be derived from the CIF layer attributes and associated with the appropriate topological entities. The creation and association of the local model attributes is done through direct calls to SAM operators.

If any of the analyses require information in a form other than what is provided, these two operators may need to be appropriately reprogrammed. This requires familiarity with the design of SAM and syntax of SAM operators. The programmer is advised to refer to the appropriate SAM documents [33] for this information.

2.1 Global and Local Attribute Retrieval Operators

2.1.1 General information retrieval — AT_gtmoname

All the REPAS procedures use the model name (specified in the file **epii.model**) as a basename to build file names from. To get the model name, the operator **AT_gtmoname** (*modelname*) is called, where *modelname* returns a character string of the name. Each analysis uses a unique extension to this name for its own output files.

2.1.2 Global Attribute Retrieval — AT_gtvalgbl

Global analyses use the operator **AT_gtvalgbl** to retrieve information about the global idealized model. This routine can be called directly from FORTRAN. It returns a list of values based on the particular layer or spatial coordinate, the type of attribute requested, and the specific attribute requested. The syntax of this operator is given below. The convention followed in describing these two operators is as follows

- **bold** = operator name
- CAPS = input
- *italics* = output
- *ITALICS-CAPS* = input may be modified and returned as output

AT_gtvalgbl (ANALYSIS_NAME, LAYER_NAME, ATTR_TYPE, ATTR_NAME, MAX_ARRAY_SIZE, *RFLAG, returned_values, value_specs, number_returned*)

☐ ANALYSIS_NAME: character string

/ The type of analysis calling the operator. For global thermal, ANALYSIS_NAME should be "gt" while it should be "gs" for global stress.*

- LAYER_NAME: character string

  / The name of the physical layer of which the information is being requested, e.g., "layer1," "layer2,"... etc. (see REPAS User's Manual [22] for a definition of a physical layer). The layers are numbered, from the bottom up, consecutively starting with one, i.e., "layer1." If the information of the chips is requested, one can use, for LAYER_NAME, "chip1," "chip2," ... etc. "layer0" is used to store information that pertains to the entire MCM and not to a particular layer. For more information on the global idealized model, refer Section 3.2.*

- ATTR_TYPE: character string

  / This specifies the type of the attribute being queried. The possible options are: "physical_dimensions," "material_properties," "boundary_condition," and "layout_info." For global stress analysis, an additional option is "initial_condition". Depending on what information is needed, the appropriate character string should be used.*

- ATTR_NAME: character string

  / The identification of the attribute. See description of valid inputs later in this section for details. For example, some possible options are "thickness," "stiffness," and "therm_expan."*

- MAX_ARRAY_SIZE: integer

  / This number is the maximum size allocated for the arrays "returned_values" and "value_specs." If the amount of data retrieved exceeds the maximum array size, only the first MAX_ARRAY_SIZE data is returned. Therefore it is important to allocate enough space for the desired values.*

- *RFLAG*: integer

  / An integer flag that tells how many total remaining attributes can be retrieved. An input of zero will mean that the calling routine will not retrieve the remaining attributes. For example, this is useful when the calling routine wants to retrieve the stiffness coefficients of a physical layer where this layer contains two materials. With the first call to AT_gtvalgbl(), "returned_values" will contain the stiffness coefficients of one of the two materials, with RFLAG indicating there is one more material left. If after this first call to AT_gtvalgbl() RFLAG is not set to zero as input, a subsequent call to AT_gtvalgbl() will retrieve the stiffness coefficients of the second material, with RFLAG being returned as zero to indicate there is no more stiffness coefficients to retrieve. If at the first call to AT_gtvalgbl() RFLAG was already set to zero as input,*

however, the stiffness coefficients of the second material cannot be retrieved with a subsequent call to AT_gtvalgbl().

- ☐ *returned_values*: double precision array

  / The values of the particular attribute are returned here. For example, attribute of thickness will return in the order of z_min and z_max. The stiffness coefficients are returned as C_{11}, C_{12}, C_{13}, ..., C_{1j}, ..., C_{ij}. If the attribute is spatially dependent and the spatial coordinates are not supplied, spatial coordinates of (0, 0, 0) are used.*

- ☐ *value_specs*: integer array

  / The contents of this array will indicate whether the corresponding position of the* returned_values *is defined or not. The content of "1" will mean that it is specified, while a content of "0" means it is not specified. For example, say a boundary condition of zero displacement only along the z direction is to be specified. Even though the boundary condition along the x and y directions are not specified, what is returned in the* returned_values *array is [0.0, 0.0, 0.0]. The corresponding* value_specs *are [0, 0, 1]. The zeros in the first two slots of* value_specs *indicate that the zeros in the first two slots of* returned_values *are meaningless and should be ignored. Therefore, this should always be checked before a value in* returned_values *is used.*

- ☐ *number_returned*: integer

  / The total number of values returned.*

The following is a list of possible inputs to **AT_gtvalgbl** and the expected outputs

1. For heat transfer coefficients:

   ANALYSIS_NAME = **gt** for global thermal
   LAYER_NAME = **chip1**, **chip2**, ..., *or* **layer1**
   ATTR_TYPE = **boundary_condition**
   ATTR_NAME = **heat_trans**
   MAX_ARRAY_SIZE = *maximum size allotted for the arrays*
   RFLAG = **0**, *since there is only one to get*
   returned_values[1] = **1**, *if convective heat transfer*
   2, *if constant temperature*
   3, *if zero flux*
   returned_values[2] = **0**, *if constant temperature or zero flux*
   non-zero number as convective heat transfer coeffi-

cient if boundary condition is convective heat transfer

returned_values[3] = *n*, n *is the reference temperature if the b.c. is convective heat transfer, if it is constant temperature b.c., then* n *is the constant temperature.*

value_specs[1],[2] = **(1, 1)**, *if the data is correct.*

number_returned = **3**

2. For number of chips:

   ANALYSIS_NAME = **gt** *for global thermal,* **gs** *for global stress*

   LAYER_NAME = **layer0**

   ATTR_TYPE = **layout_info**

   ATTR_NAME = **nchips**

   MAX_ARRAY_SIZE = *maximum size allotted for the arrays*

   RFLAG = **0**, *since there is only one to get*

   returned_values[1] = **n**, *where n is the number of chips.*

   value_specs[1] = **1** *if the data is correct.*

   number_returned = **1**

3. For coordinates of the center the n^{th} chip:

   ANALYSIS_NAME = **gt** *for global thermal,* **gs** *for global stress*

   LAYER_NAME = **chipn**, *where "n" is the chip number*

   ATTR_TYPE = **layout_info**

   ATTR_NAME = **chip_center**

   MAX_ARRAY_SIZE = *maximum size allotted for the arrays*

   RFLAG = **0**, *since there is only one to get*

   returned_values[1],[2],[3] = **(x, y, z)**

   value_specs[1],[2],[3] = **(1, 1, 1)** *if the data is correct.*

   number_returned = **3**

4. For the distribution of area of vias under the n^{th} chip to each physical layer:

   ANALYSIS_NAME = **gt** *for global thermal and* **gs** *for global stress*

   LAYER_NAME = **chipn**, *where "n" is the chip number*

   ATTR_TYPE = **layout_info**

   ATTR_NAME = **via_area**

   MAX_ARRAY_SIZE = *the maximum size allotted for the arrays*

   RFLAG = **0**, *since there is only one to get*

   returned_values[1]..[l] = **(n_1, n_1, ..., n_l)**, *where n is the total area of*

thermal vias
value_specs[1]..[l] = **(1, 1, ..., 1)** *if the data is correct.*
number_returned = **L**, *where L is the number of physical layers*

5. For the number of solder bumps under the n^{th} chip:
ANALYSIS_NAME = **gt** *for global thermal and* **gs** *for global stress*
LAYER_NAME = **chipn**, *where "n" is the chip number*
ATTR_TYPE = **layout_info**
ATTR_NAME = **nsold_bmp**
MAX_ARRAY_SIZE = *maximum size allotted for the arrays*
RFLAG = **0**, *since there is only one to get*
returned_values[1] = **n**, *where n is the number of solder bumps*
value_specs[1] = **1** *if the data is correct.*
number_returned = **1**

6. For the average diameter of a solder bump under the n^{th} chip:
ANALYSIS_NAME = **gt** *for global thermal and* **gs** *for global stress*
LAYER_NAME = **chipn**, *where "n" is the chip number*
ATTR_TYPE = **layout_info**
ATTR_NAME = **sol_diam**
MAX_ARRAY_SIZE = *maximum size allotted for the arrays*
RFLAG = **0**, *since there is only one to get*
returned_values[1] = **n**, *where n is the average diameter of a solder.*
value_specs[1] = **1** *if the data is correct.*
number_returned = **1**

7. For the width of the n^{th} chip:
ANALYSIS_NAME = **gt** *for global thermal and* **gs** *for global stress*
LAYER_NAME = **chipn**, *where "n" is the chip number*
ATTR_TYPE = **physical_dimensions**
ATTR_NAME = **chip_width**
MAX_ARRAY_SIZE = *maximum size allotted for the arrays*
RFLAG = **0**, *since there is only one to get*
returned_values[1] = **n**, *where n is the width of the chip.*
value_specs[1] = **1** *if the data is correct.*
number_returned = **1**

8. For the length of the n^{th} chip:
   ANALYSIS_NAME = **gt** *for global thermal and* **gs** *for global stress*
   LAYER_NAME = **chipn**, *where "n" is the chip number*
   ATTR_TYPE = **physical_dimensions**
   ATTR_NAME = **chip_length**
   MAX_ARRAY_SIZE = *maximum size allotted for the arrays*
   RFLAG = **0**, *since there is only one to get*
   returned_values[1] = **n**, *where n is the length of the chip.*
   value_specs[1] = **1** *if the data is correct.*
   number_returned = **1**

9. For the power density (Watts per square meter) of the n^{th} chip:
   ANALYSIS_NAME = **gt** *for global thermal*
   LAYER_NAME = **chipn**, *where "n" is the chip number*
   ATTR_TYPE = **boundary_condition**
   ATTR_NAME = **power_density**
   MAX_ARRAY_SIZE = *maximum size allotted for the arrays*
   RFLAG = **0**, *since there is only one to get*
   returned_values[1] = **n**, *where n is the power density in Watts per meter squared..*
   value_specs[1] = **1** *if the data is correct.*
   number_returned = **1**

10. For the origin offset of the MCM layer definition:
    ANALYSIS_NAME = **gt** *for global thermal and* **gs** *for global stress*
    LAYER_NAME = **layer0**
    ATTR_TYPE = **layout_info**
    ATTR_NAME = **org_offset**
    MAX_ARRAY_SIZE = *the maximum size allotted for the arrays*
    RFLAG = **0**, *since there is only one to get*
    returned_values[1],[2],[3] = **(x, y, z)**
    value_specs[1],[2],[3] = **(1, 1, 1)** *if the data is correct.*
    number_returned = **3**

11. For the minimum wire width:
    ANALYSIS_NAME = **gt** *for global thermal and* **gs** *for global stress*
    LAYER_NAME = **layer0**

ATTR_TYPE = **layout_info**
ATTR_NAME = **min_wire_width**
MAX_ARRAY_SIZE = *maximum size allotted for the arrays*
RFLAG = **0**, *since there is only one to get*
returned_values[1] = **n**, *where n is the minimum width of the wire.*
value_specs[1] = **1** *if the data is correct.*
number_returned = **1**

12. For the minimum wire pitch:
ANALYSIS_NAME = **gt** *for global thermal and* **gs** *for global stress*
LAYER_NAME = **layer0**
ATTR_TYPE = **layout_info**
ATTR_NAME = **min_wire_pitch**
MAX_ARRAY_SIZE = *maximum size allotted for the arrays*
RFLAG = **0**, *since there is only one to get*
returned_values[1] = **n**, *where n is the minimum wire pitch*
value_specs[1] = **1** *if the data is correct.*
number_returned = **1**

13. For the average geometrical information on wires of the n^{th} layer:
ANALYSIS_NAME = **gt** *for global thermal and* **gs** *for global stress*
LAYER_NAME = **layern** *where "n" is the layer number*
ATTR_TYPE = **physical_dimensions**
ATTR_NAME = **averages**
MAX_ARRAY_SIZE = *maximum size allotted for the arrays*
RFLAG = **0**, *since there is only one to get*
returned_values[1] = **a**, *where a is the average area ratio; it is zero if no wire were in this layer*
returned_values[2] = **b**, *where b is the average center-to-center pitch; it is zero if no wire were in this layer*
returned_values[3] = **c**, *where c is the average wire width; it is zero if no wire were in this layer*
value_specs[1],[2],[3] = **1** *if the data is correct.*
number_returned = **3**

14. For number of layers:
ANALYSIS_NAME = **gt** *for global thermal and* **gs** *for global stress*

LAYER_NAME = **layer0**
ATTR_TYPE = **layout_info**
ATTR_NAME = **nlayers**
MAX_ARRAY_SIZE = *the maximum size allotted for the arrays*
RFLAG = **0**, *since there is only one to get*
returned_values[1] = **n**, *where n is the number of layers.*
value_specs[1] = **1** *if the data is correct.*
number_returned = **1**

15. For the major direction of a layer:
ANALYSIS_NAME = **gt** *for global thermal and* **gs** *for global stress*
LAYER_NAME = **layern**, *where "n" is the layer number*
ATTR_TYPE = **layout_info**
ATTR_NAME = **maj_dir**
MAX_ARRAY_SIZE = *the maximum size allotted for the arrays*
RFLAG = **0**, *since there is only one to get*
returned_values[1] = **0** *(layers with no wires)* **1** *(layer with wires) if it is x-directional*; **2** *(layer with wires) if it is y-directional*
value_specs[1] = **1** *if the data is correct.*
number_returned = **1**

16. For the width of a layer:
ANALYSIS_NAME = **gt** *for global thermal and* **gs** *for global stress*
LAYER_NAME = **layer0**
ATTR_TYPE = **physical_dimensions**
ATTR_NAME = **layer_width**
MAX_ARRAY_SIZE = *maximum size allotted for the arrays*
RFLAG = **0**, *since there is only one to get*
returned_values[1] = **n**, *where n is the width of the layer.*
value_specs[1] = **1** *if the data is correct.*
number_returned = **1**

17. For the length of a layer:
ANALYSIS_NAME = **gt** *for global thermal and* **gs** *for global stress*
LAYER_NAME = **layer0**
ATTR_TYPE = **physical_dimensions**
ATTR_NAME = **layer_length**

MAX_ARRAY_SIZE = *maximum size allotted for the arrays*
RFLAG = **0**, *since there is only one to get*
returned_values[1] = **n**, *where n is the length of the layer.*
value_specs[1] = **1** *if the data is correct.*
number_returned = **1**

18. For the thickness of a layer:
ANALYSIS_NAME = **gt** *for global thermal and* **gs** *for global stress*
LAYER_NAME = **layern**, *where "n" is the layer number*
ATTR_TYPE = **physical_dimensions**
ATTR_NAME = **thickness**
MAX_ARRAY_SIZE = *maximum size allotted for the arrays*
RFLAG = **0**, *since there is only one to get*
returned_values[1] = **n**, *where n is the thickness of the layer.*
value_specs[1] = **1** *if the data is correct.*
number_returned = **1**

19. For the heat conduction coefficients of the n^{th} layer:
ANALYSIS_NAME = **gt** *for global thermal*
LAYER_NAME = **layern**, *where "n" is the layer number*
ATTR_TYPE = **material_properties**
ATTR_NAME = **thermal_cond**
MAX_ARRAY_SIZE = *maximum size allotted for the arrays*
RFLAG = **0** *to get the first one only*; **1**, *to get the heat conduction coefficients of the next material in the given physical layer with a subsequent call. As output, it will return how many more can be retrieved. For a layer with more than one material region (e.g., wires embedded in polymer), the last material returned will always be the dielectric (e.g., the polymer) and all the previous returned materials are embedded in the dielectric (e.g., the wires).*
For Isotropic material:
returned_values[1] = κ
value_specs[1] = **1** *if the data is correct.*
number_returned = **1**
For Orthotropic material:
returned_values[1],[2],[3] = (κ_{11}, κ_{22},κ_{33})
value_specs[1],[2],[3] = (**1, 1, 1**) *if the data is correct.*
number_returned = **3**

20. For the thermal expansion coefficients of the n^{th} layer:
    ANALYSIS_NAME = **gs** *for global stress*
    LAYER_NAME = **layern**, *where "n" is the layer number*
    ATTR_TYPE = **material_properties**
    ATTR_NAME = **thermal_expand**
    MAX_ARRAY_SIZE = *maximum size allotted for the arrays*
    RFLAG = **0** *to get the first one only*; **1**, *to get the thermal expansion coefficients of the next material in the given physical layer with a subsequent call. As output, it will return how many more can be retrieved. For a layer with more than one material region (e.g., wires embedded in polymer), the last material returned will always be the dielectric (e.g., the polymer) and all the previous returned materials are embedded in the dielectric (e.g., the wires).*
    For Isotropic material:
        returned_values[1] = α
        value_specs[1] = **1** *if the data is correct.*
        number_returned = **1**
    For Orthotropic material:
        returned_values[1],[2],[3] = (α_{11}, α_{22},α_{33})
        value_specs[1],[2],[3] = **(1, 1, 1)** *if the data is correct.*
        number_returned = **3**
    For Anisotropic material:
        returned_values = (α_{11}, α_{12},α_{13}, α_{22}, α_{23},α_{33})
        value_specs = **(1, 1, 1, 1, 1, 1)** *if the data is correct.*
        number_returned = **6**

21. For the stiffness coefficients of the n^{th} layer:
    ANALYSIS_NAME = **gt** *for global thermal and* **gs** *for global stress*
    LAYER_NAME = **layern**, *where "n" is the layer number*
    ATTR_TYPE = **material_properties**
    ATTR_NAME = **stiffness**
    MAX_ARRAY_SIZE = *maximum size allotted for the arrays*
    RFLAG = **0** *to get the first one only*; **1**, *to get the stiffness coefficients of the next material in the given physical layer with a subsequent call. As output, it will return how many more can be retrieved. For a layer with more than one material region (e.g., wires embedded in polymer), the last material returned will always be the dielectric (e.g., the polymer) and all the previous returned*

materials are embedded in the dielectric (e.g., the wires).

For Isotropic material:

returned_values[1],[2] = (**E**, ν)

value_specs[1],[2] = **(1, 1)** *if the data is correct.*

number_returned = **2**

For Orthotropic material:

returned_values = (c_{11}, c_{12}, c_{13}, c_{14}, c_{15}, c_{16}, c_{22}, c_{23}, c_{24}, c_{25}, c_{26}, c_{33}, c_{34}, c_{35}, c_{36}, c_{44}, c_{45}, c_{46}, c_{55}, c_{56}, c_{66}) *where* c_{14}, c_{15}, c_{16}, c_{24}, c_{25}, c_{26}, c_{34}, c_{35}, c_{36}, c_{45}, c_{46}, c_{56} *are zeros*

value_specs = **(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)** *if the data is correct.*

number_returned = **21**

For Anisotropic material:

returned_values = (c_{11}, c_{12}, c_{13}, c_{14}, c_{15}, c_{16}, c_{22}, c_{23}, c_{24}, c_{25}, c_{26}, c_{33}, c_{34}, c_{35}, c_{36}, c_{44}, c_{45}, c_{46}, c_{55}, c_{56}, c_{66})

value_specs = **(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)** *if the data is correct.*

number_returned = **21**

22. For the heat conduction coefficients of the n^{th} chip:

ANALYSIS_NAME = **gt** *for global thermal*

LAYER_NAME = **chipn**, *where "n" is the chip number*

ATTR_TYPE = **material_properties**

ATTR_NAME = **thermal_cond**

MAX_ARRAY_SIZE = *the maximum size allotted for the arrays*

RFLAG = **0** *to get the first one only*; **1**, *to get the heat conduction coefficients of the next material in the given chip with a subsequent call. As output, it will return how many more can be retrieved. For now, the chip is assumed to have been composed of entirely one material.*

For Isotropic material:

returned_values[1] = κ

value_specs[1] = **1** *if the data is correct.*

number_returned = **1**

For Orthotropic material:

returned_values[1],[2],[3] = (κ_{11}, κ_{22},κ_{33})

value_specs[1],[2],[3] = **(1, 1, 1)** *if the data is correct.*

number_returned = **3**

23. For the thermal expansion coefficients of the n^{th} chip:
    ANALYSIS_NAME = **gs** *for global stress*
    LAYER_NAME = **chipn**, *where "n" is the chip number*
    ATTR_TYPE = **material_properties**
    ATTR_NAME = **thermal_expand**
    MAX_ARRAY_SIZE = *the maximum size allotted for the arrays*
    RFLAG = **0** *to get the first one only*; **1**, *to get the thermal expansion coefficients of the next material in the given chip with a subsequent call. As output, it will return how many more can be retrieved. For now, the chip is assumed to have been composed of entirely one material.*
    For Isotropic material:
        returned_values[1] = α
        value_specs[1] = **1** *if the data is correct.*
        number_returned = **1**
    For Orthotropic material:
        returned_values[1],[2],[3] = (α_{11}, α_{22},α_{33})
        value_specs[1],[2],[3] = **(1, 1, 1)** *if the data is correct.*
        number_returned = **3**
    For Anisotropic material:
        returned_values = (α_{11}, α_{12},α_{13}, α_{22}, α_{23},α_{33})
        value_specs = **(1, 1, 1, 1, 1, 1)** *if the data is correct.*
        number_returned = **6**

24. For the stiffness coefficients of the n^{th} chip:
    ANALYSIS_NAME = **gt** *for global thermal and* **gs** *for global stress*
    LAYER_NAME = **chipn**, *where "n" is the chip number*
    ATTR_TYPE = **material_properties**
    ATTR_NAME = **stiffness**
    MAX_ARRAY_SIZE = *maximum size allotted for the arrays*
    RFLAG = **0** *to get the first one only*; **1**, *to get the heat conduction coefficients of the next material in the given chip with a subsequent call. As output, it will return how many more can be retrieved. For now, the chip is assumed to have been composed of entirely one material.*
    For Isotropic material:
        returned_values[1],[2] = (**E**, ν)

value_specs[1],[2] = **(1, 1)** *if the data is correct.*
number_returned = **2**

For Orthotropic material:

returned_values = **(c_{11}, c_{12}, c_{13}, c_{14}, c_{15}, c_{16}, c_{22}, c_{23}, c_{24}, c_{25}, c_{26}, c_{33}, c_{34}, c_{35}, c_{36}, c_{44}, c_{45}, c_{46}, c_{55}, c_{56}, c_{66})** *where c_{14}, c_{15}, c_{16}, c_{24}, c_{25}, c_{26}, c_{34}, c_{35}, c_{36}, c_{45}, c_{46}, c_{56} are zeros*

value_specs = **(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)** *if the data is correct.*

number_returned = **21**

For Anisotropic material:

returned_values = **(c_{11}, c_{12}, c_{13}, c_{14}, c_{15}, c_{16}, c_{22}, c_{23}, c_{24}, c_{25}, c_{26}, c_{33}, c_{34}, c_{35}, c_{36}, c_{44}, c_{45}, c_{46}, c_{55}, c_{56}, c_{66})**

value_specs = **(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)** *if the data is correct.*

number_returned = **21**

25. For the heat conduction coefficients of a solder bump:

ANALYSIS_NAME = **gt** *for global thermal and* **gs** *for global stress*
LAYER_NAME = **layer0**
ATTR_TYPE = **material_properties**
ATTR_NAME = **thermal_cond**
MAX_ARRAY_SIZE = *maximum size allotted for the arrays*
RFLAG = **0**, *since solder bumps are assumed to be made of only one material.*

For Isotropic material:

returned_values[1] = κ
value_specs[1] = **1** *if the data is correct.*
number_returned = **1**

For Orthotropic material:

returned_values[1],[2],[3] = (κ_{11}, κ_{22},κ_{33})
value_specs[1],[2],[3] = **(1, 1, 1)** *if the data is correct.*
number_returned = **3**

26. For the thermal expansion coefficients of a solder bump:

ANALYSIS_NAME = **gt** *for global thermal and* **gs** *for global stress*
LAYER_NAME = **layer0**
ATTR_TYPE = **material_properties**

ATTR_NAME = **thermal_expand**
MAX_ARRAY_SIZE = *maximum size allotted for the arrays*
RFLAG = **0**, *since solder bumps are assumed to be made of only one material.*
For Isotropic material:
returned_values[1] = α
value_specs[1] = **1** *if the data is correct.*
number_returned = **1**
For Orthotropic material:
returned_values[1],[2],[3] = (α_{11}, α_{22},α_{33})
value_specs[1],[2],[3] = **(1, 1, 1)** *if the data is correct.*
number_returned = **3**
For Anisotropic material:
returned_values = (α_{11}, α_{12},α_{13}, α_{22}, α_{23},α_{33})
value_specs = **(1, 1, 1, 1, 1, 1)** *if the data is correct.*
number_returned = **6**

27. For the spring constant bottom layer of the interconnect:
ANALYSIS_NAME = **gs** *for global stress*
LAYER_NAME = **layer0**
ATTR_TYPE = **material_properties**
ATTR_NAME = **spring**
MAX_ARRAY_SIZE = *maximum size allotted for the arrays*
RFLAG = **0**, *since there is only one to get*
For Isotropic material:
returned_values[1] = **k**
value_specs[1] = **1** *if the data is correct.*
number_returned = **1**
For Orthotropic material:
returned_values = ($\mathbf{k_{11}}$, $\mathbf{k_{22}}$, $\mathbf{k_{33}}$)
value_specs = **(1, 1, 1)** *if the data is correct.*
number_returned = **3**
For Anisotropic material:
returned_values = ($\mathbf{k_{11}}$, $\mathbf{k_{12}}$, $\mathbf{k_{13}}$, $\mathbf{k_{21}}$, $\mathbf{k_{22}}$, $\mathbf{k_{23}}$, $\mathbf{k_{31}}$, $\mathbf{k_{32}}$, $\mathbf{k_{33}}$)
value_specs = **(1, 1, 1, 1, 1, 1, 1, 1, 1)** *if the data is correct.*
number_returned = **9**

28. For the temperature expansion coefficients:
    ANALYSIS_NAME = **gs** *for global stress*
    LAYER_NAME = **layern** or **chipn**
    ATTR_TYPE = **boundary_condition**
    ATTR_NAME = **temp_expand**
    MAX_ARRAY_SIZE = *maximum size allotted for the arrays*
    RFLAG = **0**, *since there is only one to get*
    returned_values[1] = *vector of temperature expansion coefficients*
    value_specs[1] = **(1,1,1...)**
    number_returned = **4+3*(xnum+1)*(ynum+1)** *for layers and* **6+6*(xnum+1)*(ynum+1)** *for chips, where* **xnum** *and* **ynum** *are the number of Fourier series terms in the x and y directions respectively.*

29. Initial temperature of interconnect:
    ANALYSIS_NAME = **gt** *for global thermal and* **gs** *for global stress*
    LAYER_NAME = **layern** or**chipn**
    ATTR_TYPE = **initial_condition**
    ATTR_NAME = **initial_temp**
    MAX_ARRAY_SIZE = *maximum size allotted for the arrays*
    RFLAG = **0**, *since there is only one to get*
    returned_values[1] = **T**
    value_specs[1] = **1** *if data is correct*
    number_returned = **1**

30. Processing temperature of interconnect:
    ANALYSIS_NAME = **gt** *for global thermal and* **gs** *for global stress*
    LAYER_NAME = **layern** or**chipn**
    ATTR_TYPE = **initial_condition**
    ATTR_NAME = **process_temp**
    MAX_ARRAY_SIZE = *maximum size allotted for the arrays*
    RFLAG = **0**, *since there is only one to get*
    returned_values[1] = **T**
    value_specs[1] = **1** *if data is correct*
    number_returned = **1**

31. For stiffness of pin (spring, hook, ..) at the bottom of the substrate:
    ANALYSIS_NAME = **gs** *for global stress*

LAYER_NAME = **layer0**
ATTR_TYPE = **material_properties**
ATTR_NAME = **stiffness_pins**
MAX_ARRAY_SIZE = *the maximum size allotted for the arrays*
RFLAG = **0**, *since there is only one to get*
returned_values[1],[2] = (**E**,ν), *where* **E** *is Young's modulus and* ν *is Poisson's Ratio*
value_specs[1],[2] = (**1**, **1**) *if the data is correct*
number_returned = **2**

32. For total number of pins:
ANALYSIS_NAME = **gs** *for global stress*
LAYER_NAME = **layer0**
ATTR_TYPE = **layout_info**
ATTR_NAME = **npins**
MAX_ARRAY_SIZE = *the maximum size allotted for the arrays*
RFLAG = **0**, *since there is only one to get*
returned_values[1] = **n**, *where* **n** *is the total number of pins.*
value_specs[1] = **1** *if the data is correct*
number_returned = **1**

33. For the geometry of pins:
ANALYSIS_NAME = **gs** *for global stress*
LAYER_NAME = **layer0**
ATTR_TYPE = **physical_dimensions**
ATTR_NAME = **pin_size**
MAX_ARRAY_SIZE = *the maximum size allotted for the arrays*
RFLAG = **0**, *since there is only one to get*
For circular cross-section
returned_values[1],[2] = (**h**, **d**) *where* **h** *is the average height of the pins and* **d** *is the average diameter of the pins.*
value_specs[1],[2] = (**1**, **1**) *if the data is correct*
number_returned = **2**
For rectangular cross-section
returned_values[1],[2] = (**h**, **w**, **l**) *where* **h** *is the average height of the pins,* **w** *is the average width of the pins, and* **l** *is the average length of the pins.*

value_specs[1],[2],[3] = (**1**, **1**,**1**) *if the data is correct*
number_returned = **3**

2.1.3 Local Attribute Retrieval Operator — AT_gtvalocal

AT_gtvalocal (ANALYSIS_NAME, CIF_LAYER_NAME, POINT_SPATIAL_LOCATION, ATTR_TYPE, ATTR_NAME, MAX_ARRAY_SIZE, *RFLAG, returned_values, value_specs, number_returned*)

/* This routine is used by the two local analyses to retrieve information from SAM. It returns a list of values based on the particular CIF layer or spatial coordinate of a point location, the type of attribute wanted, and the specific attribute wanted.

- ANALYSIS_NAME: character string
  / *The type of analysis calling the operator. For global thermal, ANALYSIS_NAME should be "lt" while it should be "ls" for local stress.*
- CIF_LAYER_NAME: character string
  / *The name of the CIF layer, as read from the CIF file, of which the information is being requested, e.g., "signal_x," "ground,"... etc. If the calling routine is querying for pointwise information, this argument must be given as NULL, with the spatial location given in the next argument.*
- POINT_SPATIAL_LOCATION: double precision array
  / *The CIF_LAYER_NAME should be NULL and a location given here if the value being inquired is spatially dependent. In this case, the value returned is evaluated on the location given in this argument. The spatial location should be specified in the order of (x, y, z).*
- ATTR_TYPE: character string
  / *This specifies the type of the attribute being queried. The possible options are: "physical_dimensions," "material_properties," "boundary_condition," "problem_definition," and "accuracy_constants." Depending on what information is needed, the appropriate character string should be used.*
- ATTR_NAME: character string
  / *The identification of the attribute. Some possible options are "thickness," "local_wind," and "heat_cond." The complete set of valid names are given later in this section.*

- ☐ MAX_ARRAY_SIZE: integer

  / This number is the maximum size allocated for the arrays "returned_values" and "value_specs." If the amount of data retrieved exceeds the maximum array size, only the first MAX_ARRAY_SIZE data are returned. Therefore it is important to allot enough space for the desired values.*

- ☐ *RFLAG*: integer

  / An integer flag that tells how many total remaining attributes can be retrieved. When there is only one attribute of a particular name and type to be retrieved, RFLAG must be sent in as zero. When there are multiple attributes of a particular name and type to be retrieved, this flag should be sent in as zero, on the first call. On subsequent calls, RFLAG must be sent in as it was returned on the previous call until it finally becomes zero again, indicating that all the attributes of that name and type have been retrieved.*

- ☐ *returned_values*: double precision array

  / The values of the particular attribute are returned here. For example, attribute of thickness will return in the order of z_min and z_max. The stiffness coefficients are returned as C_{11}, C_{12}, C_{13}, ..., C_{1j}, ..., C_{ij}. If the attribute is spatially dependent and the spatial coordinates are not supplied, spatial coordinates of (0, 0, 0) are used.*

- ☐ *value_specs*: integer array

  / The contents of this array will indicate whether the corresponding position of the* returned_values *is defined or not. The content of "1" will mean that it is specified, while a content of "0" means it is not specified. For example, a boundary condition of zero displacement may have been specified, say, only along the z-direction. Even though the boundary condition along the x and y directions are not specified, what is returned in the* returned_values *array is [0.0, 0.0, 0.0]. The corresponding* value_specs *are [0, 0, 1]. The zeros in the first two slots of* value_specs *indicate that the zeros in the first two slots of* returned_values *are not specified. Therefore, this should always be checked before a value in* returned_values *is used.*

- ☐ *number_returned*: integer

  / The total number of values returned.*

The detailed list of possible input and output parameters to **AT_gtvalocal** is as follows:

1. For the local window coordinates:
   CIF_LAYER_NAME = *CIF layer name*
   POINT_SPATIAL_LOCATION[1],[2],[3] = **(0, 0, 0)**
   ATTR_TYPE = **problem_definition**
   ATTR_NAME = **local_wind**
   MAX_ARRAY_SIZE = *maximum size allotted for the arrays*
   RFLAG = **0**, *since there is only one to get*
   returned_values[1],[2],[3],[4],[5],[6] = $(x, y, z)_{min}$, $(x, y, z)_{max}$
   value_specs[] = **(1, 1, 1, 1, 1, 1)** *if the data is correct.*
   number_returned = **6**

2. For the position resolution:
   CIF_LAYER_NAME = **layer0**
   POINT_SPATIAL_LOCATION[1],[2],[3] = **(0, 0, 0)**
   ATTR_TYPE = **accuracy_constants**
   ATTR_NAME = **position_res**
   MAX_ARRAY_SIZE = *the maximum size allotted for the arrays*
   RFLAG = **0**, *since there is only one to get*
   returned_values[1] = **n**, *where n is the position resolution*
   value_specs[] = **1** *if the data is correct.*
   number_returned = **1**

3. For the temperature resolution:
   CIF_LAYER_NAME = **layer0**
   POINT_SPATIAL_LOCATION[1],[2],[3] = **(0, 0, 0)**
   ATTR_TYPE = **accuracy_constant**
   ATTR_NAME = **temp_res**
   MAX_ARRAY_SIZE = *maximum size allotted for the arrays*
   RFLAG = **0**, *since there is only one to get*
   returned_values[1] = **n**, *where n is the temperature resolution*
   value_specs[] = **1** *if the data is correct.*
   number_returned = **1**

4. For the thickness of a particular CIF layer:
   CIF_LAYER_NAME = *CIF layer name*
   POINT_SPATIAL_LOCATION[1],[2],[3] = **(0, 0, 0)**
   ATTR_TYPE = **physical_dimensions**

ATTR_NAME = **thickness**
MAX_ARRAY_SIZE = *the maximum size allotted for the arrays*
RFLAG = **0**, *just want the first one since there is only one to get*
returned_values[1],[2] = **(z_min, z_max)** *where z_max – z_min is the thickness of the CIF layer*
value_specs[1],[2] = **(1, 1)** *if the data is correct.*
number_returned = **2**

5. For the heat conduction coefficients of a particular CIF layer:
CIF_LAYER_NAME = *CIF layer name*
POINT_SPATIAL_LOCATION[1],[2],[3] = **(0, 0, 0)**
ATTR_TYPE = **material_properties**
ATTR_NAME = **heat_cond**
MAX_ARRAY_SIZE = *maximum size allotted for the arrays*
RFLAG = **0**, *to get the first one only*; **1** *to get the heat conduction coefficients of the next material in the given physical layer with a subsequent call. As output, it will return how many more can be retrieved.*
For Isotropic material:
returned_values[1] = κ
value_specs[1] = **1** *if the data is correct.*
number_returned = **1**
For Orthotropic material:
returned_values[1],[2],[3] = (κ_{11}, κ_{22}, κ_{33})
value_specs[1],[2],[3] = **(1, 1, 1)** *if the data is correct.*
number_returned = **3**

6. For temperature field from the global heat conduction analysis at a given spatial location:
CIF_LAYER_NAME = *NULL*
POINT_SPATIAL_LOCATION[1],[2],[3] = **(x, y, z)**
ATTR_TYPE = **boundary_condition**
ATTR_NAME = **temperature**
MAX_ARRAY_SIZE = *maximum size allotted for the arrays*
RFLAG = **0** *since there is only one for this location*
returned_values[1] = **T**
value_specs[1] = **1** *if the data is correct.*

number_returned = **1**

7. For displacement field from the global thermal stress analysis at a given spatial location:
   CIF_LAYER_NAME = *NULL*
   POINT_SPATIAL_LOCATION[1],[2],[3] = **(x, y, z)**
   ATTR_TYPE = **boundary_condition**
   ATTR_NAME = **displacement**
   MAX_ARRAY_SIZE = *the maximum size allotted for the arrays*
   RFLAG = **0** *since there is only one for this location*
   returned_values[1] = $\mathbf{U_x}$, $\mathbf{U_y}$, $\mathbf{U_z}$
   value_specs[1] = **1** *if the data is correct.*
   number_returned = **1**

3 Preprocessing — REPASpre

REPASpre is an input processing procedure which translates the input information into attributes and derives idealized models from the input. The building of idealizations for client analyses (analyses querying the SAM for model attributes) requires a knowledge of all the information associated with the domains to be analyzed. The first to be built is the physical model, from which all other domains are derived. The *physical model* is an abstract description of the MCM in terms of layers, referred to as physical layers. Each physical layer contains one or more CIF layers and the structure of its cross-section is constant through the layer thickness. To maintain consistency in the information used to build the analysis domains for the global and local procedures, each client analysis domain is derived from the physical model domain. The domain information is obtained from the CIF file supplemented with information from the attribute file. All other model building procedures draw upon this representation for information about the MCM.

REPASpre primarily consists of essentially three major parts: database preparation, physical model building, and global idealized model building. Figure 1 shows the algorithm of the REPASpre for preprocessing the input data.

First, the data structure for the creation and storage of attributes for the various model domains are set up. Since SAM is the means of storing domain information and providing it to client analyses, it is setup before any information is processed. This enables the interface operators of SAM be used in the model building process.

Then the CIF file is parsed from top to bottom to obtain the CIF layer information for the physical model domain (see Section 7.9 for a description of the CIF parser). The cumulative x-y cross-sectional areas of each CIF layer is tracked during this first parsing step. As each new CIF layer is encountered, a set of routines is called to build the physical model from the corresponding attribute information specified in the attribute file. Three 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. This first pass ignores any definition with wire path or polygon commands; all CIF components, except solder bumps, must be specified in box form. 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 cumulative x-y cross-sectional area of the CIF

layer in the physical model. As the need arises, extensions to other geometries may be added simply as more parsing options.

After the CIF file has been completely parsed, the idealized global model is then built. Another pass through the CIF file is made at this time to obtain information of the solder bumps and vias under each chip. The global model attributes are then created and added to the global model. Finally, all the attributes is stored in three output files to be used for each of the five client analyses.

3.1 Physical Model Building

3.1.1 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 — until all CIF layers are processed. The additional layer of pins specified in the attribute file but not in CIF are then processed in the same way.

The requirements on the input 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 2. 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 in Section 5.3 of REPAS User's Manual [22] 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 CIFsup(), 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 2. These modules search for and locate in the attribute file the target level 2 data

```
REPASpre()
{
  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 1 REPAS preprocessing

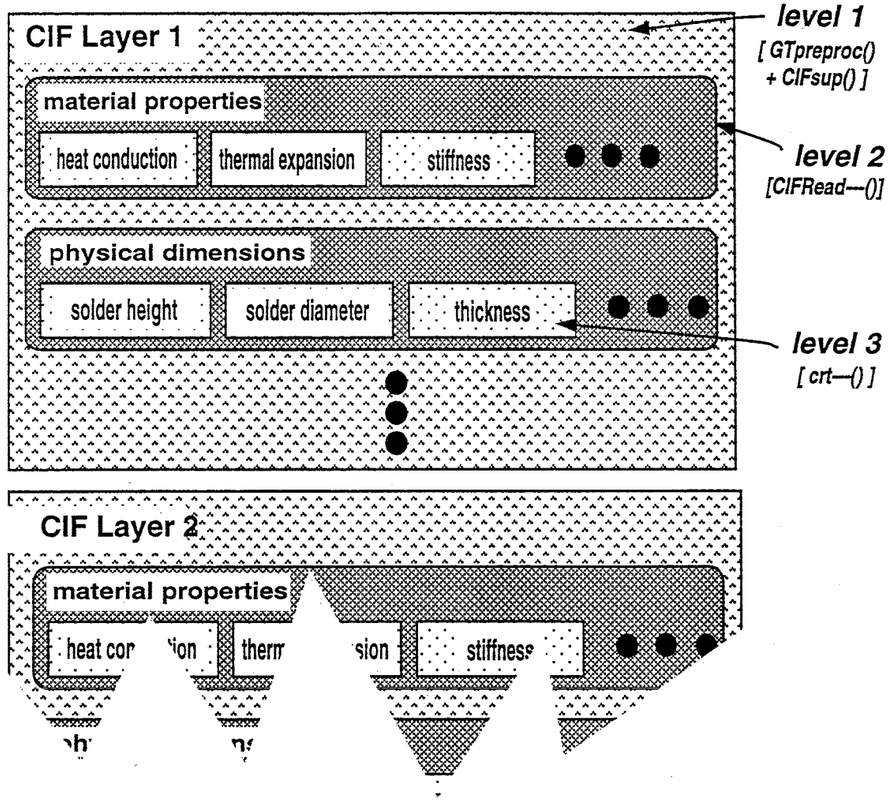


Figure 2. Attribute File Parsing Routine Format

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 [33] 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

```
/* 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 setup 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();

     /* 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 2 Attribute Processing Modules

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 5.3 of REPAS User's Manual [22] are strictly followed, and the integrity of the attributes created is strongly dependent upon the strict adhesion to the rules.

```
/* 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 linked list */
  struct attlist *next;           /* of attributes */
  void *attPtr;
} attlist;

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

Pseudo-Code 3 Data Structure Definition for Physical Model

3.1.2 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 3. 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 sorting are done through 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 1 shows an example of the

CIF name	z-min	z-max	Area	Attributes
substrate	0	0.00060	5.94e-3	*pointer*→
gnd_1	0.00060	0.000625	5.94e-3	*pointer*→
via_ground	0.000625	0.0007222	2.75e-7	*pointer*→
vdd	0.000625	0.000650	5.94e-3	*pointer*→
dielectric_ins1	0.000650	0.000680	5.94e-3	*pointer*→
via_power	0.000650	0.0007222	2.95e-7	*pointer*→
signal_h	0.000662	0.000668	1.82e-4	*pointer*→
via_contact	0.000668	0.000698	3.29e-07	*pointer*→
gnd_2	0.000680	0.000686	5.94e-3	*pointer*→
dielectric_ins2	0.000686	0.000716	5.94e-3	*pointer*→
signal_v	0.000698	0.000704	7.90e-5	*pointer*→
gnd_3	0.000716	0.000722	5.94e-3	*pointer*→
chip_10	0.0007222	0.0007972	1.00e-4	*pointer*→
chip_9	0.0007222	0.0007972	1.00e-4	*pointer*→
chip_2	0.0007222	0.0007972	1.00e-4	*pointer*→
chip_8	0.0007222	0.0007972	1.00e-4	*pointer*→
chip_19	0.0007222	0.0007972	1.00e-4	*pointer*→
:	:	:	:	:

Table 1 Physical Model Information Stored in a 25–Chip MCM Design

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.

3.2 Global 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

thermomechanical analyses. The information of the actual MCM is obtained from the physical model created in the previous preprocessing step. With the exception of those layers containing the interconnect signals, all other layers are modeled as homogeneous global layers with averaged properties of the real physical layer; the vias through each layer are considered to have negligible area compared to the interconnect and hence have little effect on the material properties. The layers with the interconnect signals are idealized as 2–material heterogeneous layers, consisting of dielectrics and signals. The two global analyses use this information to determine layer properties based on the volume fraction of the wires using an averaging procedure appropriate for that analysis. In addition to interconnect layer information, the global idealized model also contains the layout, geometry and material properties of chips, and solder and pin information. Figure 3 shows a schematic of the physical model on the left and the global idealized model on the right.

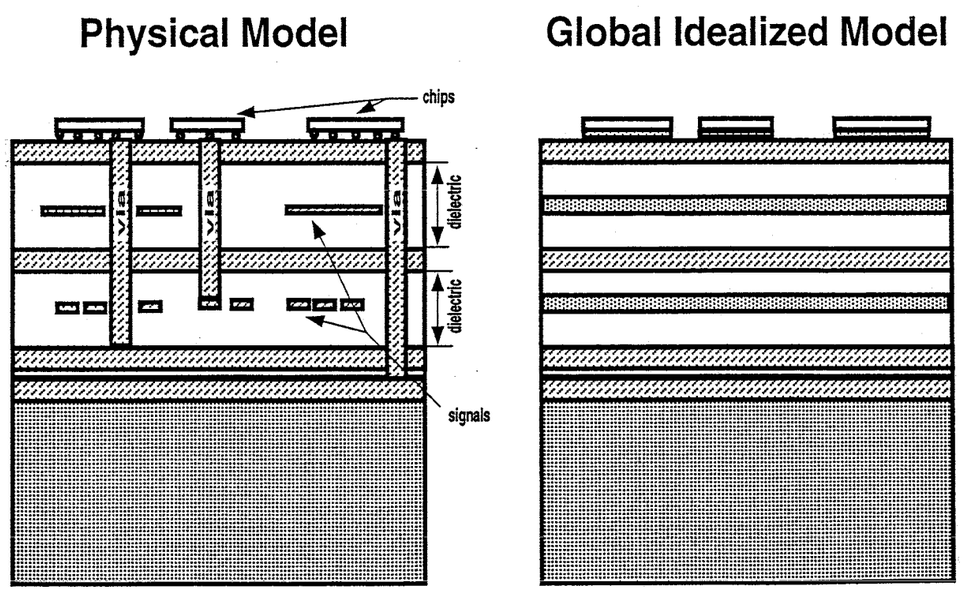


Figure 3. Illustration of the Global Idealized Model

3.2.1 Approach

The strategy for building the idealized model from the physical model requires two major operations (Figure 4): a) decompose the physical model into appropriate layers and b) 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.

As one can see from step 2 of Figure 4, 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 are 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 averaged height of the solder bumps under each chip.

This algorithm is illustrated in detail in Pseudo-Code 4 and some of the important points are discussed following the pseudo-code.

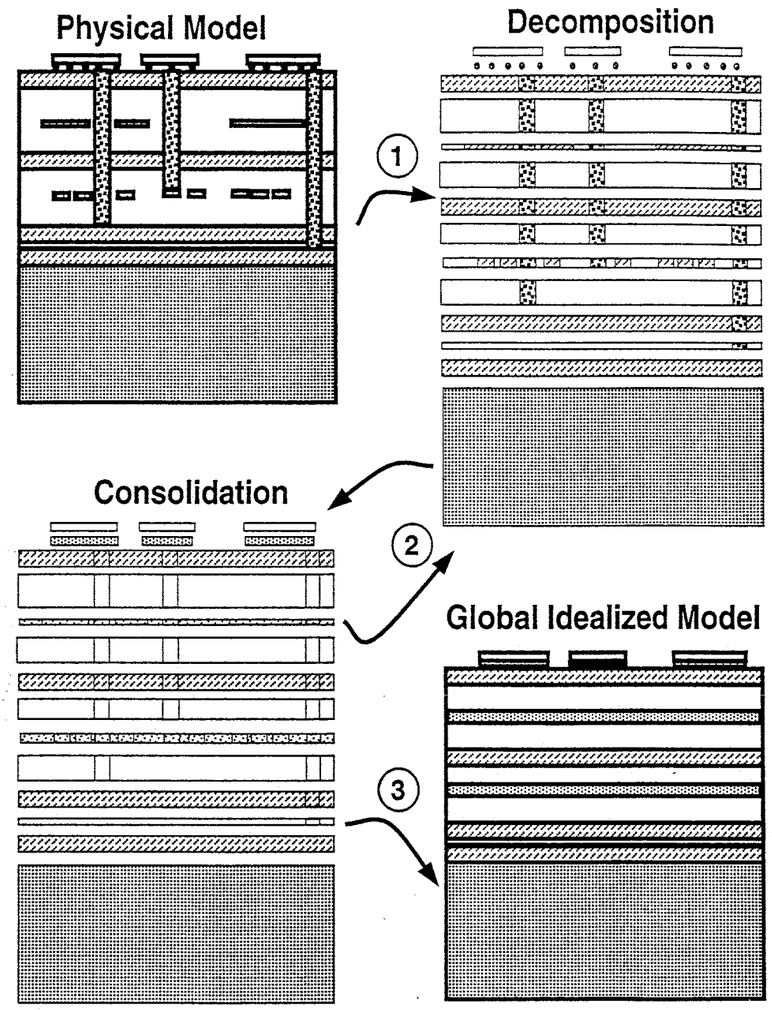


Figure 4. Global Idealized Model Building Approach

Pseudo-Code 4 Global Model Building Algorithm (Continued) . . .

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

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

   /* based on the keyword, sift out all chips, substrate, via &
dielectric layers */
   switch( keyword )
   {
     case "substrate":
     {

        /* From the assumption, the substrate is always physically
         * the bottom-most layer of the MCM. Therefore, the substrate
         * should be the first CIF layer encountered. This layer is
         * set to be the first (bottom) layer of the global model */
        Set_Global_Layer

        /* set back pointer on CIF layer to this first 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 back pointer of CIF layer to negative of the chip */
        Set_Back_Pointer
     }
     case "via":
     {
        /* keep count of number of and keep track of the vias
encountered */
        Store_Via_Index
        Increment_Via_Count
```

Pseudo-Code 4 Global Model Building Algorithm (Continued) . . .

```
    }
    case "dielectric":
    {
       /* keep count of 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 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 # of CIF layers this global layer contains*/
         Increment_Member_layers

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

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

Pseudo-Code 4 Global Model Building Algorithm (Continued) . . .

```
 * 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 over member CIF layers & retrieve owning attributes */
  Foreach ( Member_CIF_Layer )

   {
      Get_Attributes
      /* loop through each attribute and associate it with
       * 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
    * current chip */
   Foreach ( attribute )

      Associate_Attribute_With_Chip
}
```

Pseudo-Code 4 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 5, **MCM Physical Description** of [22]. 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. Figure 1 shows the CIF layers of the physical model sorted in this order of the vertical position. Notice the first

layer (i.e., the bottommost layer) must necessarily be the substrate according to assumption 13 of the **CIF File Specifications** (Section 5.2, [22]). 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 5.2, [22]) 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 4 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 1 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 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 5 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 5.2 **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.

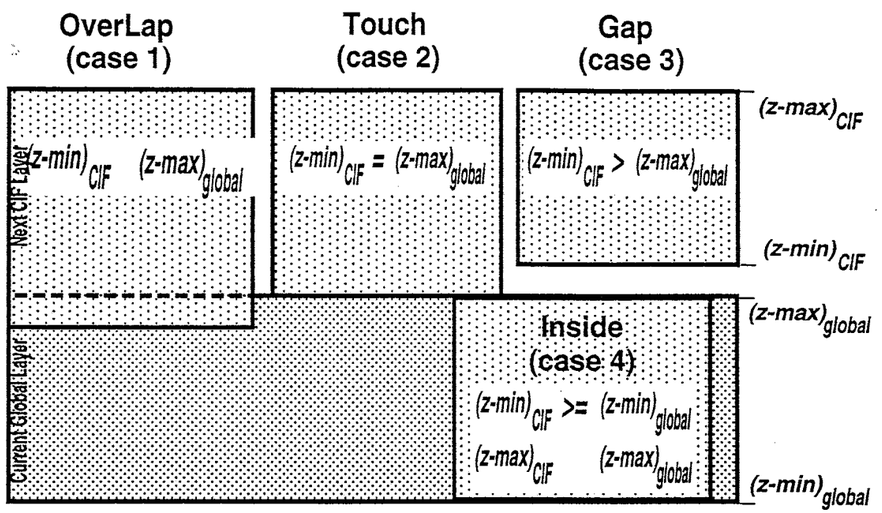


Figure 5. Criteria of Layer Comparison

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 5.2, 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 solders 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 5.2) that the x-y dimension of the substrate is used as the x-y dimension of the MCM. The dielectric layers are basically used as "fillers" for the global layer, that is, it fills in gaps between two layers as well as filling 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.

3.2.2 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 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 6 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 Figure 1. A tabulation of the global model information is shown in Figure 2. 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, the two dielectric layers are broken into three global layers each. Within a global layer, the first position of the 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

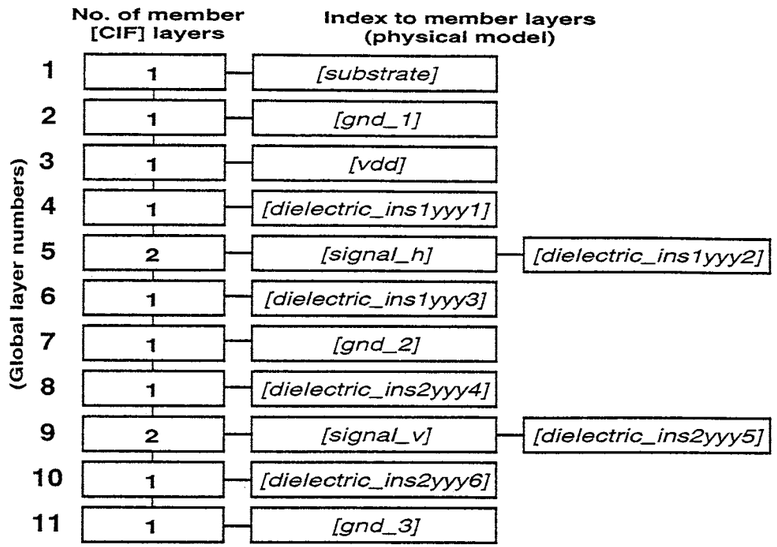


Figure 6. Data Structure of the Global Idealized Model

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	z-max	Area	Attributes
1	substrate	0	0.00060	5.94e-3	*pointer→*
2	gnd_1	0.00060	0.000625	5.94e-3	*pointer→*
3	vdd	0.000625	0.000650	5.94e-3	*pointer→*
4	dielectric_ins1yyy1	0.000650	0.000662	5.94e-3	*pointer→*
5	signal_h	0.000662	0.000668	1.82e-4	*pointer→*
	dielectric_ins1yyy2	0.000662	0.000668	5.76e-3	*pointer→*
6	dielectric_ins1yyy3	0.00068	0.000680	5.94e-3	*pointer→*
7	gnd_2	0.000680	0.000686	5.94e-3	*pointer→*

Table 2 Global Model Extracted From the Physical Model (Continued) . . .

Global Layer	CIF name	z-min	z-max	Area	Attributes
8	dielectric_ins2yyy4	0.000686	0.000698	5.94e-3	*pointer→*
9	dielectric_ins2yyy5	0.000698	0.000704	5.86e-3	*pointer→*
	signal_v	0.000698	0.000704	7.90e-5	*pointer→*
10	dielectric_ins2yyy6	0.000704	0.000716	5.94e-3	*pointer→*
11	gnd_3	0.000716	0.000722	5.94e-3	*pointer→*

Table 2 Global Model Extracted From the Physical Model

4 Global Heat Conduction Analysis

4.1 Description and Algorithm

The global heat conduction analysis procedure computes a steady-state temperature field in the MCM in order to provide

1. the thermal loading conditions for the global thermal stress analysis, and
2. the temperature boundary conditions for the local thermal analysis.

The basis of the computation is a variational principle given by Tiersten [31] for a vector system of equations. In this principle, all boundary conditions of the constrained type are transformed to natural boundary conditions by the use of Lagrangian multipliers. This variational principle can be specialized to the case of a scalar system and thus is well-suited for the steady-state heat conduction calculation.

In the global heat conduction analysis, the signal planes which contain discrete metal wires and insulating polymer are represented as "effective" homogeneous layers, having effective thermal properties. Zero normal heat flux boundary conditions are enforced exactly in the software on (i) the four external vertical faces of the interconnect, and (ii) the four external vertical faces of each chip. A combination of constant temperature, zero normal heat flux and convective heat transfer thermal boundary conditions can be selected on (i) the bottom surface of the interconnect and (ii) the top of the chips. The software allows for the selection of the thermal boundary conditions on a chip by chip basis. Layerwise geometrical information and attributes are obtained by querying SAM through operators. The output of the global heat conduction analysis involves information on the global temperature field. The output data is organized so that the local thermal analysis can obtain the global temperature at any specified spatial location through operators managed by SAM. Also, the coefficients of the series representation of the global temperature field required by the global thermal stress analysis are organized in a form that can be queried by the global thermal stress analysis through the operators managed by SAM.

4.2 Source Code

4.2.1 Global Heat Conduction Analysis

The source codes for the global heat conduction analysis are stored in groups according to their functionalities. The top level sub-directory is called **$REPAS_HOME/gt/src/analysis** and it has 7 sub-directories: **Main**, **Sam**, **Database**, **Util**, **Form**, **Eqn** and **Post**. The sub-directory **Main** contains the main program and a driver that controls the flow of the analysis. There is also a file called **dimen.h** which contains parameter statements that control the sizes of the arrays in the global heat conduction analysis program. The sub-directory **Sam** contains routines that interface with SAM. These routines query SAM to obtain the geometrical data and the material properties of the MCM and the thermal boundary conditions. The layerwise thermal properties are then computed and organized for subsequent use in the program. The routines in the sub-directory **Database** are responsible for setting up the database structure of the global heat conduction analysis. The routines in the sub-directory **Form** carry out the evaluation of the integrals in the variational principle and assemble the coefficient matrix and the right hand side vector to form the linear algebraic equations. These equations are solved by the routines in the sub-directory **Eqn** which contains routines that perform diagonal scaling of the coefficient matrix and a symmetric equation solver from the linear algebra package LAPACK. The source codes for the equation solver are placed in the sub-directories **LAPACK** and **BLAS** under the sub-directory **Eqn**. The output data from the global heat conduction analysis are prepared by the routines in the sub-directory **Post**. These data are to be used by the global thermal stress and local thermal analyses. The data can also be used by the post-processing program to generate a suitable database for visualization of the results from the global heat conduction analysis. Routines in the sub-directory **Post**, also perform calculations to estimate how well the interface and boundary conditions are satisfied by the global heat conduction solution. Finally, the files in the sub-directory **Util** are utility routines that are frequently called by the global heat conduction analysis.

4.2.2 Global Heat Conduction Interface Routines

The global heat conduction analysis generates the temperature field in the MCM based on layerwise effective thermal properties. The temperature field is

used by the global thermal stress and local thermal analyses through interface Fortran and C routines. The source codes of these interface routines are stored in the sub-directory **$REPAS_HOME/gt/src/interface**.

4.3 Input and Output

The necessary inputs for the global heat conduction analysis are obtained through the attribute manager SAM and program parameters set up in the global heat conduction analysis code. The output consists of a data file which is to be used by the global thermal stress and local thermal analyses. The following is a description of the input and output.

4.3.1 Input from SAM

In the initial phase of the global heat conduction analysis, the geometrical information and material data of the MCM, as well as the thermal boundary conditions, are obtained by querying the attribute manager SAM. In this input phase, the following parameters are either obtained directly from SAM or computed using information provided by SAM.

1. *x0mcm*, *y0mcm* (reals) - The size of the interconnect in the x and y-directions.
2. *zmcm[i]* (real) - The thickness of interconnect layer no. *i*. The bottommost layer in the interconnect is taken to be layer no. 1.
3. *geochp[1,j]* (real array) - The size of the j^{th} chip/solder unit in the x-direction.
4. *geochp[2,j]* (real array) - The size of the j^{th} chip/solder unit in the y-direction.
5. *geochp[3,j]* (real array) - The thickness of the chip of the j^{th} chip/solder unit.
6. *geochp[4,j]* (real array) - The thickness of the solder continuum of the j^{th} chip/solder unit.
7. *kmcm[1,j]*, *kmcm[2,j]*, *kmcm[3,j]* (real arrays) - The x, y, z components, respectively, of the effective heat conduction coefficients in the j^{th} layer of the interconnect.
8. *kchp[1,j]*, *kchp[2,j]*, *kchp[3,j]* (real arrays) - The x, y, z components, respectively, of the effective heat conduction coefficients of the chip in the j^{th} chip/solder unit.
9. *ksld[1,j]*, *ksld[2,j]*, *ksld[3,j]* (real arrays) - The x, y, z components, respectively, of the effective heat conduction coefficients of the solder continuum in the j^{th} chip/solder unit.

10. *bcmcm[1]* (real array) - The value of the prescribed constant temperature which enters into the effective convective heat transfer condition for the bottom surface of the interconnect.
11. *bcmcm[2]* (real array) - The value of the effective convective heat transfer coefficient for the bottom surface of the interconnect.
12. *bcchp[1,j]* (real array) - The value of the prescribed constant temperature which enters into the effective convective heat transfer condition on the top surface of the j^{th} chip/solder unit.
13. *bcchp[2,j]* (real array) - The value of the effective convective heat transfer coefficient on the top surface of the j^{th} chip/solder unit.
14. *bcchp[3,j]* (real array) - The surface heat generation density measured in W/m^2 for the j^{th} chip/solder unit.
15. *omcm[1]*, *omcm[2]*, *omcm[3]* (real) - The x, y, z offsets, respectively, between the origins of the CIF coordinates and the global thermal coordinates. These input values are used to transform the input CIF coordinates to the ones used in the global thermal analysis.
16. *ochp[1,j]*, *ochp[2,j]* (real arrays) - The x and y global thermal coordinates, respectively, of the corner of the j^{th} chip/solder unit.
17. *ntmcm[1]*, *ntmcm[2]* (integers) - The number of cosine terms used to represent the variation of the temperature field in the interconnect in the x and y directions, respectively.
18. *ntmcm[3]* (integer) - Control flag for denoting the different types of thermal boundary condition on the bottom of the interconnect, 1 - for effective convective heat transfer boundary condition, 2 - constant temperature boundary condition, 3 - zero normal heat flux boundary condition.
19. *ntmcm[4]* (integer) - The number of analysis layers in the interconnect.
20. *nunit* (integer) - The number of chip/solder continuum units placed on the top surface of the interconnect.
21. *ntchp[1,j]*, *ntchp[2,j]* (integer arrays) - The number of cosine terms used to represent the variation of the temperature field in the j^{th} chip/solder unit and in the x and y directions, respectively.
22. *ntchp[3,j]* (integer array) - Control flag for denoting the different types of thermal boundary condition on the top surface of the j^{th} chip/solder unit, 1 - for effective convective heat transfer boundary condition, 2 - constant temperature boundary condition, 3 - zero normal heat flux boundary condition.

4.3.2 Dimension Parameters

This section lists parameters which are related to maximum array dimensions in the global thermal analysis. They are set up in the file **$REPAS_HOME/gt/src/analysis/Main/dimen.h**. These arrays should be configured to handle the largest problem that must be analyzed. Note that all the following parameters are integers.

1. *ndmm* - Maximum number of cosine terms used to describe the temperature variation in the interconnect in the x direction.
2. *ndnn* - Maximum number of cosine terms used to describe the temperature variation in the interconnect in the y direction.
3. *nd3* - Maximum number of analysis layers in the interconnect.
4. *ndpp* - Maximum number of cosine terms, among all the chip/solder units, used to describe the temperature variation in the chip/solder unit in the x direction.
5. *ndqq* - Maximum number of cosine terms, among all the chip/solder units, used to describe the temperature variation in the chip/solder unit in the y direction.
6. *nd6* - Maximum number of chip/solder units.

Based on these six parameters, the correct size of the arrays in the global heat conduction Fortran codes is computed accordingly.

4.3.3 Output

There are three output files from the global thermal analysis. They are ***modelname*_gt_analysis.LOG**, ***modelname*_gt_analysis_err.LOG** and ***modelname*_gt_interface.dat**. These three files are created in the current working directory where the global thermal program is executed. The ***modelname*_gt_analysis.LOG** provides a log of the global thermal analysis and allows users to monitor the progress of the program. This is an ASCII file. In general, this information is useful for trouble-shooting only. The file ***modelname*_gt_analysis_err.LOG** is an error log file which contains error messages, if any, for the global thermal analysis. This file is also in ASCII format. Users should check for any error messages related to abnormal abort of the program in this error log. Appropriate response to these error messages as suggested in Section 7.6 of the REPAS Users' Manual [22] should be taken. The last output file, ***modelname*_gt_interface.dat**, contains the interface data for the subsequent

global thermal stress and local thermal analyses. This data file is also used by the global thermal post-processing routines for generating global temperature data for visualization. This output file is in binary format.

4.4 Global Heat Conduction Interface Procedure

The interface routines found in **$REPAS_HOME/gt/src/interface** allow the output from the global thermal analysis to be used by (i) the global thermal stress analysis, and (ii) the local thermal analysis. There is an initialization phase that is common to both global thermal stress and local thermal analyses. For each of these analyses, an initialization routine is called. The data in the file ***modelname*_gt_interface.dat** is read and organized into an internal data structure to facilitate the extraction of appropriate information for each analysis. This initialization is done only once for all calls to this procedure from each analysis.

For the global thermal stress analysis, the information required are the amplitudes in the polynomial (two terms) and cosine series representation of the temperature field in each layer of the interconnect and the chip/solder units. Other data required are the wavelengths of these cosine terms; the number of terms employed in the series representation of the temperature; and parameters related to the thermal properties. The procedure responds to queries from the global thermal stress analysis through the attribute manager SAM and returns all of this information on a layer by layer and chip by chip basis.

For the local thermal analysis, the information required are the temperatures at specified locations in the MCM. In this case, the local thermal procedure queries the interface procedure through SAM about the temperature at a given location. The interface procedure first determines whether the queried location is inside the interconnect or the chip/solder units. The layer number of the interconnect or the chip/solder unit number and the corresponding layer number are then determined so that the data stored in the internal database can be processed and the temperature at that location computed. The procedure then returns this temperature through SAM to the local thermal analysis.

Note that the interface program cannot link and execute by itself as there is no main routine. These routines can only be used for calling from the global thermal stress and local thermal programs. Users who need the temperature result at any point in the global structure should refer to the post-processing procedure which is described in Section 4.6 in this manual.

The parameters which decide the sizes of the arrays in the interface routines have to be the same as the ones used in the global thermal analysis for generating the interface data file ***modelname*_gt_interface.dat**. Otherwise, erroneous output or abnormal abort of the program will result. A check is made in the interface program to ascertain such a consistency. Error messages from the interface routines and appropriate responses are outlined in Section 7.6 of the REPAS Users' Manual [22].

4.4.1 Output from Global Heat Conduction Interface Procedure

The output file from the global heat conduction interface procedure is called **modelname_gt_interface_err.LOG**. It is in ASCII format and it resides in the current working directory. Error messages, diagnostics and trouble shooting suggestions for the global heat conduction interface routines are outlined in Section 7.6 of the REPAS Users' Manual [22].

4.5 Compiling and Linking

Makefiles are available for the global thermal analysis and interface programs. To compile the global thermal analysis routines, users can execute the Makefile in the directory **$REPAS_HOME/gt/src/analysis** by issuing the command *make*. Likewise, the routines for the global thermal interface procedure can be compiled by executing the Makefile in the directory **$REPAS_HOME/gt/src/analysis**. If new values for the array size parameters defined in the file **$REPAS_HOME/gt/src/analysis/Main/dimen.h** have to be used, all the object files for the global thermal analysis, the interface procedure, and the post-processing procedure (see Section 4.6) have to be removed and remade as described above. All of these object files can be removed by executing the command *make clean* in each of the directories where the Makefile resides.

Note that the object files for the global thermal interface procedure must stay in the standard directory **$REPAS_HOME/gt/src/interface** as other procedures linking with it will require it to be there.

4.6 Global Heat Conduction Post-Processing

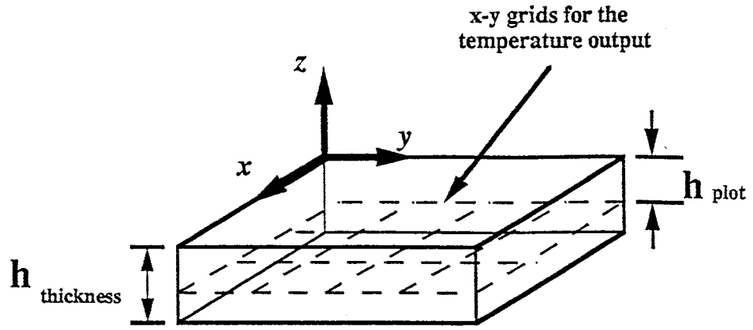
The post-processing procedure is a utility that enables users to output the global temperature on a two-dimensional grid in the x-y plane, either in the

interconnect or in a chip/solder unit, of the global structure for the purpose of visualization. Currently, the only format that this routine supports is the IBM Data Explorer [18] format. However, the source codes in the directory **$REPAS_HOME/gt/src/post** can be modified easily to suit an individual user's requirement for the output format.

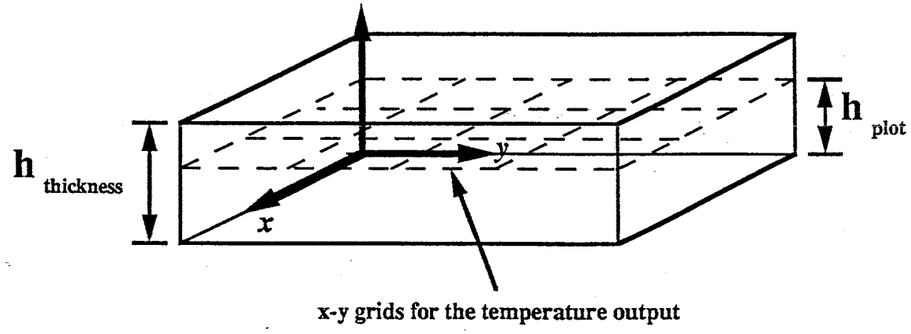
The post-processing procedure is almost the same as the interface procedure described in Section 4.4, except that a main program is available and therefore an executable binary can be created. The user is first prompted for the information on the location of the desired x-y plane and the number of grid points for the output temperature. The format of this is as follows:

1. *iwhere* (integer input) - " 0" if the temperature in the interconnect is required; "chip/solder unit number" if the temperature in the chip/solder unit is required.
2. *layer* (integer input) - layer number; if *iwhere* = "0", use the layer numbering convention for the layers in the global structure in the interconnect, assuming that the bottom layer is layer no. 1; if *iwhere* > "0", set *layer* = 1 for the chip and *layer* = 2 for the solder continuum.
3. *height* (real input, between zero and one) - the location of the x-y plane on which the temperature data is required. This input parameter is defined so that the user can select the location of this x-y plane as follows. Each layer of the global structure in either the interconnect or the chip/solder unit has its own local coordinates. The origins of these local coordinates are shown schematically in Figure 7. The input value *height* represents the ratio $h_{\text{plot}}/h_{\text{thickness}}$.
4. *nx* - number of grid points for the temperature in the x-direction.
5. *ny* - number of grid points for the temperature in the y-direction.

It is noted that the temperatures calculated for the grid points are not stored in an array, but rather, they are written to an output data file as soon as they are computed. Thus there is no restriction on the number of grid points to be used as far as the sizes of arrays in the post-processing program are concerned. However, an excessive number of grid points might require long CPU time. In order to visualize the temperature distribution, a grid of 50 by 50 in the interconnect and a grid of 15 by 15 in the chip/solder unit should be adequate. Of course, users can decide on the number of grid points that would give satisfactory resolution for visualizing the temperature distribution.



Chip or Solder Layer of Global Structure



Interconnect Layer of Global Structure

Figure 7. Coordinate systems for the chip/solder unit and interconnect structure

After these user-controlled input are read in, the post-processing program goes through an initialization phase to read in the data file ***modelname*_gt_interface.dat** and organize the data according to an internal database structure. This step is the same as the initialization phase for the interface procedure. Similar to the interface procedure, the parameters which decide the sizes of the arrays in the interface routines have to be the same as the ones used in the global thermal analysis for generating the interface data file ***modelname*_gt_interface.dat**. Otherwise, erroneous output or abnormal abort of the program will result. A check is also made in the post-processing program to ascertain such a consistency. Error messages from the post-processing routines and appropriate responses are the same as those for the interface procedure and they are outlined in Section 7.6 of the REPAS Users' Manual [22].

After the initialization phase is completed, the post-processing program processes all the grid points one by one. For each grid point, the program first calculates its coordinates, then computes the corresponding temperature, and then writes out the coordinates and the temperature into two separate output ASCII data files that are suitable for processing by Data Explorer. The output files are (i) ***modelname*_gt_post_xy.dat** for the coordinates; and (ii) ***modelname*_gt_post_t.dat** for the corresponding temperature values.

A Makefile is available for the post-processing routines in the directory **$REPAS_HOME/gt/src/post**; executing the UNIX command *make* will create the post-processing executable.

4.7 Replaceability of module with an equivalent module

The coefficient matrix obtained from the global thermal analysis is a dense symmetric matrix. After the coefficient matrix has been diagonally scaled, the linear algebraic equations are solved by using the symmetric equation solver *dsysv* from LAPACK. This linear algebra package is highly optimized and has been well tested in a variety of computers, from engineering workstations to supercomputers. A very important feature of this LAPACK equation solver for the global thermal analysis is the attention to *locality of reference* in the program. Also, BLAS level 3 routines are employed. These serve to drastically reduce the number of memory page faults and hence improves the turn around time of a global thermal analysis run. User prescribed control parameters to further fine-tuning the solver *dsysv* can be made. Reference to the LAPACK Users' Guide

[1] is recommended. If the user are like to replace this equation solver, the file **$REPAS_HOME/gt/src/analysis/Main/driver3.f** can be changed by replacing the statement

`call dsysv(uplo,n,nrhs,sx,lda,ipiv,xl,ldb,work,lwork,info)` with an equivalent call to another solver. Here, the important quantities are: *sx* - the coefficient matrix, *xl* - the right hand side vector, *lda* - the leading dimension of the coefficient matrix *sx*. The new equation solver should return the unknowns in the vector *xl*. The entries in the coefficient matrix *sx* could be destroyed during the equation solution phase.

5 Global Thermal Stress Analysis

5.1 Description and Algorithm

The global thermo-elastic stress analysis programs solve linear thermal stress problems through a semi-numerical procedure which is based on the variational approximation principle [27].

Global stress analysis uses a three-step process for solving the boundary value problems. In step 1, a chip layer and a solder continuum layer are assumed to cover the top of interconnect layers. The input temperature field in the idealized chip layer is derived from the global heat conduction analysis such that there is no thermal load in the inter-chip spacing. Based on this structure, displacements and tractions at the interfaces between the solder continuum and the interconnect are derived. Step 2 uses the tractions under the solder continuum from step 1 as traction boundaries at the top of interconnect layers and calculates displacements in the interconnect. Displacements in chips and the solder continuum are calculated by step 3 using displacements under the solder continuum from step 2 as boundary conditions.

Global thermal stress procedures also contain an interface procedure for the local stress analysis, details of which are described in Section 5.4 of this manual. A post-processing program is available for extracting the solution in various formats; currently, the only format supported is the IBM Data Explorer™[18] format. Details of the post-processing programs are given in Section 5.6 of this manual.

5.2 Source Code

The source codes for the analysis procedure are stored in the sub-directories **$REPAS_HOME/gs/src/analysis/step*** (where * is the number 1, 2 or 3). In sub-directory **$REPAS_HOME/epii/gs/src/analysis/step1**, there are 15 FORTRAN files, 1 C file and 16 other files for the common blocks for step 1. For step 2, there are 14 FORTRAN files and 16 common block files stored in **$REPAS_HOME/epii/gs/src/analysis/step2**. For step 3, there are 14 FORTRAN files, 1 C file and 16 common block files in sub-directory **$REPAS_HOME/epii/gs/src/analysis/step3**.

The source codes for the interface procedures and the post-processing procedure include 10 FORTRAN files and 1 file for common block. They are stored in sub-directory **$REPAS_HOME/gs/src/interface**. The two procedures share most of the source code because they have similar functions.

5.3 Input and Output

The necessary inputs for the global stress analysis procedure are derived from the global heat conduction interfaces, SAM, and program parameters set up in the code. Sections (5.3-1) to (5.3-3) describe these inputs.

5.3.1 Global Heat Conduction Interface Input

The file named ***modelname*_gt_interface.dat** is the interface data file from the global heat conduction analysis. Information from this file is obtained by the analysis procedure through SAM.

5.3.2 Input from SAM

In step 1 of the analysis procedure, the following parameters are derived through programs by either calling SAM directly, or modified by the programs after getting information from SAM. In step 2 and step 3, they are derived from reading the interface data files[1] between each step

1. *nlay* (integer) — Total number of layers. In step 1, *nlay* includes the virtual layers for chip and solder bump; in step 2, it does not.
2. *nchip* (integer) — Total number of chips.
3. *xoff, yoff, zoff* (reals) — Origin offset of MCM layers in x, y and z direction, respectively.
4. *ax, ay* (reals) — Width and length of the interconnect layers, respectively.
5. *z[i]* (real array) — z-coordinate at the bottom of interconnect layer number *i*. Note that *i* is $nlay + 1$ at the top of the interconnect layers where *nlay* is the total number of layers.
6. *idir* (integer) — Major direction of metal wires in an interconnect layer. $idir = 0$ if no wire; $idir = 1$ if wire along X direction; $idir = 2$ if wire along Y direction.

[1] i.e., files ***modelname*_gs_step1.dat** and ***modelname*_gs_step2.dat**

7. *ww, wp* (reals) — Average wire width and average center-to-center pitch of wires, respectively.
8. *arf[i,j]* (real array) — Effective linear thermal expansion coefficient of layer number *j*. $i = 1 \sim 3$ means linear expansion coefficient in x, y and z direction, respectively.
9. *c[i,j]* (real array) — Effective stiffness of layer number *j*. $i = 1 \sim 9$ means 9 independent stiffness constants for an orthotropic material.
10. *xc, yc, zc* (reals) — x, y, z coordinate of the center of a chip, respectively.
11. *chxyz[i,j]* (real array) — Maximum and minimum coordinate of the chip number *j*. $i = 1$ and 2 means minimum and maximum in x, respectively; $i = 3$ and 4 means minimum and maximum in y, respectively.
12. *ds* (real) — Average diameter of the solder bumps.
13. *ns* (integer) — Total number of solder bumps under all chips.
14. *spri[i,j]* (real array) — 3 by 3 spring constant matrix under the bottom of the substrate.
15. *nsp* (integer) — A parameter which is derived from spring constants. $nsp = 1$ means the spring constants only have diagonal terms so that the analysis is faster; $nsp = 0$ means the spring constants include off-diagonal terms, thereby increasing the amount of computation required.

5.3.3 Dimension Parameters

This section lists parameters which are related to maximum array dimensions in global stress analysis procedures. They are set up in the file *par.i* from step 1 to step 3. These arrays should be configured to handle the largest problem that must be analyzed. Note that all the following parameters are integers.

1. *ily* — Maximum number of layers allowed. *ily* should be at least equal to the total number of interconnect layers in the global structure plus two. Note: The substrate is counted as one interconnect layer (Default = 15).
2. *idmi* — Maximum number of Fourier series terms allowed for the solution functions for interconnect layers (Default = 20).
3. *idmc* — Maximum number of Fourier series terms allowed for solution functions of chips and solder continuum (Default = 15).
4. *itmi* — Maximum number of Fourier series terms allowed for the transformation functions for interconnect layers. In step 1, *itmi* should be greater than the number of terms in the Fourier series which expand temperature field in

the virtual chip and virtual solder continuum layer (Default = 50). In step 2, *itmi* should be the same as in step 1. In step 3, it should be the same as *ittm* (described below).

5. *itmc* — Maximum number of Fourier series terms allowed in the transformation functions for chips and solder continuum (Default = 30).
6. *ich* — Maximum number of chips allowed (Default = 25).
7. *idk* — Maximum dimension of the final matrix. *idk* is set to be equal to $4 * (idmi + 1) * (idmi + 1) + 15$ in step 1 and step 2, where *idmi* is described in (2) above (Default = 1779). In step 3, *idk* is set to be equal to $4 * (idmc + 1) * (idmc + 1) + 15$ (Default = 1039)
8. *ittm*, *ittn* — Maximum number of Fourier series terms allowed for the transformation function in x and y directions for interconnect layers in analysis step 2. *ittm* and *ittn* are suggested to be set at 2 to 3 times the value of *itmi* (Default = 120).

5.3.4 Output

Outputs of the analysis procedure include 5 files. The files named ***modelname*_gs_step2.dat** and ***modelname*_gs_step3.dat** are the interface data files between the analysis procedure and the interface and post-processing procedures, and they contain the coefficients of the Fourier series for the solution functions after analysis. The file named ***modelname*_gs_step*.log** is the log file with which users can monitor the progress of the program. The log file also contains CPU time and the average traction on boundaries where the approximate solutions do not satisfy boundary conditions exactly. If the analysis parameter *nave* is set to be zero, the traction average will not be calculated.

5.4 Interface Procedure

This section discusses the interface procedure between local stress analysis and global stress analysis.

Input files for the interface procedure are the interface data files ***modelname*_gs_step2.dat** and ***modelname*_step3.dat** which are derived from the analysis procedure. When executing, the procedure first reads these files and derives the necessary parameters[2]. This is done only once for all calls to this procedure

[2] Which are similar to those described in Section 5.3.2 of this manual

by SAM. The procedure responds to queries from local stress analysis (through SAM) about the solution at any given location by returning the displacements at that location.

Note that the interface program cannot link and execute by itself because there is no main routine. These routines can only be used for calling from other programs. Users who need stress and displacements results at any point in the global structure should refer to the post-processing procedure which is described in Section 5.6 of this manual.

The parameters which decide the maximum dimensions of interface routines are exactly the same as described in Section 5.3.3. Those parameters are set up in file **gs_com.i**. To avoid erroneous results, it is recommended that these parameters be set to the same values as they are in the analysis procedure.

5.5 Compiling and Linking

Makefiles are available for each step of the analysis procedure under the sub-directories **$REPAS_HOME/gs/src/analysis/step*** (* = 1, 2 or 3). Since the temperature field is a necessary input for step 1 and step 3, source codes in these steps must be linked with the global heat conduction interface. Also, step 1 and step 3 must be linked with SAM.

To compile and link the routines in each step, type the command *make* in each of the sub-directories.

Note that the object files for the interface procedure must stay in the standard directory **$REPAS_HOME/gs/src/interface** since other procedures linking with it will require it to be there.

5.6 Post-processing

The post-processing procedure is a utility that enables users to output the solution on a grid representing any portion of the global domain for the purpose of visualization. Currently, the only format that this routine supports is the IBM Data Explorer™[18] format. This section discusses modification of this routine to suit an individual user's requirement for the output format.

The post-processing procedure is almost the same as the interface procedure described in Section 5.4 except that a main routine is available and therefore it can be executed. This routine can be modified so that results can be output in a

format other than what is currently available. The routine initially reads in the results of the global stress analysis. After this a subroutine **gs_disp** is called with coordinates of the points of interest as input. **gs_disp** returns the displacement and stress components at these points are returned. According to individual needs, the locations at which this routine is queried for the solution may be varied. Also, the output statements may be modified to write the results out in desired format. The arguments to **gs_disp** are

1. *gx* (Input) — x coordinate of point of interest
2. *gy* (Input) — y coordinate of point of interest
3. *gz* (Input) — z coordinate of point of interest
4. *u(i)*, *i*=1,2,3 (Output) — displacements at (gx,gy,gz)
5. *s(i)*, *i*=1 to 6 (Output) — tractions at (gx,gy,gz)
6. *ierr* (Output) — *ierr* = 1 indicates that results were requested at a point outside the MCM. Otherwise *ierr* is 0.
7. *nar* (Output) — Layer number (in the global idealized model) that the point lies in.
8. *nch* (Output) — Chip number that the point lies in (0 if the point lies in the interconnect)

A makefile is available for the postprocessing routines under the sub-directory **$REPAS_HOME/gs/src/interface**; type **make** to create the post-processing executable, **gs_post**

Like the interface procedure, post-processing routines need the interface data files **modelname_gs_step2.dat** and **modelname_gs_step3.dat** as input. After reading these files and retrieving information through SAM, all the necessary parameters as described in Section 5.3.2 and 5.3.3 are found, and stresses and displacements at the input points are calculated.

The interactive input/output of the current post-processing routines are discussed in the **REPAS User's Manual** [22]. However, that may be altered to suit specific needs.

6 Local Heat Conduction Analysis

6.1 Overview/Description

The local thermal analysis module produces a parameterized description of the temperature field within a specified window by using a random-walk technique. Input to the program consists of geometric information, global-thermal results, and resolution parameters. Output consists of a data file containing the parameterized thermal field, and a log file.

Figure 8 shows a block diagram of the program submodules that make up the local-thermal analysis module. The CIF parser clips the geometric data to the local window and divides the window into a non-regular rectilinear grid of homogeneous cells — each cell consisting of only one type of material. The temperature at the window boundary is obtained from the global- thermal analysis module. A random-walk method is used to find the temperature at points on the surfaces of the internal cells as a function of the boundary condition. Results are written to a data file.

An interface procedure has been written to interpret the local thermal analysis results file and provide the local stress analysis module with pointwise temperature data through SAM. Descriptions of the input, output, and error conditions appear later in this chapter. See chapter 8 of REPAS User's Manual [22] for an overview of the theoretical operation of the local-thermal analysis package.

Figure 8 shows a block diagram of the program submodules that make up the local-thermal analysis module. User input is combined with global-thermal results to produce the detailed local temperature field, to be used by the local-stress analysis module. An MCM is described by the CIF file and model attributes, blocks (1) and (2). This information is combined and clipped to the local window (3) by the CIF parser (4) to form a full 3D description of the MCM within the local window. The local-thermal resolution parameters (5) control the accuracy of the results and affect the execution time of the program. The boundary conditions are evaluated using the global- thermal spatial resolution (6) and the global-thermal output routine (7). The local-thermal parameter extractor (8) uses the random-walk procedure and parameter fitting to produce a data file (9) containing the parameterized temperature distribution. The local-thermal output routine (10) is used to provide the local temperature to the local-stress analysis task.

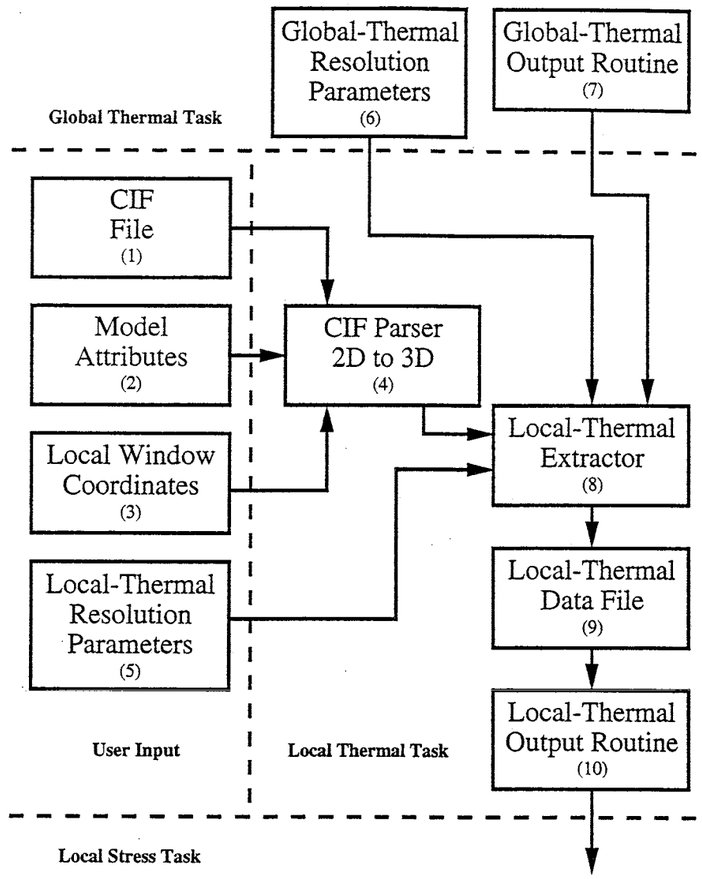


Figure 8. Local-thermal block diagram

The local-thermal parameter extractor is the body of the local-thermal analysis and can be divided into three subblocks, global thermal lookup table, random-walk method and parameter extractor, as shown in Figure 9.

The CIF parser provides three types of information: a grid of homogeneous cells, used by the parameter extractor; a list of rectilinear objects in the window,

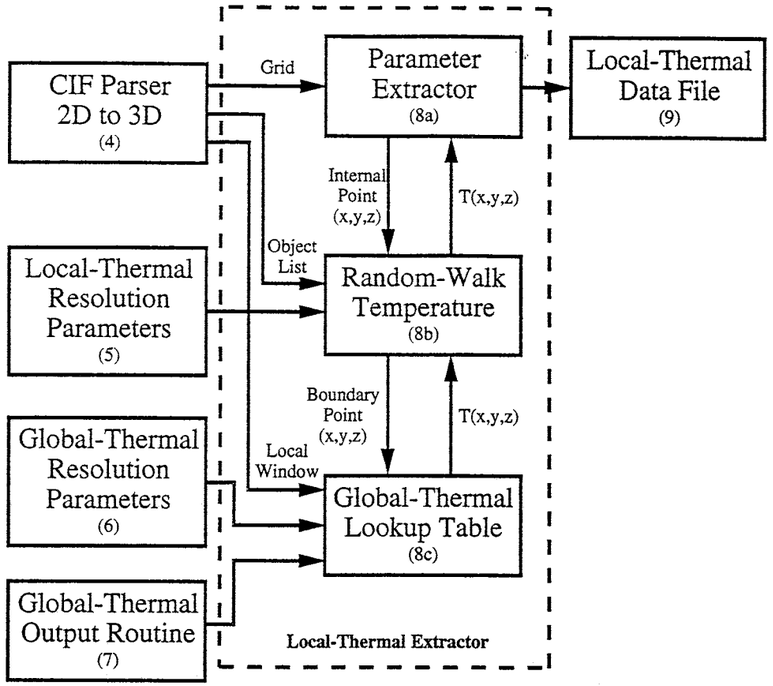


Figure 9. Details of the local-thermal extractor

used by the random-walk kernel; and the local window coordinates, used to generate lookup tables for the boundary temperature.

6.2 Input

1. Coordinates of local window: The coordinates of the local window are obtained from **SAM**.
2. Geometric data: The 2D structure of each layer of an MCM is obtained using a CIF file parser (see Section 7.9). The rest of the information about the layer (such as the thickness, material properties, etc.) are obtained through queries to SAM. The CIF parser routines are present in the file **$REPAS_HOME/lt/src/ltCIFparse.c**.

3. Spatial Resolution: The spatial resolution for the random-walk method is equivalent to a meshing size for finite-element or finite-difference methods, but does not significantly affect run time or accuracy and has no effect on memory requirements. This information is specified for the analysis in the file **epii.model** and obtained by local-thermal analysis by querying **SAM**.
4. Temperature resolution: This parameter too is set in **epii.model** and can be obtained from **SAM**.
5. Order of fit: The order of the fit is determined by the value of *nTerms* and is a programmable parameter. It can be changed by editing the definition of *nTerms* in the **$REPAS_HOME/lt/src/ltWalk.h** header file and recompiling and relinking all source code. A first-order fit (*nTerms*=1), however, has been found to be sufficient for representing the local temperature field.

6.3 Output

1. Standard Output: Error messages are the only output to the standard output file. These are described in Section 8.5 of **REPAS User's Manual** [22]
2. Log file: The local-thermal analysis module produces a log file named ***modelname*.lt.log**. A sample log file with suitable explanation is given in Section 8.3.2 of **REPAS User's Manual** [22]. The log file will also include any error messages generated.
3. Data file: The local-thermal analysis module produces a binary data file named ***modelname*.lt.dat**. The data file is read and interpreted by routines provided to the local-stress analysis module.

6.4 Compiling

A Makefile is present in the directory **$REPAS_HOME/lt/src** to compile the local thermal analysis program **lt**. Running the UNIX utility `make` will compile and link the program.

The interface routine to local stress analysis is present in the directory **$REPAS_HOME/lt/src/interface** and is called **ltTemp.c**. Like the analysis program, the interface procedure can be compiled by running the `make` utility in that directory. Note that the interface subroutine is compiled as an object file and can only be linked to other programs. It cannot be executed by itself.

7 Local Thermal Stress Analysis

7.1 Overview

Local thermo-elastic stress analysis of the MCM uses an adaptive finite element analysis to predict thermal stress within a small portion of the MCM. Broadly, it involves the generation of a geometric model of the local analysis domain, meshing it automatically, and performing an adaptive finite element analysis on the model utilizing the results of global stress and local heat conduction analyses [30][2]. The different software components of the local thermo-elastic analysis procedure and their interactions are shown in Figure 10. In the following sections, each of the software components of the local stress analysis are described.

7.2 Building a manifold geometric model of the MCM — gmodloc

7.2.1 Description and algorithm

Automatic mesh generation of the local analysis domain requires a complete geometric model of the domain. A complete and unique geometric model of the local analysis domain is non-manifold[3] due to the presence of multiple material regions. Therefore, a non-manifold data structure must be used to represent the local model. Although data structures exist for representing non-manifold models [32], most commercial geometric modelers do not support them. Therefore, procedures have been developed to build non-manifold models using the functionality of a manifold modeler [8][30]. Using these procedures, a non-manifold model can be built from a manifold representation which meets specific requirements. **gmodloc** is a program which builds such a manifold representation of the local analysis domain from information in the CIF file and geometric attribute information from the attribute manager.

gmodloc consists of a set of procedures to parse the CIF file (Section 7.9) and a second set of procedures to create the geometric model of the portion of the MCM inside a given window using the solid modeler, Parasolid™ [26].

[3] Non-manifold objects may have surfaces touching at a single point or an edge, have internal faces and be general combinations of wires, surfaces and solids [20][32][8].

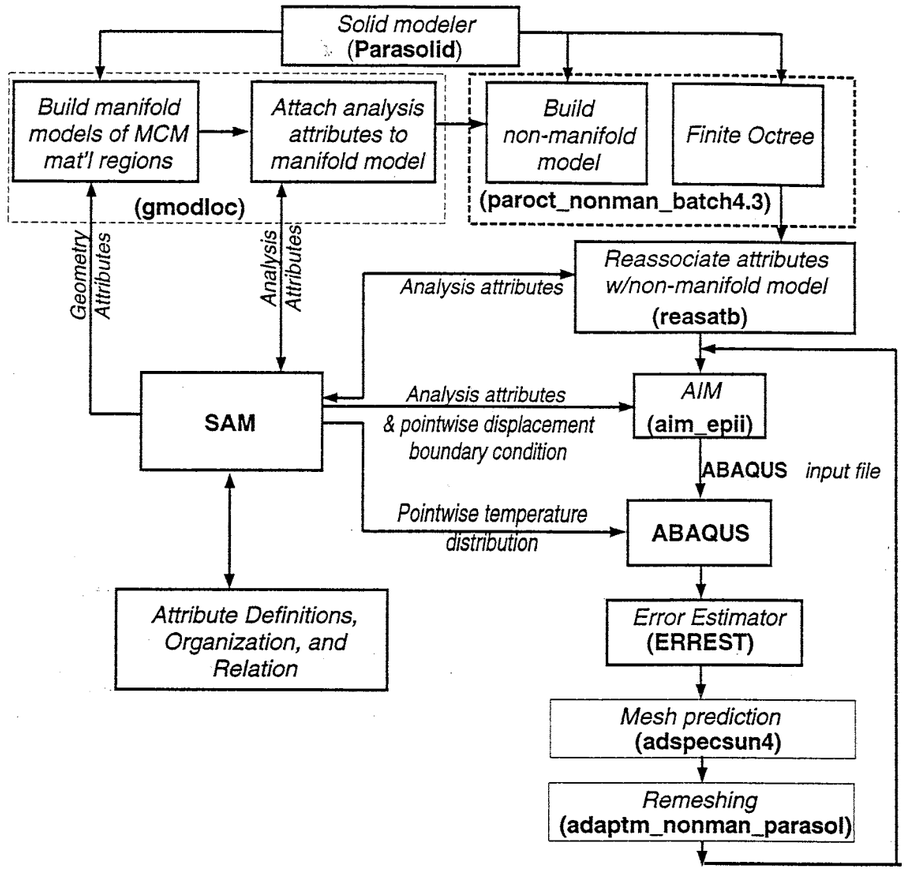


Figure 10. Organization of software for local stress analysis

The parsing routines are used to read the 2–D geometry of components (assumed to lie in a plane parallel to the X-Y plane) from the CIF files. Currently, the parser is only capable of reading circles or rectangles (boxes). Since each set of component descriptions is preceded by the name of the CIF layer they are in, the attribute manager is queried for the extents of the CIF layer in the Z direction. The minimum and maximum Z dimensions of the layer are used for

constructing box and cylinder primitives for all subsequent components read in until the next layer definition (not necessarily different from the previous one) is encountered. Before constructing the primitives with the geometric modeler, the components are clipped to the extents of the local window (obtained from the attribute manager). If the attribute manager does not return the extents of the local window, the extents in each coordinate direction are arbitrarily set to -500 and 500. An array of strings keeps track of the names of the different CIF layers read in. A corresponding dynamic array of integer arrays stores the identifiers of the primitives created in each CIF layer. As each primitive is created, the index of the array it should go into is determined by locating its layer name in the array of CIF layer names. When the CIF file is completely read, the primitives of each CIF layer are unioned. The model bodies surviving the union are all put in an assembly [26]. The result of this process is a manifold model of the local region containing vias, wires, solder bumps, metal layers and dielectric layers. At this stage the wires and vias in the model overlap the metal and dielectric layers. Therefore, the wires and vias are subtracted from the surrounding dielectric or metal to create voids into which they can fit in. A non-manifold model is later generated from this model by procedures in **paroct_nonman_batch4.3** (see Section 7.3.1).

Also, analysis attributes must be attached to the appropriate non-manifold model entities to be able to generate the finite element analysis input. However, the initial attribute specification is in terms of the CIF layers. Therefore, attributes of each CIF layer are associated with the material regions created from its definition. They are later reassociated with the non-manifold model entities utilizing specific knowledge of the non-manifold model building procedure. Only association of the boundary condition attributes is done after the mesh generation is completed and the non-manifold model written out. This is done so that the boundary faces may be determined from the non-manifold model using topological adjacencies alone avoiding less reliable geometric inquiries required if this were done with the manifold model (see Section 7.4). Model entities are associated with material attributes, initial conditions, temperature distribution from local heat conduction analysis results, and an element type attribute which facilitates the generation of solid 10–noded tetrahedral elements for the finite element analysis input file.

gmodloc also determines the faces of the local geometric model which are free faces of the interconnect. Distinguishing these faces from other boundary faces is important because free faces must have zero traction boundary condition on them

instead of a displacement boundary condition that is on other boundary faces of the local model. Boundary faces of a non-manifold model can be recognized by the fact that they are used by only one model region. Therefore, they are detected using only topological information after the non-manifold model is constructed. However, finding the free faces of the model requires comparison of the geometry of a local model face with the surfaces forming the boundary of the complete interconnect. The approach adopted here is to detect and tag these faces in **gmodloc** so that they can be recognized later when boundary conditions are being placed on the boundary faces in general (see Section 7.4).

A face of the local geometric model can be termed a free face if

1. the face does not overlap the boundary of the local window, or
2. a. the face does overlap the boundary of the local window,
   b. it is not the top face of a via, top face of a solder or the bottom face of a solder, and
   c. the face overlaps a vertical face or the top surface of the MCM, or any face of the chip.

See Figure 11 for an illustration of the definition of free faces.

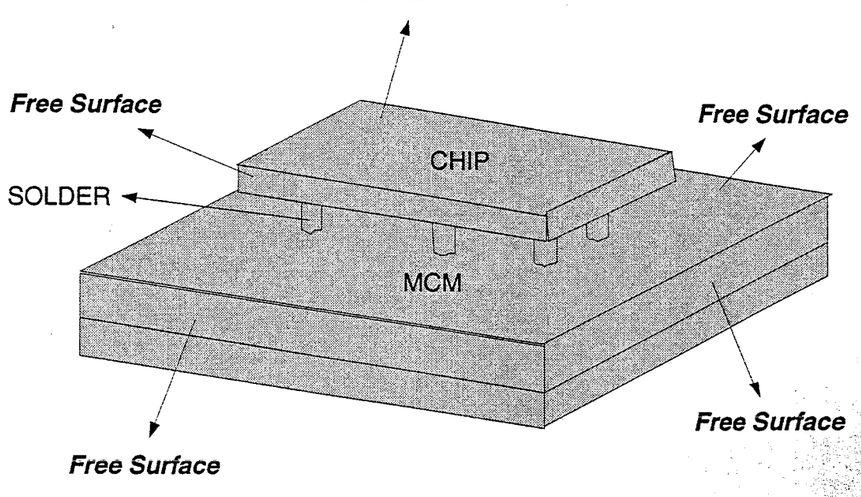


Figure 11. Simplified MCM showing free faces

For Parasolid, in particular, the model entity identifiers (tags) are not unique every time a model is retrieved. What *is* guaranteed is that tag numbers will be consistent whenever the exact sequence of modeling operations is followed in a modeler session. Therefore, it is important to ensure that the attributes are associated with the same set of tags that other programs of the local stress analysis reference. For this purpose, once the manifold model is built and the attributes are attached to certain model entity tags, the model is saved and the modeler stopped. A new modeler session is then started, the model reloaded and the attributes reassociated with the new entity identifiers. When this is done, any program which starts Parasolid and loads the model uses the same set of identifiers that the attributes are placed on.

Figure 12 and Figure 13 show pseudo code for the important procedures in **gmodloc**.

```
main () {
  Get_modelname
  Setup_attributes
  Get_local_window_coordinates_from_SAM
  Open_CIF_file
  Start_modeler

  while (next_character is_not EOF) and
        (next_character is_not 'E') {
  if (next_character is blank) Skip_blank_spaces
  switch (next_character) {
    case EOF: case 'E':
      goto done
    case 'B':
      if (layer_name is_not "dielectric_air") {
        Read_box_geometry
       if (box_is_in_local_window) {
         Clip_box_to_local_window
         Make_box_primitive
         Store_ID_of_box
       }
      }
   case 'L':
     Get_layer_name
     Get_thickness_of_layer_from_SAM
   case 'F':
     Read_round_flash_geometry
```

Figure 12. Algorithm to read CIF file and construct a geometric model of the local analysis domain (Continued) . . .

```
      If (flash_is_in_local_window) {
        clip_flash_to_local_window
        Make_cylinder_primitive
        Store_ID_of_cylinder
      }
    case '(':
      Ignore_comment
    default:
      Report_error
  }
 }
 done:
 Build_geometric_model /* from individual primitives */
 stop_modeller
} /* end main */
```

Figure 12. Algorithm to read CIF file and construct a geometric model of the local analysis domain

```
for (each_CIFlayer) {
  if (there_is_more_than_one_body_in_the_layer)
    Union_bodies_of_layer
    Retrieve_material_property_attributes_of_layer
    Associate_material_attributes_to_regions_of_layer
    Put_bodies_in_assembly
    }
  Save_model
  Stop_modeler
  Start_modeler /* new session */
  Retrieve_saved_model
  for (each_body_in_assembly) {
    Find_attributes_assoiated_with_body's_old_ID
    Reassociate_attributes_with_new_entity_number
  }
  Associate_initial_conditions_to_all_regions
  Associate_temperature_distribution_with_each_region
  Associate_element_attribute_to_each_region
  Identify_and_tag_free_faces_of_model
  Save_model
}
```

Figure 13. Building geometric model from individual primitives

7.2.2 Important modules/libraries

1. CIF file parsing routines — see Section 7.9.
2. Main program of **gmodloc** — The source code may be found in**$REPAS_HOME/ls/gmodloc/src/lsmain.c**.
3. **CIFBuildChip** — This routine combines the individual primitives built during the parsing of the CIF file into a model and places attributes on the material regions of the model. The source code for the module may be found in **$REPAS_HOME/ls/gmodloc/src/lsparasol.c**.
4. Parasolid libraries — see [26].
5. SAM libraries — see Section 2.

7.2.3 Input/Output

gmodloc and its modules obtain input from two sources — the CIF file and SAM. The CIF file describing the MCM is parsed to obtain the detailed 2D geometry of components of the MCM (see Section 7.9) while the attribute manager is queried for all other information. Given below is a list of the important queries made to the attribute manager to retrieve or associate attributes.

1. Scaling factor to be applied to dimensions read in from the CIF file (see Section 2).
2. Local window coordinates (see Section 2).
3. Layer's bottom and top plane locations (see Section 2).
4. Material properties of a layer (see Section 2).
5. Element type attribute (so that the right type of finite elements may later be generated) (see Section 2).
6. Temperature distribution attribute (see Section 2).
7. Initial condition attribute (see Section 2).

The output of **gmodloc** is a model file (***modelname*.xmt_txt**). Also, a modified set of attribute files is written out by **SAM** reflecting the changes to the attribute associations.

7.2.4 Compiling

The steps for compiling **gmodloc** are

1. *cd $REPAS_HOME/ls/gmodloc/script*
2. *gmake*

7.2.5 Software Limitations

Important limitations of the procedures of **gmodloc** are described below.

1. A maximum of 1000 *different* CIF layers can be handled by **gmodloc**. This can be easily increased by changing the parameter **MAXLAYERS**.
2. The clipping routines of **gmodloc** can only clip cylinders if the clipping plane is perpendicular to the axis of the cylinder. This is an important restriction since relaxing the restriction will imply that the clipped component is no longer a primitive.
3. The local window cannot enclose only chips, i.e., it has to at least include the solder bumps along with the chips. This constraint is present for the sake of efficiency; otherwise the bottom face of a chip can also be a free face and the procedure are have to check that.

7.2.6 Replaceability

Since **gmodloc** builds the manifold model of the interconnect region and places attributes on it, any other set of procedures which can do the same can replace it. The only constraint is that the solid modeler used to build the model and the one used for the automatic mesh generator queries be the same.

7.3 Non-manifold Modeling and Finite Octree Mesh Generation — paroct_nonman_batch

Finite Octree can broadly be viewed as consisting of two sets of procedures — a preprocessing procedure to construct a radial edge representation (non-manifold representation) from a manifold model and the meshing procedure which actually generates the mesh. This discussion is mainly concerned with the non-manifold modeling procedures. For a description of the Finite Octree mesh generator see [9],[10] and [28].

7.3.1 Description of non-manifold modeling procedures

As mentioned earlier, a set of procedures have been developed to create non-manifold models from manifold models [22][8]. Given a set of manifold objects that are disjoint and have matching (coincident) entities at interacting boundaries, and the coincident entity information, the procedures can create a radial edge representation [32]. In doing so, any one entity of a set of coincident

entities is recreated in the non-manifold representation and uses of the preserved entity are created for every member of the coincident entity set [8]. If the objects are overlapping, an operator is capable of making these objects disjoint. Since splitting of intersecting objects contains ambiguities, it must generally be resolved by additional input. The procedures can also create matching entities on interacting boundaries of objects and identify coincident entities through geometric inquiries to the modeler.

Since it is known that **gmodloc** creates a geometric model with disjoint objects, the process for building the non-manifold model of the interconnect from the manifold model can be tailored for this task. Thus, the flag for making objects disjoint (**MAKE_DISJOINT**) is set to **FALSE**, the flag for creating matching entities (**IMPRINT_BODIES**) is set to **TRUE** and the flag indicating that coincident entity information should be obtained from a file (**READ_FROM_FILE**) is set to **FALSE**. Matching entities are created on model objects touching each other if they do not already exist. Then coincident entity sets are identified using geometric inquiries to the modeler. The radial edge representation is then created from the manifold model and the coincident entity information.

7.3.2 Automatic mesh generation — Finite Octree

Finite Octree is an automatic mesh generator developed at SCOREC, RPI [28]. It is capable of generating tetrahedral meshes for general non-manifold 3–D domains. Geometric information about the model required for meshing is obtained from the solid modeler through a dynamic interface similar to the CAM-I interface specifications [5]. The mesh generator-solid modeler interface consists of a fixed set of geometric interrogation operators to query the modeler and only these operators need be rewritten for the mesh generator to be usable with a different solid modeler [9]. Currently Finite Octree interfaces exist to the Parasolid, ACIS™, CATIA™ and Shapes™ solid modelers.

7.3.3 Important modules/libraries

Given below is a list of operators (procedures) used in the non-manifold model creation.

1. **C_MakeDisjoint** — Make a given set of objects disjoint according to a given set of preferences.

2. **C_CreateMatchingEntys** — Create matching entities on boundaries of objects which touch each other.
3. **C_Get_CoEn_From_Model** — Deduce coincident entity relationships through geometric and topological inquiries to the modeler.

Refer [9] for details of the mesh generation operators.

7.3.4 Input/Output

The input to **paroct_nonman_batch** is a file containing the solid model. Since Parasolid is being used for this specific application, the expected input is a ***modelname*.xmt_txt** file containing the solid model generated using Parasolid.

There are a variety Finite Octree mesh control functions that can be controlled by user defined flags in two files, ***modelname*.atb** and **OCTREE.FLG**. The flags in the **OCTREE.FLG** file are meshing options (e.g. whether to create higher order nodes, whether to write a mesh file out, etc.) whereas the **.atb** file contains parameter values for element size control [7]. The absence of either of the files causes **Finite Octree** to assume default values for any missing data. For this application, an adaptive analysis is being performed which will eventually determine the mesh gradations in the analysis domain. Therefore, the initial mesh density is allowed to be the default coarse value. This is done by omitting the **.atb** file in the current directory. To obtain a finer mesh, the mesh refinement parameters must be changed in the ***modelname*.atb** file.

Higher order nodes are required for the elements and the mesh must be output for the analysis. This requires setting the "Create higher order nodes (HON)" flag and "Create mesh file" flag to 1 in the file **OCTREE.FLG**. An example of the **OCTREE.FLG** file is given in the **REPAS User's Manual** [22].

With the above options set, the following files written out by **Finite Octree** are useful for subsequent modules of local stress analysis.

1. ***modelname*.sms:** File containing a description of the mesh
2. ***modelname*.smd:** File containing the topology of the non-manifold model
3. ***modelname*.sav:** File containing required information for remeshing

7.3.5 Compiling

Not applicable.

7.3.6 Software Limitations

Refer [8] and [28].

7.3.7 Replaceability

Most of the local stress analysis modules are built around Finite Octree and the SCOREC mesh database. Replacing Finite Octree with a different mesh generator will not affect any module, except the remeshing procedures, provided it

1. supports the meshing of non-manifold models,
2. either interfaces to a non-manifold modeler (in which case **gmodloc** may have to be modified) or incorporates preprocessing procedures to build the non-manifold model from the manifold model,
3. supports classification of mesh entities on model entities [28],
4. can output the mesh and the model in the SCOREC mesh database format.

Currently the remeshing procedures, for a large part, use the mesh generation procedures; therefore, replacing the mesh generator will necessitate replacement of the local remeshing procedures. This may in turn affect the output of the mesh enrichment prediction procedure.

By comparison, interfacing Finite Octree with a different solid modeler for mesh generation is straightforward provided some basic functionality is satisfied by the modeler. As mentioned before, Finite Octree has a dynamic interface to the solid modeler in the form of a small fixed set of geometric inquiry operators. All the operators are keyed via the topological entities of vertex, edge and face. Approximately one half of the operators request basic topological associativities. The geometric interrogations used are limited to determining pointwise quantities such as the points of intersections and surface normals. These operators have been successfully developed for multiple geometric modeling systems.

Linking Finite Octree to another solid modeler requires the creation of these operators for the new geometric modeling system. To proceed in the development of these operators one should begin with a copy of reference [9][4] and the code for the operator set to at least one modeler.

Most geometric modeling systems provide basic sets of interrogation operators in terms of callable routines. More recent geometric modeling systems are built using a tool kit of such routines making it easy for applications such as finite

[4] Contact Professor Shephard at RPI for a copy of this document.

element mesh generation to access the needed functionality. In all cases the Finite Octree operators that request pointwise geometric information (line-surface intersection, surface normals at a point, etc.) must be constructed using the procedures provided by the solid modeler. If the solid modeler does not provide the basic operators required to construct these operators, it is not likely that a successful set of interface operators can be developed since the direct application of the functionalities of the geometric modeling system is necessary to ensure consistency of the geometric interrogations [29].

There are two approaches that have been used to provide the required topological adjacency operators. The most straight forward is to directly use operators provided by the geometric modeling system to provide this information. One drawback of this approach is that the geometric modeler's topological representation may be limited to 2–manifold representations. Even though Parasolid could perform the basic geometric operations needed to define the non-manifold geometries, neither provide a convenient method to store the required topological adjacencies. A second possible drawback is that the modeler does not explicitly store all next level adjacencies thus forcing searching, which is too time consuming for the number of interrogations required, to determine the information. If these drawbacks exist, the alternative approach is to preprocess the geometric model to create the needed topological adjacencies in a general non-manifold topological representation before the meshing process is initiated. This is the approach that was used for the Parasolid interface.

7.4 Reassociating attributes with the non-manifold model — reasatb

7.4.1 Description and Algorithm

In the process of constructing the non-manifold model of the local analysis domain, the manifold model may be augmented (model entities may get deleted or replaced or the model topology may be altered). Since, attributes have been associated with the entities of the original manifold model, they have to be reassociated with the augmented model entities. To facilitate the reassociation of attributes, augmentation information is written out by **paroct_nonman_batch4.3** in a file **augment.dat**. Each record in the file contains two fields; the first field being the parent entity, and the second field a child of the parent entity [8].

reasatb reads the file **augment.dat** and associates the attributes of the parent entities with the child entities.

reasatb also places boundary conditions on the boundary faces of the model. The procedure for determining the boundary faces of the model is based on the fact that for a non-manifold model only boundary faces point to one or no regions while interior faces point to two regions (which may be the same). To facilitate the identification of the boundary faces, the SCOREC mesh database and its operators is used [3]. The model is loaded into the mesh database from the ***modelname*.smd** file. Using topological inquiry operators, the boundary faces are identified and boundary conditions are associated with them. Traction free boundary faces are ignored in **reasatb**.

7.4.2 Important modules/libraries

The main routine of **reasatb** is in the source file **$REPAS_HOME/ls/reasatb/src/c_assatb.c**.

reasatb uses the SCOREC mesh database [3] and the SAM libraries (Section 2).

7.4.3 Input/Output

reasatb utilizes three sources of input described below

1. ***modelname*.smd** file — This file contains the non-manifold model topology which is then loaded into the SCOREC mesh database.
2. **augment.dat** file — This file contains augmentation information. Specifically it contains parent-child entity pairs in each field of the record.
3. **SAM** — The attribute manager **SAM** is queried for attribute information on model entities. The following is a list of information obtained from **SAM**.

   a. Boundary condition attribute for local stress.
   b. Material property attribute for material regions of the local stress model.
   c. A special character key attached to a face identifying it as a free face.

   The output of **reasatb** is a modified set of attribute files.

7.4.4 Compiling

Given below is the sequence of steps for compiling **reasatb**.

1. `cd $REPAS_HOME/ls/reasatb/script`

2. *gmake*

7.4.5 Software Limitations

None

7.4.6 Replaceability

reasatb can be replaced with any other module which identifies boundary faces of the non-manifold model of the local region based on topological queries to a non-manifold model database. With the present setup, the module must be capable of reconstructing the non-manifold model topology from information output by Finite Octree. Since **reasatb** only changes the associations of attributes with model entities, no other modules are affected when replaced with an equivalent module.

7.5 Analysis Interface Manager — AIM

7.5.1 Description and Algorithm

The analysis interface module is designed to perform two major tasks. First it transforms the mesh information from a standard file format to the format specific to an analysis input file. Next it retrieves all the appropriate attributes associated with the analysis case that is being run, properly associating them with the entities in the mesh and outputs them as appropriate to the analysis input file. The program itself is written in C++ and implements various classes that represent the model, the mesh, the attribute manager, and the attributes. In addition there is a class that is the "problem" itself. In this situation the problem is to write out an input file with the appropriate format with the proper information. The problem class is the one where all the work is done to output the information by querying it from the other objects in the program. A basic outline of how the program works is as follows: 1) load model 2) load mesh 3) go through each model entity in the geometric model and retrieve all attributes that are associated with that entity, make attribute objects and attach them to the appropriate model entity, and 4) write out the input file. The manner in which step 4 is done is totally dependent on the analysis code used.

7.5.2 Description of important sub-modules/subroutines/libraries

The most important routines in AIM relate to those that output the analysis input file. This is the only part of AIM that is designed to be easily changed to accommodate other analysis programs.

$REPAS_HOME/aim/prob/Abaqus.cc - This file contains all the routines that are specific to producing an ABAQUS input file from the mesh, model and attribute information.

7.5.3 Input/output

Input for mesh and attribute information is handled by routines that are described in the documentation for SAM (Section 2) and the SCOREC Mesh Database [3].

The only other input for **AIM** is the command file. This file is opened by the main routine and the first word, the problem type, is read in. Based on the type of the problem, an appropriate instance of the problem class is created. The input stream corresponding to the command file is then passed to the problem classes setup method. Here the rest of the information that is specific to the problem definition is read from the stream. The stream is then passed to the parent of the problem class (*FEProblem*) by calling its setup method. Here the appropriate information for the mesh, model and attribute manager is read in from the file and instances of the mesh, model and attribute manager objects are created.

7.5.4 Interface to other modules

The only module that **AIM** directly interfaces to is the attribute manager. There are six operators of the attribute manager that **AIM** calls directly. These operators are *AT_setup*, *attptr_C*, *AT_rtatype*, *AT_rtlabels*, *distrib_comp_C*, *AT_dstlistval*. These operators are described in the SAM documentation 2.

7.5.5 Compiling/linking with other modules

In order to properly access the results from the global analyses, AIM must be linked with the global stress interface routines.

7.5.6 Replaceablility of module with an equivalent module

This section will concentrate on the changes necessary to change the analysis package to some program other than ABAQUS. In order to do this, knowledge of C++ is required. In order for AIM to create an input file for a different analysis program, a class must be created for that analysis program. This section will describe the class that exists for creating input for ABAQUS. Here is the class definition for the ABAQUS program.

```
class Abaqus5 : public FEProblem{
 protected:
  ofstream * outfile;
 public:
  void setup(istream & infile);
  void doProblem(void);
  void doElement(Element & el, NodePList &nodes,
                 int type, MEntity &whatIn);
  void doBC(GEntity *ent, AttPList & attList);
  void doIC(GEntity *ent, AttPList & attList);
  void doProp(GEntity *ent, AttPList & attList);
  void doField(GEntity *ent, AttPArray & attList);
  void doLoad(Element *el, AttPArray & AttList,
              int side);
  void outNodes(void);
};
```

Each of these methods (except *outNodes*) are virtual functions that are first declared in one of the two parent classes of *Abaqus5: FEProblem* or *Problem*. The main program first creates an instance of the problem and then calls its *setup()* method. After everything is set up the main routine then calls the *doProblem* method of the problem and exits. The setup method is passed as input the input stream from the command file, here it reads the output file name and stores it (to be opened later). Then the *setup* method of the class *FEProblem* is called which reads in the rest of the command file and creates objects for the mesh, model and SAM. In the process of creating these objects and reading in the data, all of the attributes that are applicable to the current analysis case are associated with the proper model entities. After the *setup* method is called the *doProblem* method is called. It is this method that drives the actual process of creating the data file. For the *Abaqus5* class the *doProblem* method is as follows:

```
void Abaqus5::doProblem(void)
{
  int i;
  *outfile << "*HEADING\n";  // write out heading
  *outfile << "\n";
  outNodes();             // write out nodes
  outElem();
  gmodel->extProp();
  gmodel->extIC(analysisType);
  switch(analysisType){
  case stress:
  case thermal_stress:
    *outfile << "*STEP\n*STATIC\n";
    break;
  case heat_transfer:
    *outfile << "*STEP\n*HEAT TRANSFER, STEADY STATE\n";
    break;
  }

  // write out boundary conditions
  gmodel->extBC(analysisType);

  for(i=0; i<numElem; i++)          // write out loads
    elements[i]->extLoad(analysisType);
  gmodel->extField();
  switch(analysisType){
    case stress:
      *outfile << "*EL FILE\nE,S\nELEN\n*ENERGY FILE\n\
ALLEN\n*NODE FILE\nU\n";
      *outfile << "*EL PRINT\nCOORD,E\nCOORD,S\n\
*ENERGY PRINT\nALLEN\n";
      *outfile << "*NODE PRINT\nCOORD,U\n";
      break;
    case heat_transfer:
      *outfile << "*EL FILE\nHFL\n*ENERGY FILE\n\
ALLEN\n*NODE FILE\nNT\n";
      *outfile << "*EL PRINT\n*ENERGY PRINT\n\
ALLEN\n*NODE PRINT\nCOORD,NT\n";
      break;
    case thermal_stress:
      *outfile << "*EL FILE\nE,S\nELEN\n\
*ENERGY FILE\nALLEN\n";
      *outfile << "*NODE FILE\nU,NT\n*EL PRINT\n\
COORD,E\nCOORD,S\n";
      *outfile << "*ENERGY PRINT\nALLEN\n\
*NODE PRINT\nCOORD,U,NT\n";
    }
    *outfile << "*END STEP\n";
```

```
}
```

A general outline of the procedure to write out the analysis input file is as follows:

1. A heading is written to the output file.
2. `outNodes()` method is called which writes out the nodal locations to the output file.
3. `outElem()` method is called which writes out the element connectivities to the output file. The next few steps utilize a global variable called "`gmodel`" that stores the instance of the geometric model.
4. `gmodel->extProp()` is called. This routine loops through all the regions in the geometric model and for each region that has a material property definition calls the `doProp` method of the problem.
5. `gmodel->extIC()` is called which is similar to `extProp()` but acts on attributes that define initial conditions and calls the problem method `doIC()`.
6. `gmodel->extBC()` is called which is similar to `extProp()` but acts on attributes that define boundary conditions and calls that problem method `doBC()`.
7. For each element in the problem, the `extLoad()` method is called which checks if the element has a distributed load on one of it's faces. If it does it calls the problem method `doLoad` for that face.
8. `gmodel->extField` is called which is, again, similar to `extProp()` but acts on fields (such as temperature) that are defined over the body. It called the problem method `doField()`.

The end of the file is then written to the output stream. Various behavior may be modified by creating a subclass of `Abaqus5` and overloading the appropriate routine. For an example of this, see the files **$REPAS_HOME/aim/prob/AbaqusEpii.cc** and **$REPAS_HOME/aim/prob/AbaqusEpii.h** where some of these methods are overloaded to change the output from a general ABAQUS input file to one specific to this project.

7.6 Finite Element Analysis — ABAQUS

7.6.1 Description

The finite element analysis tool used in the local thermal stress analysis is ABAQUS (v 5.2.1). ABAQUS is chosen as the finite element analysis engine

since it provides convenient user extensions for interface to local heat conduction analyses results. This section gives a brief introduction to the use of the software for the local stress task. Programmers should refer to ABAQUS/Standard Users' Manual I, II, ABAQUS/Theory Manual and ABAQUS /Post [15, 16, 14, 13] for more detailed information.

7.6.2 Description of User Subroutines

The user subroutine developed for local thermal stress analysis follows the format requirements of ABAQUS (see [15]). In this section the user subroutine for prescribing temperature distribution on nodes. Also described is a user subroutine for prescribing displacement boundary conditions, although this information is currently written into the ABAQUS input file directly.

`utemp_` is a function used to prescribe temperatures on the nodes. It obtains nodal temperature from local heat conduction analysis through SAM. This routine is called for each node listed under keyword `*TEMPERATURE, USER`. Instead of the FORTRAN subroutine *UTEMP*, REPAS uses an equivalent C subroutine `utemp_` and has the following format

```
/* User subroutine to answer ABAQUS queries
   about nodal temperature
*/
#include <stdio.h>

void utemp_(double *temp, int *nsecpt, int *kstep, int *kinc,
            double *time, int *node, double *coords)
{
  static int first=0;
  char *modelName;
  double tempValue[1];
  int valueSpecs[1], numRet;


  if (first == 0)
  {  /* set up attribute database on first call */
    AT_gtmoname(&modelName);
    AT_setup(modelName);
    first = 1;
  }

  tempValue[0] = 0;
```

```
  /* get temperature from local thermal */
  AT_gtvalocal("ls", NULL, coords, "temperature_distribution",
               "temperature", 1, tempValue, valueSpecs, &numRet);
  if (numRet == 0) {
    fprintf(stderr,"\n\nNo temperature returned to \
    ABAQUS user subroutine.\n");
    exit(-1);
  }
  temp[0] = tempValue[0];
}
```

The following is a list of arguments for *utemp_* with an explanation of their meaning.

1. *temp* (Output) — Temperature array of size *NSECPT*
2. *nsecpt* (Input) — Pointer to size of *temp* array (1 for solid elements)
3. *kstep* (Input) — Pointer to current step number
4. *kinc* (Input) — Pointer to current increment
5. *time* (Input) — Pointer to current time
6. *node* (Input) — Pointer to node number
7. *coords* (Input) — Pointer to coordinates of *node*

The body of the subroutine includes the definition of *temp* in a user defined manner; in this case, this is simply a call to SAM asking for the temperature at the coordinates of the node. SAM, in turn, queries the local thermal interface routine *ltTemp* for the information (see Section 6).

Currently, displacement boundary conditions are written to the ABAQUS input file by AIM. The following user subroutine, *disp*, can be used instead to prescribe displacements on nodes. *disp* is called for each degree of freedom of the nodes listed under keyword **BOUNDARY, USER SUB*.

```
      SUBROUTINE DISP(U,KSTEP,KINC,TIME,NODE,JDOF)
      INCLUDE 'ABA_PARAM.INC'
      DIMENSION U(3),TIME(2)
      DIMENSION NODE(2,*), COORD(3)
      COMMON /CNS/IDUM,INCRD

      CALL ACOPY(NODE(INCRD-1), COORD, NDIM)


C     user coding to define U
C     For REPAS, SAM operator to get displacement
C     from global thermal stress module is called


      RETURN
      END
```

The following is a list of arguments for *DISP* with an explanation of their meaning.

1. *U* (Output) — Displacement array of size 3
2. *KSTEP* (Input) — Current step number
3. *KINC* (Input) — Current increment
4. *TIME* (Input) — Current time
5. *NODE* (Input) — Node number
6. *JDOF* (Input) — Degree of freedom

7.6.3 Input/output

ABAQUS/Standard User's II [16] has a summary of input, output, and intermediate files either needed or generated during an ABAQUS run. Their format is described in detail in various chapters of the ABAQUS User's Manual [15].

7.6.4 Compiling and Linking

Standard ABAQUS can be run without any compilation. However, the use of user subroutines which interface with SAM requires that a customized ABAQUS executable be created for local stress analysis. This requires using an ABAQUS command file template to create and run temporary executable each time a step

in the adaptive analysis process is executed or to run a copy of the customized executable kept permanently on disk. Currently, the former approach is adopted. Refer Section 9.11.6 of REPAS User's Manual [22].

7.6.5 Software Limitations

Programmers are directed to references [15][16][14][13].

7.6.6 Replacement of Analysis Module

ABAQUS may be replaced in the local stress analysis with any other finite element analysis software capable of doing a linear elastic and nonlinear elasto-plactic thermal stress analysis. Only two procedures will be affected by this change — **AIM**, since it produces the input file for the analysis software and **ERREST**, since it gets the finite element solution from ABAQUS. Refer Section 7.5 on incorporating the capability to interface to different finite element analysis software. If the analysis software does not support user extensions, **AIM** may be modified to write out the temperature at each node (or some other mechanism found to provide this information to the analysis software). Section 7.7 discusses modifying **ERREST** to use a finite element solution from different analysis software.

7.7 ERREST — *A Posteriori* Error Estimation

7.7.1 Description and Algorithm

ERREST performs *a posteriori* error estimation for a three dimensional thermal stress analysis. The error in the energy norm is calculated which serves as a basis for further finite element adaptive refinement. A detailed discussion of theory can be found in REPAS technical documentation [23][2].

7.7.2 Description of Important Subroutines

1. **errest.f**: main control program for *a posteriori* error estimation procedure.
2. **rtisol.f**: retrieves solution at integration points.
3. **rtmatp.f**: retrieves material properties for an element.
4. **rtprob.f**: retrieves analysis type.
5. **rtsoln.f**: retrieves the strain energy of the finite element mesh.

6. **tetr15.f**: provides Gauss quadrature information for 3–D elements.
7. **filread.f**: interface routine to ABAQUS result file ***modelname*.fil** which reads it and stores appropriate information needed for error estimation.
8. **interr.f**: error estimation procedure performed in this routine on an element by element basis.
9. **setmat.f**: sets up material properties for error estimation using the ABAQUS input datafile ***modelname*.inp**.
10. **rtmats.f**: sets up material classifications of finite elements using the SCOREC mesh database [3].
11. **shapes.f**: provides hierarchical shape functions of finite elements.
12. **refnee.f**: computes required element sizes based on interior error norm information.
13. **wradp.f**: writes out desired element sizes for mesh enrichment prediction.

7.7.3 Input/Output

The error estimation and element size prediction procedures of **ERREST** are interfaced to the finite element analysis through a set of generic retrieval operators (all functions named *rt*****, e.g., *rtmats*). The error estimation procedures query these operators for the finite element solution and these operators retrieve the results of the finite element analysis. Thus all other procedures in the program are not affected if these retrieval operators are modified.

Currently, these operators retrieve the required information from common blocks which are initialized by the routine *filread*. *filread* reads the results of the ABAQUS analysis and loads the necessary data into common blocks in the program.

If ABAQUS is replaced by another software, then the routines that will be affected, in general, will be some or all of the retrieval operators. The changes that must be made to the retrieval operators will depend on the mechanism that the new software provides to recover the solution. If the current mechanism of using *filread* to initialize common block arrays is retained, this routine too will have to be changed.

7.7.4 Compiling and Linking

A makefile is provided in **$REPAS_HOME/ls/errest/src** to automatically compile and link the error estimation routines and various libraries. To compile

and link the program, type *make* in this directory.

7.7.5 Software Limitations

The error estimation routines developed are for three dimensional 10–noded tetrahedral elements. It can be used for *a posteriori* computation of the error of thermal stress finite element analysis, as well as three dimensional heat conduction analysis and mechanical stress analysis.

7.7.6 Replacement of Error Estimation Procedure

The main concern when replacing the error estimation procedures is to ensure that the input/output procedures maintain compatibility with the finite element analysis software and the mesh enrichment prediction procedure. On the input side, an interface must exist to extract the finite element solution and provide it to the error estimator. Also, depending on the type of error estimation procedure used, the procedure generating the analysis input file may have to be modified, so that the analysis software provides all the necessary solution information. On the output side, the error estimation procedure must be able to provide desired element size information to the mesh enrichment prediction module.

7.8 Adaptive Local Remeshing — adaptm_nonman_parasol

7.8.1 Description

adaptm_nonman_parasol is a set of local remeshing procedures which modify an existing Finite Octree representation of the model based on element sizes specified at a set of spatial locations. Based on these refinement points, the procedures refine or coarsen portions of the tree and then retriangulate modified portions of the tree. The program incorporates preprocessing procedures used to generate the non-manifold representation of the local domain (Section 7.3.1).

7.8.2 Input and Output

Since the local remeshing procedures are based on Finite Octree, they utilize the information of the Finite Octree representation of the model along with refinement information to effect changes in portions of the mesh. Thus the procedures require the octree as well as the mesh before collapsing or smoothing. The

procedures also utilize other inputs required by the mesh generation procedures, namely, the geometric model file and the options file.

Currently, refinement is requested by other procedures in terms of spatially located refinement points. This may be modified to incorporate some other mechanism if the interface to the mesh enrichment prediction program is suitably changed.

7.8.3 Compiling and Linking

A link script is available in the directory **$REPAS_HOME/scripts** and is called **adaptm_script**. Run this script to create the executable **adaptm_nonman_parasol** in the directory **$REPAS_HOME/bin**. Note that Parasolid libraries **parasolid.lib**, **frustum.lib** and **fg.lib** must be available for linking this program and that the link script must be edited to indicate the path to these libraries.

7.8.4 Replacing the local remeshing module

In the current implementation, the local remeshing procedures use the octree representation of the model, and the mesh generated by Finite Octree together with much of Finite Octree functionality to enrich the mesh. As discussed in Section 7.3, this makes it necessary to replace the local remesher if the mesh generator is replaced.

However, the local remeshing procedure itself is not constrained to use the same algorithm as the mesh generator and may be replaced with an equivalent remeshing procedure. Note that the functionality of this procedure is subject to the same requirements as that of the mesh generator (see Section 7.3). In addition, an interface between the mesh enrichment predictor must be established if the existing one cannot be used.

7.9 CIF Parser[5]

The CIF parser is a set of routines that reads a CIF file by looking at the first character in each line and calling a suitable routine to parse the entire line (refer [21] for a description of the CIF format). The CIF parser recognizes commands

[5] This code has been adapted from the MAGIC design system, Copyright (C) 1984, Regents of the University of California.

specifying a layer name or the 2–D description of the geometry of a component in the last encountered layer. Comments and other commands are ignored. The parser can parse the "Box" and "Round flash" geometries. Each geometry parsing routine takes as input a scaling factor to multiply the CIF dimensions read.

The important higher level routines in the CIF parser are

1. **CIFOpen** — Opens the file ***modelname*.cif**. It gets the ***modelname*** from SAM.
2. **CIFParseDouble** — Parses a real number.
3. **CIFParseName** — Parses a string.
4. **CIFParseInteger** — Parses a positive integer.
5. **CIFParseSInteger** — Parses a signed integer.
6. **CIFParsePoint** — Parses a pair of real numbers and assign to special structure for points with x and y coordinates as fields.
7. **CIFParseLayer** — Parses layer changes in the CIF file and return the new layer name. This is indicated in the CIF file by a line starting with an 'L' and followed by a string. All geometry descriptors occurring after a layer command is encountered are taken to belong to that layer until a new layer is encountered. The routine to parse CIF layers also keeps track of the names of unique layers that have occurred in the CIF file.
8. **CIFParseBox** — Parses a box command starting with a 'B' in the CIF file.
9. **CIFParseFlash** — Parses a round flash (circle) command starting with an 'R'.
10. **CIFSkipToSemi** — Skips everything up to the next semicolon.
11. **CIFSkipSemi** — Skips a semicolon.

The CIF parser can be tailored to parse other commands such as wires and polygons by adding new functions. It may also be replaced with other CIF parsers.

8 MCM Router and Electromagnetic Analysis

8.1 Overview

The multichip module (MCM) is routed using the RPI-MCM router (Donlan Router [6]), which is a high speed line probe router. The router uses Hightower[17] routing coupled with AI rules to route an MCM. It is a very versatile router and can incorporate MCM blockout regions too. The output of the router is in CIF (Caltech Intermediate Format) and can be used directly by other modules. The input to this router is a wafer level netlist which describes the interchip connections and the wafer configuration along with the routing blockout regions. The electromagnetic analysis module uses MagiCAD [24], a tool developed by the Special Purpose Processor Development Group (SPPDG) at Mayo Foundation. MagiCAD can help the user model transmission line networks on an MCM and simulate them in both time and frequency domain. The electromagnetic tool suite in MagiCAD consists of schematic compilers, netlist compilers, netlist simulators, and transmission line cross-section extractors. It uses boundary element method (BEM) to extract the parameters of a transmission line from its cross-section. The RPI-MCM router is available from the RPI CIE[6] and the MagiCAD can be requested from Mayo Foundation. A short program has been developed to extract the information about the wires in the final route. The program is called as "extractor" which is an awk script executable on any system supporting awk language and is attached in the appendix.

8.2 MCM Routing Overview

A block level diagram of the MCM routing procedure is shown in Figure 14. User input comes in the form of interchip connections as a net list and the definitions of the location of the chip pads. The user also supplies information about the configuration of the MCM such as MCM size, wire width and pitch, and output pin locations.

The output of the router can be stored in several ways. The desired format is CIF as the rest of the analysis tools use the same database. The CIF output of the router consists of all the nets with their horizontal and vertical segments in

[6] Center for Integrated Electronics

order. This CIF output is supplied to the extractor, which can extract the length of any net from this CIF file and supplies it to MagiCAD.

8.3 Electromagnetic Analysis Overview

Electromagnetic analysis procedure is shown in the block diagram in Figure 15. After the completion of MCM routing, the user extracts the information about the desired net, whose performance is to be checked, and inputs it using the graphical editor of MagiCAD. It also takes the information from the supplemental file about the 3-D structure of MCM. Using this information a cross-section of the transmission line is drawn and transmission line parameters are extracted from it. The results of this extraction are supplied to the network simulation step to simulate the time and frequency domain characteristics of the net. The block diagram of the network simulation is shown in Figure 16.

8.4 Extractor

Extractor is an awk program developed to parse a CIF file and generate a list of all the nets with their lengths in an ascending order. A listing of extractor is shown in Figure 17 and it can be run on any UNIX system. This program needs a CIF file as its input in the current directory. The name of the CIF file is currently hard coded and should be named as **input.cif**. The program generates an output file named as **length.net**. The program can be invoked by typing *extractor* at the prompt.

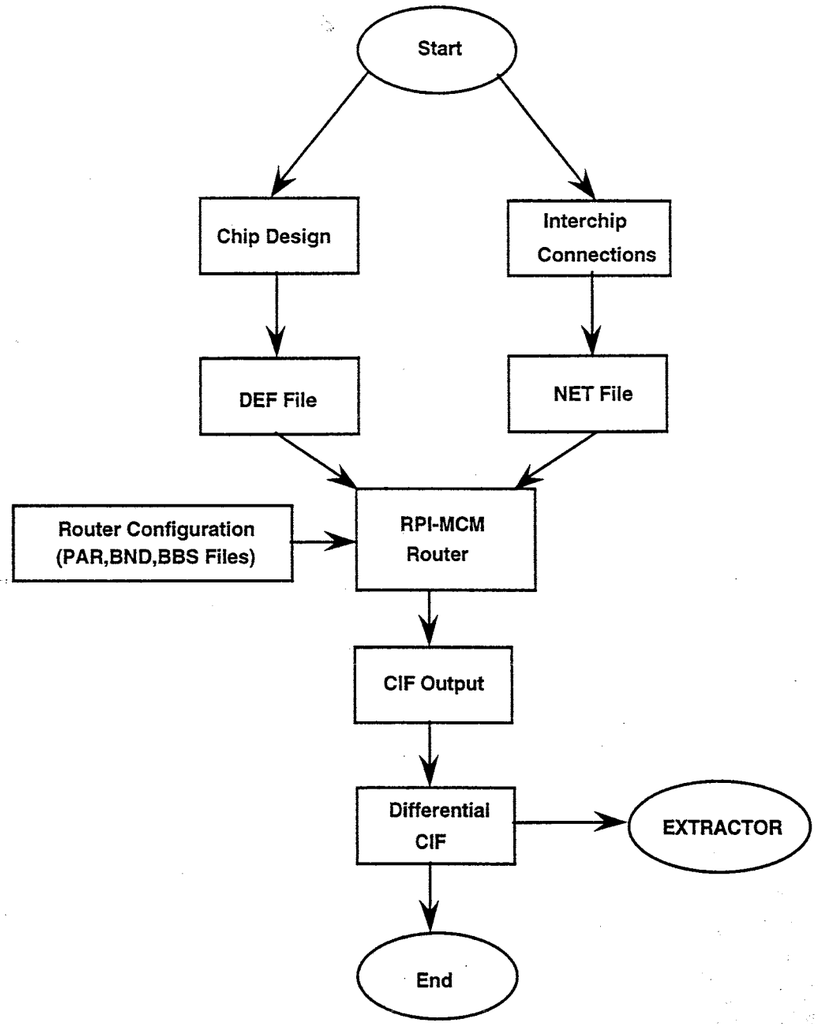


Figure 14. Block diagram of the MCM routing method.

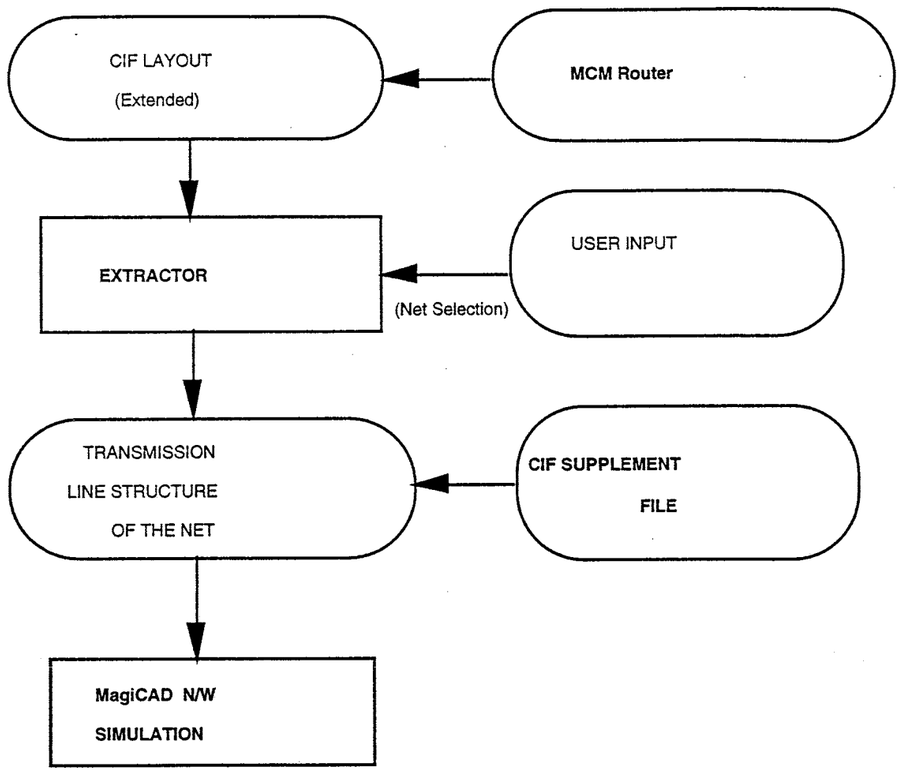


Figure 15. Flow chart of EM analysis with MagiCAD.

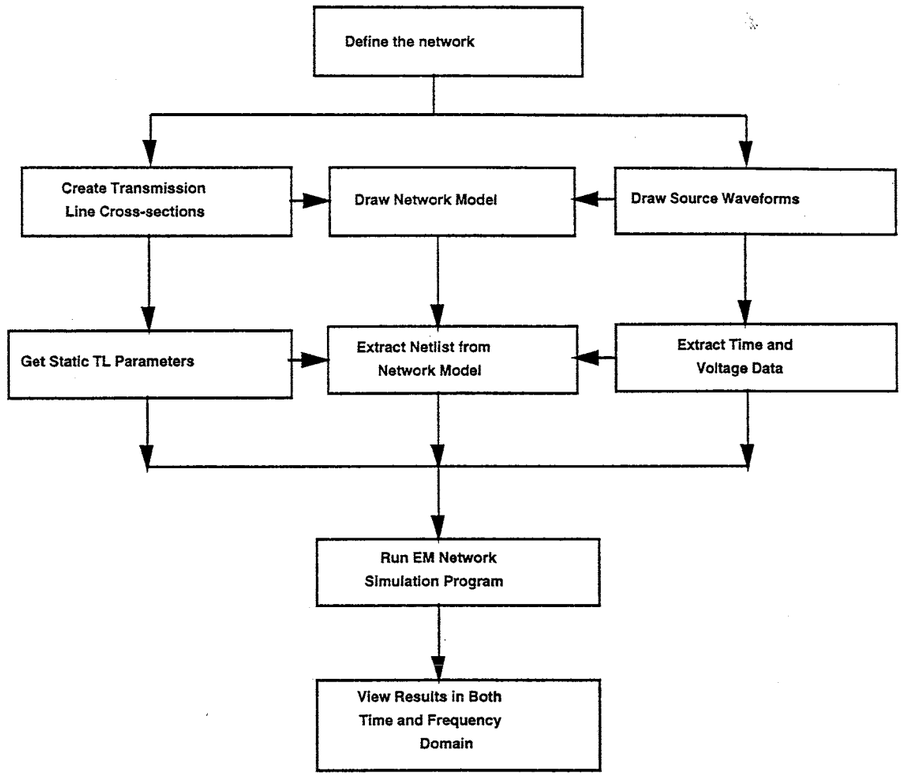


Figure 16. Block diagram of MagiCAD network simulation.

```
#! /bin/sh
rm router.cif
rm mcm.dat
nawk ' BEGIN { while (getline < "input.cif" > 0)
                {  if ($0 ~ /^\(/)
                   tot_nets++
                   # print $0
                }  # End of While
               close("input.cif")
#-----------------------------------------------------#
     # Variable list
     # tot_segs      -- Total cif segments
     # tot_nets      -- Total MCM nets
     # cum_h_segs    -- Cumulative horizontal segs
     # cum_v_segs    -- Cumulative vertical segs
     # cum_h_length  -- Cumulative horizontal length
     # cum_v_length  -- Cumulative vertical length
     # cum_length    -- Overall cumulative length
     # longest_v_seg -- Longest vertical seg
     # longest_h_seg -- Longest horizontal seg
     # longest_net   -- Longest net
     # old_longest   -- Temporary longest net
     # old_long_v    -- Temporary longest vertical seg
     # old_long_h    -- Temporary longest horizontal seg
     # curr_h_segs   -- Current horizontal seg count
     # curr_v_segs   -- Current vertical seg count
     # net_name      -- Current net name being processed
#_____________________________________________________#

     print "TOT_NETS: " tot_nets > "router.cif"
     first = 1
     old_tally = 0
     while(getline < "input.cif" > 0)
     {  if ($0 ~ /^\(/)
         { if (!first)
            { print "(NET_NAME " $0 >> "router.cif"
              # print "processing net " net_name
              print "H " curr_h_segs >> "router.cif"
              net_tally = 0
              one_tally[net_name] = 0
              for (k = 1; k <= curr_h_segs; k++)
                { print "B " hx_cif[k],hy_cif[k],hxcnt_cif[k],
                   hycnt_cif[k] >> "router.cif"
                  net_tally = net_tally + hx[k]
                  one_tally[net_name] = one_tally[net_name] + hx[k]
```

Figure 17. Listing of Extractor (program also available as **$REPAS_HOME/em/extractor**) (Continued) . . .

```
    }
  print "V " curr_v_segs >> "router.cif"
  for (k = 1; k <= curr_v_segs; k++)
    { print "B " vx_cif[k],vy_cif[k],vxcnt_cif[k],
      vycnt_cif[k] >> "router.cif"
      net_tally = net_tally + vy[k]
      one_tally[net_name] = one_tally[net_name] + vy[k]
    }
  if (net_tally > old_tally)
    { old_tally = net_tally
      old_name = ""
      old_name = net_name
    }
  h_flag = 0
  v_flag = 0
  h_index = 0
  v_index = 0
  net_name = ""
  net_name = substr($1,2,8)
 }
else
 { net_name = substr($1,2,8)
   # print "processing net " net_name
   h_flag = 0
   v_flag = 0
   h_index = 0
   v_index = 0
   first = 0
  }
 }
if ($2 == "W2;")
  { h_flag = 1
    v_flag = 0
    h_index = 0
    curr_h_segs = 0
  }
if ($2 == "W1;")
  { h_flag = 0
    v_flag = 1
    v_index = 0
    curr_v_segs = 0
  }
if ($1 == "B")
  { if (h_flag)
     { h_index++
       hx[h_index] = ($2/100)
```

Figure 17. Listing of Extractor (program also available as **$REPAS_HOME/em/extractor**) (Continued) . . .

```
      hx_cif[h_index] = $2
      hy[h_index] = ($3/100)
      hy_cif[h_index] = $3
      hxcnt[h_index] = ($4/100)
      hxcnt_cif[h_index] = $4
      hycnt[h_index] = ($5/100)
      hycnt_cif[h_index] = $5
      tot_segs++
      cum_h_length = cum_h_length + ($2/100)
      curr_h_segs++
    }
   else
    { v_index++
      vx[v_index] = ($2/100)
      vx_cif[v_index] = $2
      vy[v_index] = ($3/100)
      vy_cif[v_index] = $3
      vxcnt[v_index] = ($4/100)
      vxcnt_cif[v_index] = $4
      vycnt[v_index] = ($5/100)
      vycnt_cif[v_index] = $5
      tot_segs++
      cum_v_length = cum_v_length + ($3/100)
      curr_v_segs++
    }
}
print "(NET_NAME " $0 >> "router.cif"
print "H " curr_h_segs >> "router.cif"
net_tally = 0
one_tally[net_name] = 0
for (k = 1; k <= curr_h_segs; k++)
  { print "B " hx_cif[k],hy_cif[k],hxcnt_cif[k],
    hycnt_cif[k]  >> "router.cif"
    net_tally = net_tally + hx[k]
    one_tally[net_name] = one_tally[net_name] + hx[k]
  }
print "V " curr_v_segs >> "router.cif"
for (k = 1; k <= curr_v_segs; k++)
  { print "B " vx_cif[k],vy_cif[k],vxcnt_cif[k],
    vycnt_cif[k]  >> "router.cif"
    net_tally = net_tally + vy[k]
    one_tally[net_name] = one_tally[net_name] + vy[k]
  }
if (net_tally > old_tally)
  { old_tally = net_tally
```

Figure 17. Listing of Extractor (program also available as **$REPAS_HOME/em/extractor**) (Continued) . . .

```
        old_name = ""
        old_name = net_name
      }
print "Total Nets Processed: " tot_nets
print "Total CIF Segments: " tot_segs
cum_length = (cum_v_length + cum_h_length)/1000
print "Total MCM Wire Length: " cum_length " (mm)"
print "Average Segment Length: " cum_length/tot_segs " (mm)"
print "Longest Net: ",old_name," Length: ",old_tally/1000 " (mm)"
      for (i in one_tally)
        { printf("%10s%10.2f mm\n",i,one_tally[i]/1000) > "mcm.dat" }
 }
# End of BEGIN
  '
sort +rn1 -2  < mcm.dat > length.net
```

Figure 17. Listing of Extractor (program also available as **$REPAS_HOME/em/extractor**)

Bibliography

[1] E. Anderson and et. al. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1992.

[2] Peggy. L. Baehmann, T. -L. Sham, L. -Y. Song, and Mark. 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] G. M. Brown. Monte carlo methods. In E. F. Beckenbach, editor, *Modern Mathematics for Engineers*. McGraw Hill, NY, 1956.

[5] CAM-I. Applications interface specification (restructured version). Technical report, CAM-I Report R-86-GM-01, Arlington, TX, Jan. 1986.

[6] B. Donlan. *Design Automation for Wafer Scale Integration*. PhD thesis, Rensselaer Polytechnic Institute, Troy, NY 12180-3590, August 1986.

[7] R. Garimella, S. Dey, R. Ramamoorthy, M. K. Georges, and M. S. Shephard. Specification of mesh control functions in finite octree. Technical Report SCOREC Report # 5-1994, Rensselaer Polytechnic Institute, Troy, NY 12180-3590, January 1994.

[8] Rao Garimella, Vincent Wong, Ravichandran Ramamoorthy, Marcel Georges, and Mark S. Shephard. Support of non-manifold modeling with manifold modelers - a design document. Technical report, Scientific Computation Research Center, Rensselaer Polytechnic Institute, Troy, NY, 1993.

[9] M. K. Georges. Geometric operators for the Finite Octree mesh generator. Technical Report SCOREC Report # 13-1991, Scientific Computation Research Center, Rensselaer Polytechnic Institute, Troy, NY 12180-3590, 1990.

[10] M. K. Georges. Finite Octree data structures. Technical report, Scientific Computation Research Center, Rensselaer Polytechnic Institute, Troy, NY, 1991.

[11] A. Haji-Sheikh and E. M. Sparrow. The solution of heat conduction problems by probability methods. *Trans. ASME*, C-89:121–131, 1967.

[12]Hibbitt, Karlsson and Sorensen, Inc., 100 Medway Street, Providence, RI. *ABAQUS User's Manual Version 4.8*, July 1989.

[13]Hibbitt, Karlsson and Sorensen, Inc., 100 Medway Street, Providence, RI. *ABAQUS/Post Version 5.2*, 1992.

[14]Hibbitt, Karlsson and Sorensen, Inc., 100 Medway Street, Providence, RI. *ABAQUS/Standard Theory Manual Version 5.2*, 1992.

[15]Hibbitt, Karlsson and Sorensen, Inc., 100 Medway Street, Providence, RI. *ABAQUS/Standard User's Manual, Volume I Version 5.2*, 1992.

[16]Hibbitt, Karlsson and Sorensen, Inc., 100 Medway Street, Providence, RI. *ABAQUS/Standard User's Manual, Volume II Version 5.2*, 1992.

[17]D. Hightower. A solution to line-routing problems on the continuous plane. In *Proc. of 6th Design Automation Workshop*, pages 1–24, 1969.

[18]IBM Corporation, T.J. Watson Research Center/Hawthorne, P.O. Box 704, Yorktown Heights, NY 10598. *IBM Visualization Data Explorer, User's Guide*, second edition, August 1992.

[19]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.

[20]Märtti Mäntyla. *An Introduction to Solid Modeling*. Computer Science Press, Rockville, Maryland, 1992.

[21]C. Mead and L. Conway. *Introduction to VLSI Systems*. Addison-Wesley Publ. Co., Reading, MA, 1980.

[22]Rensselaer Polytechnic Institute, Troy, NY 12180. *REPAS - Rensselaer Electronic Packaging Analysis Software, User's Manual*, April 1994.

[23]Rensselaer Polytechnic Institute, Troy, NY 12180. *REPAS - Rensselaer Electronic Packaging Analysis Software, Technical Documentation*, April 1994.

[24]Scientific Application Internation for National Ocean Systems Center, San Diego, CA. *MagiCAD Version 2.3 User's Guide*, November 1989.

[25]T.-L. Sham, H.F. Tiersten, P.L. Baehmann, L.-Y. Song, Y.S. Zhou, B.J. Lwo, Y.L. Le Coz, and M.S. 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.

[26]Shape Data Limited, Parker's House, 46 Regent Street, Cambridge CB2 1DB England. *PARASOLID v4.0 Programming Reference Manual*, August 1991.

[27]M. S. Shephard, P. L. Baehmann, Y. L. Le Coz, and T. -L. Sham. Methodology for the integration of global/local thermal and thermo-mechanical analysis of multichip modules. In D. Agonafer and R. L. Fulton, editors, *Computer Aided Design in Electronic Packaging*, volume EEP-3, pages 65–72, New York, NY, 1992. ASME.

[28]M. S. Shephard and M. K. Georges. Automatic three-dimensional mesh generation by the Finite Octree technique. *Int. J. Numer. Meth. Engng.*, 32(4):709–749, 1991.

[29]M. S. Shephard and M. K. Georges. Reliability of automatic 3-D mesh generation. *Comp. Meth. Appl. Mech. Engng.*, 101:443–462, 1992.

[30]Mark S. Shephard, Ting-Leung Sham, L.-Y. Song, Vincent S. Wong, Rao Garimella, Harry F. Tiersten, B.J. Lwo, Yannick LeCoz, and Ralph 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.

[31]H.F. Tiersten, T.-L. Sham, B.J. Lwo, Y.S. Zhou, L.-Y. Song, P.L. Baehmann, Y.L. Le Coz, and M.S. Shephard. A global-local procedure for the thermo-elastic 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.

[32]K. J. Weiler. *Topological Structures for Geometric Modeling*. PhD thesis, Rensselaer Design Research Center, Rensselaer Polytechnic Institute, Troy, NY, May 1986.

[33]V. Wong. *Generalized SCOREC Attribute Manager*. PhD thesis, Mechanical Eng., Aeronautical Eng., & Mechanics, Rensselaer Polytechnic Institute, Troy, NY 12180-3590, 1994.