SANDIA REPORT

SAND2012-7891
Unlimited Release
Printed September 2012

Geometry Adaptive Crack Modeling and Variable Mapping

Michael G. Veilleux, John M. Emery

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2012-7891
Unlimited Release
Printed September 2012

Geometry Adaptive Crack Modeling and Variable Mapping

Michael G. Veilleux
Multi-Physics Modeling and Simulation Department
Sandia National Laboratories
P.O. Box 969
Livermore, CA 94551-0969
mgveill@sandia.gov

John M. Emery
Solid Mechanics Department
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-0372
jmemery@sandia.gov

Abstract

A high fidelity fracture modeling approach is presented that is focused on accurately quantifying the cause and uncertainty of failure for applications that involve the nucleation and propagation of dominant cracks. Two capabilities are presented:

1. A semi-automated, geometry and mesh adaption procedure for modeling arbitrarily non-planar crack evolution.
2. A robust framework for accurately mapping history-dependent variables with specific consideration of large deformations and element field gradients.

Both capabilities are considered foundations for future research, development, and application.

Acknowledgment

The authors would like to thank the Lab Directed Research and Development (LDRD) program for funding this work as an Early Career Research and Development (ECRD) project. We would like to thank the anonymous reviewers of the project's proposal, as well as the Project Manager, Mike Chiesa, for valuable insight and feedback that influenced both the direction and deliverables of the project. We greatly appreciate the generosity of Tony Ingraffea, Wash Wawrzynek, and Bruce Carter of Fracture Analysis Consultants Incorporated for granting us free source code access to FRANC3D and providing their professional assistance. We would also like to thank Jay Foulk for helping significantly with the project proposal, for mentoring during the first year of the project, and for countless valuable conversations. Finally, we would like to thank the numerous other collaborators along the way, particularly: Jake Ostien, Alejandro Mota, and the Sierra/Solid Mechanics team.

Contents

1 Introduction **9**

1.1 Background 9

1.2 Overview 10

2 Geometry Adaptive Crack Modeling **11**

2.1 Modeling Steps 11

2.1.1 UPDATE FIELDS? 11

2.1.2 RUN FEA (prior to crack insertion) 13

2.1.3 READ FEM 13

2.1.4 CREATE FLAW 14

2.1.5 WRITE FEM 15

2.1.6 MAP STATE 15

2.1.7 RUN FEA (during crack growth) 15

2.1.8 FINISHED? 16

2.1.9 FLAW GROWTH? 16

2.1.10 READ FIELDS 16

2.1.11 GROW FLAW 17

2.1.12 WRITE RESULTS 19

2.2 Python Scripting Interface 19

2.3 Examples 21

2.3.1 Uniaxial Tension 21

2.3.2 Simple Shear 22

2.3.3 Glass-Metal Seal 22

3 Variable Mapping **27**

3.1 Mapping Approach 28

3.1.1 Finite Element Model 29

3.1.2 Search Tree 31

3.1.3 Element Types 32

3.2 Unique Considerations 35

3.2.1 Exodus II Databases 35

3.2.2 Element Volumes 36

3.2.3 Composite Tetrahedral Elements 36

3.3 Examples 37

3.3.1 Mapping in the Undeformed Configuration 37

3.3.2 Mapping in the Deformed Configuration 39

3.3.3 Mapping Element Variables at Integration Points 39

4 Summary and Future Work **43**

References **46**

List of Figures

2.1 Flowchart of the geometry adaptive crack modeling approach. 12

2.2 Uniaxial tension fracture example 23

2.3 Simple shear fracture example 24

2.4 Example of a thermal strain driven crack in a glass-to-metal seal 25

3.1 Flowchart of the framework for variable mapping after mesh adaption. 29

3.2 Diagram of the finite element model class implemented for mapping 30

3.3 Diagram of the search tree class implemented for mapping 32

3.4 Diagram of the element classes implemented for mapping 33

3.5 Example of nodal variable mapping in the undeformed configuration 38

3.6 Example of nodal variable mapping in the deformed configuration 40

3.7 Example of element variable mapping from integration points 41

Chapter 1

Introduction

Many applications, at Sandia and elsewhere, involve metallic components of critical systems subject to failure due to hostile or abnormal environments. Failure necessarily involves the nucleation and propagation of cracks. However, analysts cannot determine the root cause of failure or reasonably quantify uncertainty using common modeling approaches. Such approaches typically reduce model complexity at the costs of generality and/or physical fidelity in the representation of crack nucleation and propagation.

We seek to overcome these shortcomings by explicitly representing fracture, *i.e.* arbitrarily non-planar evolution of geometrical facets (fracture surfaces) typically driven by highly non-linear material response. The focus of this paper is on accurate geometrical representation of fracture through local topology and mesh adaption, and subsequent mapping of history-dependent variables.

1.1 Background

The main hypothesis overarching this approach is that accurate fracture modeling can be achieved by combining three high fidelity modeling capabilities: local damage modeling, strain localization elements, and geometrically explicit fracture surface representation with adaptive remeshing.

Local damage models, such as those incorporated in the BCJ plasticity model [1], are commonly used by analysts, because the fracture process is included in the constitutive response, thus avoiding the implementation complexities of explicit fracture representation. However, local damage models are substantially limited in accuracy due to mesh size dependency and post-peak softening leading to bifurcation of the solution and corruption of the partial differential equation.

Strain localization elements are surface elements that construct a deformation gradient with a user-specified thickness [17]. Implementing these elements in combination with local damage models regularizes the fracture representation via the localization length scale. However, this combination is still insufficient for modeling fracture in a generalized manner without a technique for adaptively representing an arbitrary, predicted crack path along which to place localization elements.

To facilitate this need, a geometrically explicit fracture representation with adaptive remeshing scheme is implemented in this study. Such a scheme consists of a database of geometry and/or topology information describing the solid model and the fracture surfaces and a robust meshing algorithm for generating high quality volume meshes that conform to arbitrary crack shapes[1] [8]. Many flavors of geometrical and non-geometrical fracture representation methods are available [7], but the method proposed here is one of the few that is geometrically accurate and independent of the mesh topology. The mesh conforms to the crack rather than the crack conforming to the mesh. Inherently, this approach requires the ability to appropriately map history-dependent variables at each crack adaption step.

1.2 Overview

Accordingly, the work summarized in this paper consists of two main steps: Chapter 2 – integrating a crack geometry adaption approach into Sandia's finite element analysis environment, and enhancing the approach as necessary to model quasi-static, ductile fracture; and, Chapter 3 – developing a variable mapping approach that robustly maps fields and material state data for models with large deformations and gradients, *e.g.* models of cracks in ductile materials. Chapter 4 summarizes the current state of the approach and outlines the path forward.

[1] Arbitrary crack shapes include non-planarities, multiple crack tips (*e.g.* branching), and coalescense. For the purposes of simplifying the initial implementation, this study focuses on arbitrarily non-planar cracks without branching or coalescence.

Chapter 2

Geometry Adaptive Crack Modeling

The first objective of this project was to implement a geometrically explicit, adaptive crack modeling approach. The flowchart in Figure 2.1 depicts this implementation. The steps outlined and written in green are performed by the crack and mesh adaption software. FRANC3D [15] is used for crack and mesh adaption, because it is specifically designed for the application and the authors of this paper have extensive prior experience with its capabilities and source code. Commercial license agreements will have to be negotiated for future usage of FRANC3D. The steps outlined and written in blue are performed by the finite element analysis software, *i.e.* SIERRA/SM [11]. Currently, all steps outlined and written in black are neither performed by FRANC3D nor SIERRA/SM, so separate codes, scripts, and/or user interactions are required to perform these steps. The gray-colored region indicates the tasks automated during this project by implementation of a Python [4] scripting interface. All steps outlined and written in bold were created or modified during this project; a bold, solid outline indicates a newly created capability, and a bold, dashed outline indicates a modified capability. The steps shown in Figure 2.1 are sequentially detailed in Section 2.1. Individual components of the Python scripting interface are described, as appropriate, in each section. The overall structure of the scripting interface is then summarized in Section 2.2.

Section 2.3 concludes this chapter with three adaptive crack modeling examples: uniaxial tension, simple shear, and thermal strain. The latter is intended to represent a realistic application: a crack nucleating and propagating near a glass-metal interface in a thermally loaded component.

2.1 Modeling Steps

The following is a description of each step shown in Figure 2.1.

2.1.1 UPDATE FIELDS?

The **UPDATE FIELDS?** decision operator is required to determine:

1. whether finite element analyses need to be performed prior to crack insertion; and, if

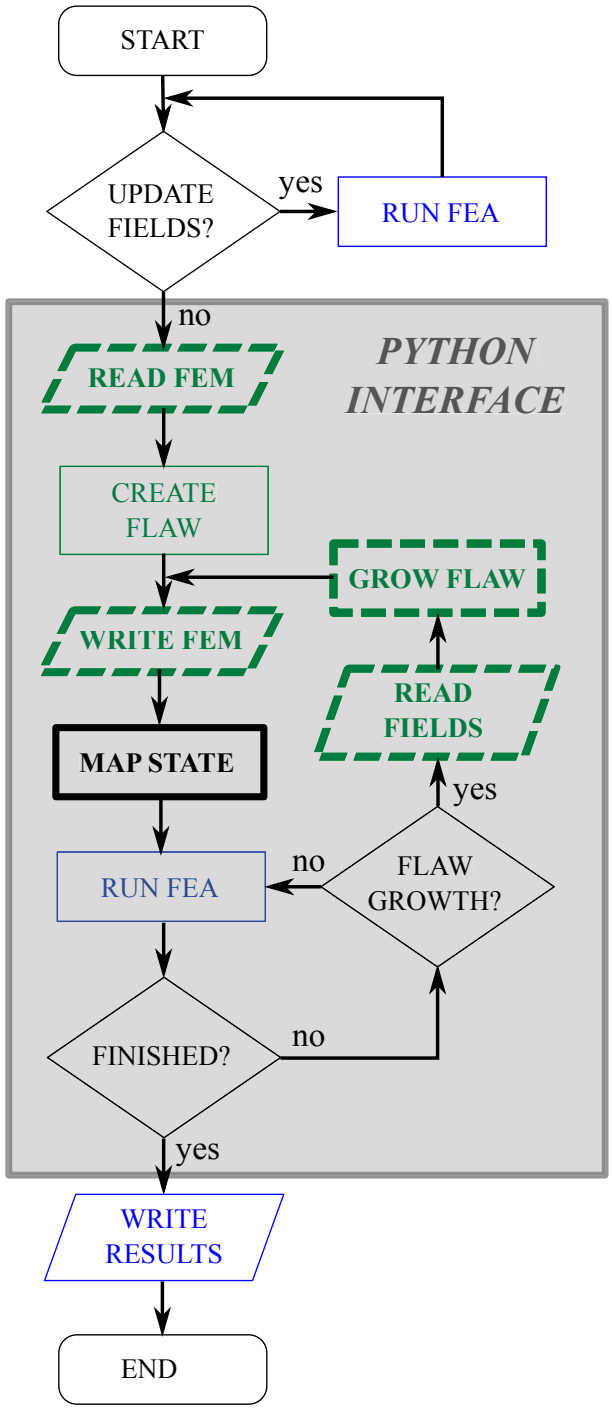

Figure 2.1. Flowchart of the geometry adaptive crack modeling approach.

so,

2. when crack nucleation is predicted by the finite element analysis.

These are currently user determinations that could be automated in the future, either by externally wrapping the finite element analysis code or by embedding the decision operator into the analysis code. Both types of automation require the ability to terminate an analysis when a crack should be inserted. If the crack is an initial flaw, *i.e.* existing prior to loading, then fields do not need to be computed prior to crack insertion. Otherwise, one or more iterations of finite element analyses are performed to calculate the fields governing nucleation.

Accurately predicting when, where, and how a crack nucleates is fundamental to fracture modeling, but it is still an area of active research with yet many unknowns. The authors of this paper envision future research projects that utilize the geometrically explicit modeling approach to:

1. explicitly model fracture processes at the lower length scale(s), prior to 'nucleation' at the component scale, so that nucleation can be more accurately predicted; and/or,

2. computationally calibrate and validate models that predict nucleation.

2.1.2 RUN FEA (prior to crack insertion)

The first **RUN FEA** step is associated with the **UPDATE FIELDS?** decision operator, as described in the previous section.

2.1.3 READ FEM

READ FEM is the first step performed by FRANC3D. A finite element model can be read from either a file or a `FemModel` class instance; the latter is the internal data structure used by FRANC3D to represent a finite element model. FRANC3D can read multiple finite element file formats, including those from ABAQUS™ and ANSYS™. Initially, this study sought to use an EXODUS II [9] reader in FRANC3D for compatibility with SIERRA/SM. However, this reader is not maintained by the FRANC3D developers, thus leading to potential incompatibilities as FRANC3D is updated.

Thus, a Python module was written for this study to translate between an EXODUS II file and a `FemModel` class instance. Although FRANC3D is being actively developed, the `FemModel` class is mature; therefore, an EXODUS II – `FemModel` translator requires little maintenance. The translator utilizes the FRANC3D Python API and a Python wrapper of the EXODUS II API. Both of these Python applications had to be extensively augmented during this study, so that all communication between FRANC3D and EXODUS II could

be in memory. The procedure for converting an EXODUS II file to a `FemModel` instance is encapsulated in one translator function call, `exoToFemModel`.

A subsequent Python function call creates a FRANC3D `F3DApp` class instance from a `FemModel` instance. The function, `createF3DAppFromFemModel`, reads a `FemModel` and returns a `F3DApp`, the main driver of all FRANC3D operations. When the `F3DApp` is created, all nodesets and sidesets from the original EXODUS II file are flagged as groups of nodes and element sides, respectively, that must not be removed during crack insertion and remeshing. This guarantees that any definitions applied to the nodesets and sidesets by another application, such as the boundary conditions defined in a SIERRA/SM input deck, will still function properly.

2.1.4 CREATE FLAW

After the finite element model is read in, the **CREATE FLAW** step generates and inserts a nucleated crack into the model. FRANC3D can create two types of flaws: voids or cracks. The former has volume and the latter does not. Void insertion and adaption was not investigated for this project; herein, a flaw and a crack are synonymous. Flaw creation follows two sub-steps: flaw definition and flaw insertion.

A flaw can be defined by any one of three methods: graphically, parametrically, or by file. The graphical approach uses the FRANC3D flaw insertion Graphical User Interface (GUI), which allows the user to interactively define the crack insertion parameters. The parametric approach defines these same parameters directly inside the FRANC3D Python API. Both the GUI and parametric approaches create an instance of the FRANC3D `Flaw` class – a faceted, geometrical description of the crack that can be saved to a text file. The file approach creates a `Flaw` from one of these text files.

The reader is referred to the FRANC3D manual [14] for descriptions of the types of flaw shapes that can be created, and how they can be adapted. Virtually any crack shape, size, and location, and mesh density can be defined. One powerful capability of FRANC3D is the ability to define a well-structured template of volumetric elements around a crack front that lend to better resolution of near-crack singularities, thus giving optimal computations of crack driving forces, such as stress intesity factors and J-integral values. However, this template currently cannot be used by SIERRA/SM due to the element types utilized in the template[1].

In this project, the Python interface wraps all three of the crack definition methods. The function calls are `createFlaw` and `insertFlaw`. The first command defines the flaw by any of the three methods described above. The second command, `insertFlaw`, inserts the `Flaw`

[1]The crack front template in FRANC3D utilizes multiple element types, including quadratic wedges. To be compatible with the EXODUS II file format, code will have to be added to the Python interface that generates a unique element block for each element type in the template. Also, a quadratic wedge element will have to be added to the element library in SIERRA/SM.

into the finite element model.

The insertion process follows four steps: construct a geometry model (*i.e.* a boundary representation) from the original `FemModel`; delete the mesh in a region local to where the crack is being inserted; update the geometry model to include the crack; and, remesh the local region with the crack included. The default remeshing routine has very few controls for mesh density at and nearby the crack, especially when the crack front template is not used. However, FRANC3D has been extended to external meshing programs, such as ABAQUS and ANSYS, as an alternative for remeshing so that users can have more control on mesh density. This approach would be especially useful for mesh convergence studies. The `insertFlaw` call automatically updates the `F3DApp` and `FemModel` to include the crack and new mesh.

2.1.5 WRITE FEM

The **WRITE FEM** step is iteratively called, after each crack adaption, by FRANC3D. As stated in the description of the **READ FEM** step, a Python module has been written to translate between an EXODUS II file and a `FemModel`. The Python function `femModelToExo` was written for this project to translate a `FemModel` to an EXODUS II file.

2.1.6 MAP STATE

The **MAP STATE** step is required to map history-dependent variables, such as material state information, after each crack adaption. FRANC3D only updates the finite element model geometry and mesh, so analysis variables are not carried forward to the updated mesh. Variable mapping is a challenging problem that demands careful attention to both algorithms and theory. This is a continuously developing area of research and application, at Sandia and in the entire finite element modeling community, that has many extensions beyond the scope of this project. However, it was deemed necessary for completion of this project to implement a new mapping framework that could work robustly for large deformation and fracture modeling. This framework is intended to significantly evolve in maturity through future research and development activities. Given the complexity of the problem and the significant effort already focused on mapping in this project, a separate chapter of this report, Chapter 3, is dedicated to summarizing this step. Thus, in this chapter, the **MAP STATE** step is treated as a black box that transfers all history dependent variables from a source finite element model (prior to crack adaption) to a target finite element model (after crack adaption).

2.1.7 RUN FEA (during crack growth)

The second **RUN FEA** step in the flowchart is required to analyze fields after each increment of crack growth. This step is performed by the finite element analysis code, *i.e.* SIERRA/SM.

Finite analyses for adaptive crack modeling must be closely linked with decision operators that determine when a crack has reached a critical length or when the crack is predicted to continue propagation. These decision operators, **FINISHED?** and **FLAW GROWTH?**, respectively, are described in the following two sections.

2.1.8 FINISHED?

The **FINISHED?** decision operator is a determination of when the entire fracture analysis is complete. This is currently a user decision that could be automated in the future, either by externally wrapping the finite element analysis code or by embedding the decision operator into the analysis code. Both types of automation require the ability to terminate code execution when the fracture analysis is determined to be complete. For example, termination could be signified by the crack reaching a critical length or by one of the near-crack fields reaching a critical value that indicates unstable crack growth. If the analysis is considered to be complete, then the finite element analysis results are written to file. Subsequently, FRANC3D can be called as a post-processor to evaluate histories of crack parameters, such as stress intensity factors, J-integral values, and crack displacements. If the analysis is not complete, then a decision must be made as to when the crack will propagate.

2.1.9 FLAW GROWTH?

The **FLAW GROWTH?** decision operator determines whether a crack should be propagated. Like the other decision operators, this is currentlty a user decision that could be automated in the future, which would require the abilities to periodically evaluate the fields governing crack growth and to interrupt an analysis once the crack should be grown. This decision operator could become an interaction between FRANC3D and the analysis code, utilizing the crack growth calculation capabilities of the former. While the crack is not predicted to grow, the finite element analysis continues to increment in time. Once crack growth is predicted, the finite element analysis results governing crack growth are read into FRANC3D for automated crack propagation. This exchange of analysis results is further explained in the following section.

2.1.10 READ FIELDS

The **READ FIELDS** step transfers the finite element analysis fields governing crack growth from the analysis code to FRANC3D. These fields are currently the most recently computed nodal displacements, which FRANC3D then uses to compute crack growth parameters via displacement correlation (see Section 2.1.11). The finite element analysis results can either be read in by file or by a `FemResults` class instance.

For this project, a Python module was written that translates finite element results from

an EXODUS II file to a FRANC3D `FemResults` class instance. Like the model translator described in Section 2.1.3, the results translator utilizes the FRANC3D Python API and a Python wrapper of the EXODUS II API. This translation procedure is wrapped in one Python function call, `readFemResultsFromExo`. See Section 2.2 for an example that calls this function. When an analysis is driven by the Python interface, one `F3DApp` exists for the entire fracture analysis procedure; at each crack growth increment, new `FemResults` instances are added to the `F3DApp`.

Also for this project, methods had to be added to the `F3DApp` for processing field data for a model without a crack front template. Many of the methods required for computing crack growth distances and directions were designed for displacement correlation calculations on a model with a template. These calculations, as described in the following section, have now been successfully extended to a model without a template. Most importantly, methods were added for computing a correct, consistent local basis at each point along a crack front, which is used to compute many derived fields, including crack opening/sliding displacements, crack growth distances, and crack growth rates.

2.1.11 GROW FLAW

The **GROW FLAW** step is the main FRANC3D driver for propagating a crack, which consists of four sub-steps: determination of new crack front locations, local mesh deletion, crack geometry insertion, and local remeshing. The methods for determining the new crack front locations are briefly described here. The final three steps are not discussed here; they are performed automatically by FRANC3D.

New crack front locations can be determined by numerous methods in FRANC3D, as described in the FRANC3D manual [14]. The most versatile method is reading the new front locations from file. However, this requires user interaction or separate programs to calculate or define the new front locations. Such an approach is required when employing a crack growth rule that is not programmed in FRANC3D. Currently, multiple growth rules are available, several for fatigue and one for quasi-static propagation.

For the example problems shown in Section 2.3, the quasi-static rule, following linear elastic fracture mechanics, is employed. This methodology is given in the FRANC3D manual, but briefly described here for completeness. First, the modes I, II, and III stress intensity factors, K_I, K_{II}, and K_{III}, respectively, are calculated by displacement correlation via the plane-strain K-field,

$$K_I = \frac{E}{8(1-\nu^2)} \frac{COD(r)}{\sqrt{\frac{r}{2\pi}}} \tag{2.1}$$

$$K_{II} = \frac{E}{8(1-\nu^2)} \frac{CSD(r)}{\sqrt{\frac{r}{2\pi}}} \tag{2.2}$$

$$K_{III} = \frac{E}{8(1+\nu)} \frac{CTD(r)}{\sqrt{\frac{r}{2\pi}}} \tag{2.3}$$

where r is distance from the crack tip (crack front in 3D), E is Young's modulus, ν is Poisson's ratio, and $COD(r)$, $CSD(r)$, and $CTD(r)$, are the crack opening, sliding, and tangential (screw) displacements, respectively, at r. The crack displacement components are computed at $r = 2L_{el}$ behind the crack front, where L_{el} is the average characteristic element length along the crack front. The crack direction then follows the maximum tensile stress (MTS) theory, *i.e* the direction that maximizes the magnitude of the near-crack tensile stress,

$$\sigma_{\theta\theta}\sqrt{2\pi r} = \cos\frac{\theta}{2}\left[K_I \cos^2\frac{\theta}{2} - \frac{3}{2}K_{II}\sin\theta\right] \tag{2.4}$$

where θ is angle with respect to the local crack axes. $\theta = 0$ degrees is equivalent to planar crack growth and $\theta = 90$ degrees is equivalent to a perpendicular kink in the crack growth direction. Equation 2.4 is evaluated at a series of points along the crack front, so the crack shape for each step of growth can be arbitrarily non-planar. The crack propagation distance is calculated by a median distance formula,

$$\Delta a_i = \Delta a_{median}\left[\frac{K_i}{K_{median}}\right] \tag{2.5}$$

where Δa_i and K_i are the crack growth distance and K_I, respectively, at crack front point i, and Δa_{median} and K_{median} are the crack growth distance and K_I, respectively, at the crack front point that has the median K_I value. Δa_{median} is currently a user input, but this could be replaced by a crack growth distance predicted by the finite element analysis.

The envisioned approach, as mentioned in Section 1.1, is to virtually propagate the crack a distance by similar steps to those described here, so that surface elements, such as strain localization elements, could be inserted along the virtual crack growth increment. Subsequent analyses would allow the crack to naturally propagate by separation of the surface elements. Future research is intended to evaluate this approach.

Although the aforementioned methodology is limited to linear elastic fracture mechanics assumptions, the user can grow cracks for virtually any fracture regime, including ductile fracture. For example, the capability is currently in place for performing the following procedure to model ductile fracture:

1. Read results fields into FRANC3D.
2. Call FRANC3D methods for computing J-integral values and/or crack displacement components (opening/sliding) at points along the crack front.
3. Call FRANC3D methods to output the J-integral or crack displacement values.

4. Use an external procedure for computing new crack front points based on the J-integral or crack displacement values.

5. Call FRANC3D methods that read in the externally computed crack front points and grow the crack in a manner that best fits these new points.

As previously mentioned, the last step listed here gives the user the freedom to apply virtually any method to predict crack growth. For such cases, FRANC3D is used solely to adapt the geometry and mesh to fit the user's input set of crack front locations.

2.1.12 WRITE RESULTS

The **WRITE RESULTS** step signifies the end of a fracture analysis, where the end result is a results database, *i.e.* an EXODUS II file, containing the entire history of computed fields for all time steps, from crack nucleation to failure. As mentioned in Section 2.1.8, FRANC3D can also be used to compute histories of crack fields. Examples of these computed fields are plotted in Section 2.3.

2.2 Python Scripting Interface

The following is an example usage of the Python scripting interface delineated in Figure 2.1. Our fictitious model has an elliptical initial crack with a radius of 0.1 in both principal directions and a median crack crack growth increment of 0.01.

```
 line 1:  import franc3DMethods
 line 2:  import exoFranc3DMethods
 line 3:  #
 line 4:  # create a FemModel from an Exodus file
 line 5:  inputFile = ‘myModelName.g’
 line 6:  exoFranc3DMethods.exoToFemModel( inputFile )
 line 7:  #
 line 8:  # initialize an F3DApp with this FemModel
 line 9:  f3dApp = franc3DMethods.createF3DAppFromFemModel( femModel )
line 10:  #
line 11:  # insert a parametrically defined flaw
line 12:  f3dApp.InsertParamFlaw( ‘flaw_type’ = ’CRACK’,
line 13:                          ‘crack_type’ = ’ELLIPSE’,
line 14:                          ‘flaw_params’ = [0.1, 0.1],
line 15:                          ‘translation’ = [0.5, 0.5, 0.5],
line 16:                          ‘x_axis’ = [1.0, 0.0, 0.0],
```

```
line 17:                          'y_axis' = [0.0, 0.0, -1.0],
line 18:                          'refinement_level' = 2,
line 19:                          'do_template' = False )
line 20:  #
line 21:  # output the initially cracked model for analysis
line 22:  step = 1
line 23:  outputFile = 'myModelName.step%i.g' % step
line 24:  exoFranc3DMethods.femModelToExo( femModel,
line 25:                                   outputFile )
line 26:  #
line 27:  # tell user to map & analyze, then continue
line 28:  exoFranc3DMethods.userAnalyze()
line 29:  #
line 30:  # grow crack until user indicates completion
line 31:  while exoFranc3DMethods.notFinished():
line 32:      #
line 33:      # read in current nodal displacements
line 34:      resultFile = 'myModelName.step%i.e' % step
line 35:      exoFranc3DMethods.readFemResultsFromExo( f3dApp,
line 36:                                               resultsFile )
line 37:      #
line 38:      # use displacement correlation to grow the crack
line 39:      f3dApp.GrowCrack( sif_method = 'DISP_CORR',
line 40:                        const_median_step = 0.01,
line 41:                        use_template = False )
line 42:      #
line 43:      # output the updated cracked model for analysis
line 44:      step = step + 1
line 45:      outputFile = 'myModelName.step%i.g' % step
line 46:      exoFranc3DMethods.femModelToExo( femModel,
line 47:                                       outputFile )
line 48:      #
line 49:      # tell user to map & analyze, then continue
line 50:      exoFranc3DMethods.userAnalyze()
line 51:  #
line 52:  # crack growth simulation complete!
```

The `franc3DMethods` module, imported on **line 1**, originated from this project and it is designed to provide a clear set of functions that interact solely with the FRANC3D Python API. Only one method from the `franc3DMethods` module is used in this example, the call

to `createF3DAppFromFemModel` on **line 9**, but other methods are available. For example, there is a wrapper of the process for inserting a parametric flaw that reads parameters from a file, rather than explicitly defining them in the script, as is done here on **line 12**. There are additional methods for inserting flaws from the FRANC3D flaw insertion GUI and from a flaw description file.

The `exoFranc3DMethods` module, imported on **line 2**, also originated from this project and it is designed to provide a clear set of methods that interact between the Python wrapper of the EXODUS II API and the FRANC3D Python API. These methods include those for translating between EXODUS II and `FemModel` instances, *i.e* the `exoToFemModel` and `femModelToExo` commands on **line 6** and **line 24**, respectively. In addition, the `userAnalyze` method, on **line 28**, signifies to the user that a model with an updated crack state is available for material state mapping and analysis, both of which are currently performed outside of the script. `userAnalyze` pauses the script until the user indicates that a new set of results are available for growing a crack. Every time `userAnalyze` is called, the `notFinished` method, on **line 31**, is called to determine whether the user wants to continue with the crack growth process. If the crack growth process is signified to continue, then the `readFemResultsFromExo` command, on **line 35**, is called to apply the current analysis results information relevant to crack growth, *e.g.* nodal displacements, to the `F3DApp` instance.

A few methods shown here are calls made directly on a `f3dApp` instance. The reader is referred to the FRANC3D command language manual [13] for further description of these methods. FRANC3D is a versatile, modular tool with many functionalities not described here, and most of these functionalities are exposed through the Python API. Thus, the script shown here is just one example of many ways to utilize the Python interface shown in Figure 2.1.

2.3 Examples

The following are three examples of geometrically explicit fracture analyses following the procedure given in Figure 2.1. The first two examples are idealized configurations where the crack path is known *a priori*, and the final example is derived from a user application where the crack path is unknown.

2.3.1 Uniaxial Tension

This example, shown in Figure 2.2, is a plate with a crack loaded in uniaxial tension. The model, illustrated in Figure 2.2(a), is square in the x-y plane with length and width equal to 5.0. The initial crack length, $2a$, and the depth in the z-direction are both equal to 1.0. The material is elastic, with a Young's modulus of 100000.0 and a Poisson's ratio of 0.3. Since the material is elastic, no mapping is performed and the model starts from the unloaded state at each crack increment. At maximum load, a displacement of 0.01 is

applied in the y-direction on the y_{max} surface. Figure 2.2(b) and Figure 2.2(c) show the displaced shape, magnified by $10X$ for visualization purposes, at maximum applied load for the crack at the first and fifth crack growth steps, respectively. Figure 2.2(d) shows the crack surface at the first step, *i.e.* for the initial crack, and illustrates the normalized distance, L, along which FRANC3D computes crack propagation parameters. There are two crack fronts for a through-crack. $L < 0.5$ represents one crack front and $L > 0.5$ represents the other. Figure 2.2(e) shows the crack opening and sliding displacements, COD and CSD, respectively, for the initial crack. These values are expected to be identical between the two crack fronts and symmetric along each crack front, due to the x- and z-direction symmetries of the problem. However, these symmetries are not precisely represented due to the lack of structure and coarseness of the mesh. As seen upon close inspection of Figure 2.2(c), the inaccuracy of the solution eventually leads to slight crack kinking. The stress intensity factors computed by FRANC3D, Figure 2.2(f), match the crack displacements, as expected for their relationships given in Section 2.1.11. K_I is much higher than K_{II} since this is a mode I fracture problem.

2.3.2 Simple Shear

This example, summarized in Figure 2.3, is a plate with a crack loaded in simple shear. The initial dimensions and material parameters are the same as the uniaxial tension problem, so the reader is referred to Section 2.3.1 for these dimensions and parameters. At maximum load, a displacement of 0.01 is applied in the positive x-direction on the y_{max} surface. Figure 2.3(b) and Figure 2.3(c) show the displaced shape at maximum applied load for the crack at the first and fifth crack growth steps, respectively. Figure 2.3(e) shows the crack opening and sliding displacements, COD and CSD, respectively, for the initial crack. These values are expected to be asymetric between the two crack fronts. This asymmetry is approximately represented; slight inaccuracies in the solution are due to a combination of mesh coarseness and an unstructured mesh. Figure 2.3(c) shows that the simulated crack kinking qualitatively matches the expected result for shear loading. Figure 2.3(f) shows the computed stress intensity factors for the initial crack. As expected, K_{II} is much higher than K_I since this is a mode II fracture problem.

2.3.3 Glass-Metal Seal

This final example, shown in Figure 2.4, is a cylindrical glass-to-metal seal with thermal strains as the only loads and a prescribed initial crack in the glass. The model, illustrated in Figure 2.4(a), is an axisymmetric wedge representing a 10-degree section of the seal. The initial crack is semi-circular, with a representing its radius. The crack plane is parallel to the glass-metal interface and located a distance of $2a$ from the interface. The glass is modeled as elastic and the metal is modeled as thermo-elastic-plastic. Figure 2.4(b) shows the xx-stress contours and deformed shape for half of the model, which is sliced through the middle of the crack to better show the near-crack stresses and deformation. The x- and r-directions

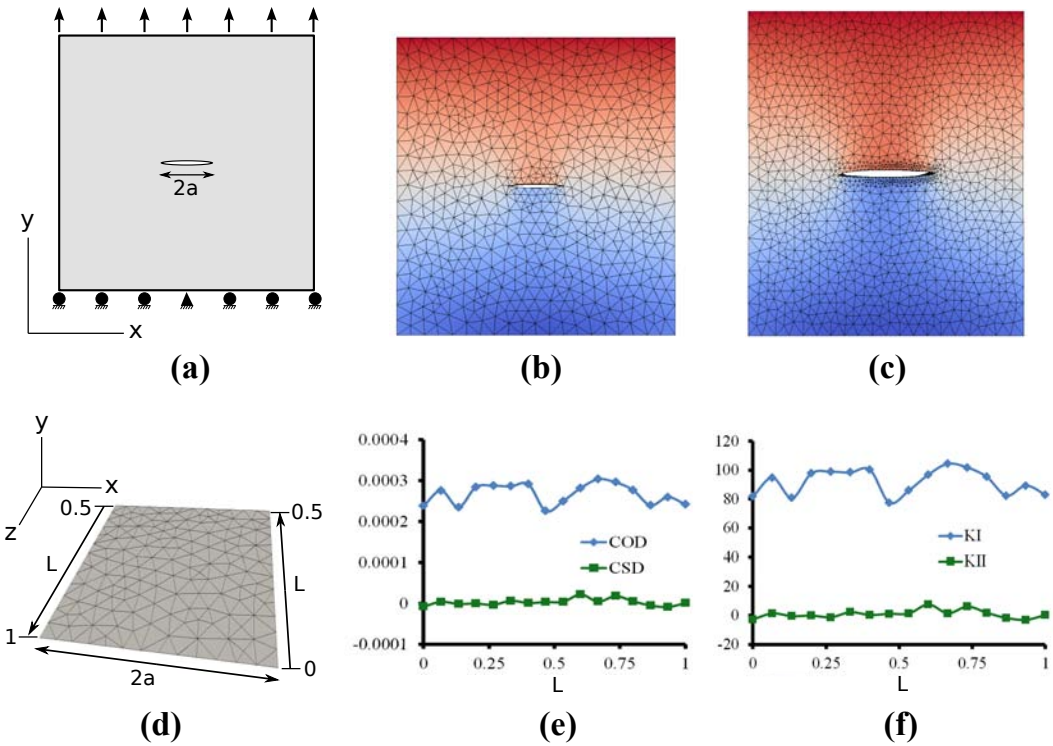

Figure 2.2. Uniaxial tension fracture example: (a) Model configuration; (b) Displaced shape ($10X$ magnification) and displacement contours at the first crack growth step; (c) Displaced shape ($10X$ magnification) and displacement contours at the fifth crack growth step; (d) Crack geometry and mesh at the first crack growth step; (e) COD and CSD at the first crack growth step; and, (f) K_I and K_{II} at the first crack growth step.

are approximately the same at the crack location; thus, the stress shown is approximately the rr-stress. The metal is more ductile than the glass, which increases stress in the glass at the assumed crack location. Figure 2.4(c) shows the shape of the initial crack and illustrates the normalized distance, L, along which FRANC3D computes crack propagation parameters. Figure 2.4(d) shows the crack opening and sliding displacements, and Figure 2.4(e) shows the modes I and II stress intensity factors. Substitution of the plotted K_I values into Equation 2.5 suggests that the next step of crack growth would not be self-similar; the crack would grow faster at the surface than at its maximum depth. Furthermore, the K_I and K_{II} values were found to be non-zero along the entire crack front, so Equation 2.4 suggests that the crack will grow in a non-planar fashion; the crack will have a near-zero kink angle at the surface and a sharp kink at its maximum depth.

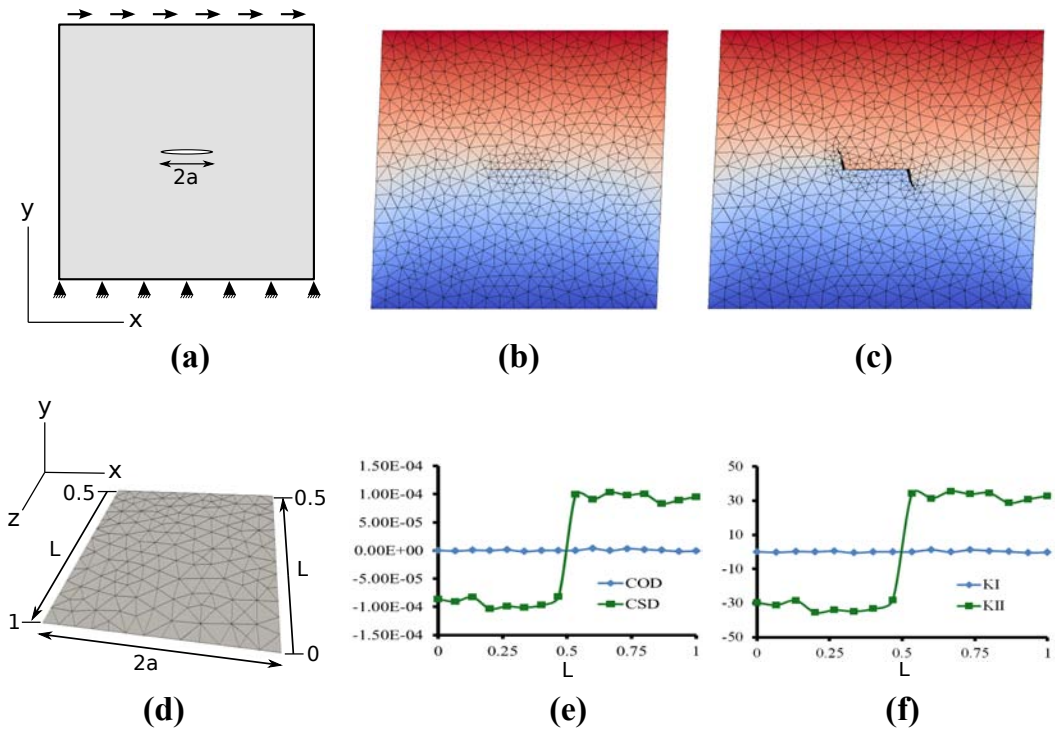

Figure 2.3. Simple shear fracture example: (a) Model configuration; (b) Displaced shape ($10X$ magnification) and displacement contours at the first crack growth step; (c) Displaced shape ($10X$ magnification) and displacement contours at the fifth crack growth step; (d) Crack geometry and mesh at the first crack growth step; (e) COD and CSD at the first crack growth step; and, (f) K_I and K_{II} at the first crack growth step.

Accurately modeling subsequent steps of crack growth for this problem likely requires variable mapping due to the non-linear response of the metal. This is just one of the many modeling applications that motivates the need for a robust variable mapping framework that transfers state data and fields as accurately as possible, especially for localized regions with high gradients (*e.g.* cracks). The following chapter presents a framework that is designed to address this need.

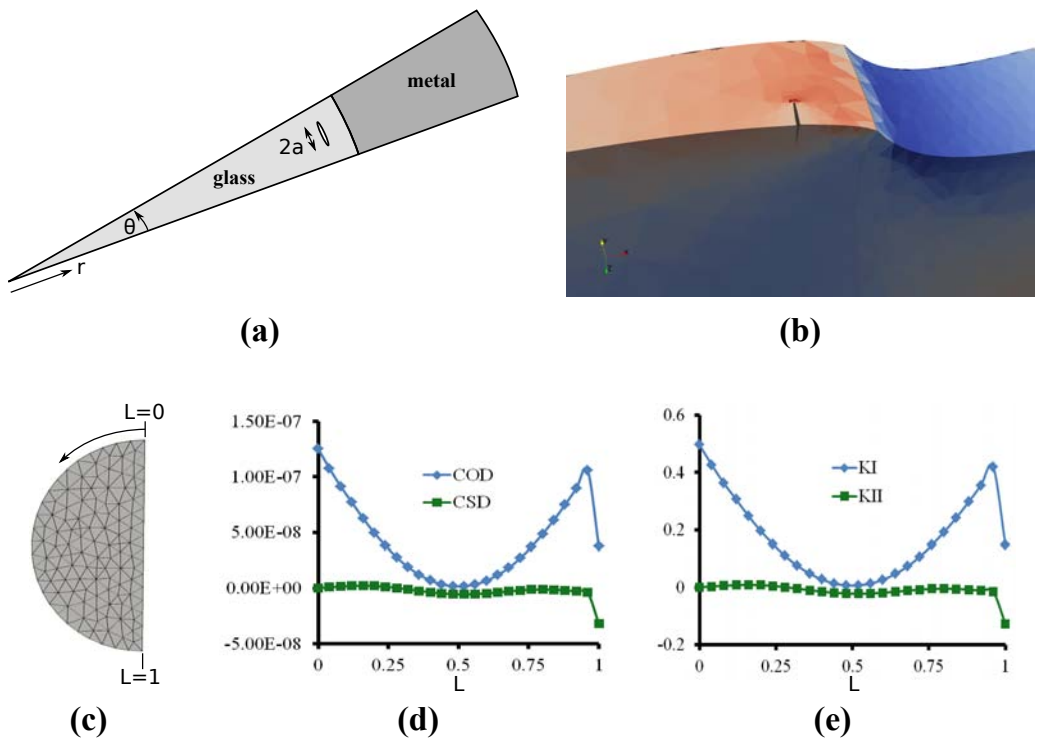

Figure 2.4. Example of a thermal strain driven crack in a glass-to-metal seal: (a) Model configuration; (b) Displaced shape (magnified) and xx-stress contours near the crack at the first crack growth step; (d) Crack geometry and mesh at the first crack growth step; (e) COD and CSD at the first crack growth step; and, (f) K_I and K_{II} at the first crack growth step.

Chapter 3

Variable Mapping

Mesh adaption applications, such as geometry adaptive crack modeling, necessarily require the ability to transfer all information defining the state of a finite element analysis from the original model (the source) to the adapted model (the target). Field variable mapping is just one component of a transfer. Defining equivalent mesh sets and attributes, global variables, and time steps are just some of the many other transfer components. For many common modeling applications, such as contact, transfer of these other components is a non-trivial task. Furthermore, the type, complexity, and robustness of the transfer routines depend greatly on the application. For example, adaptive mesh refinement can be treated as a series of local, independent transfers, each from a single element in the source to its subdivisions in the target, whereas arbitrary mesh adaption must treat the problem more globally since the mesh topology of the target generally does not depend on the mesh topology of the source. Geometry adaption is even more complex than arbitary mesh adaption, because surfaces and sub-domains can exist in the target that do not exist in the source, and vice-versa. The transfer routines required for geometry adaptive crack modeling are some of the most complex, and many of the assumptions made by other types of transfers are invalid.

The work described in this chapter is motivated by the need to perform transfers for geometry adaptive crack modeling; the immediate focus is on creating a transfer routine that can be researched and developed for this and similar applications – large deformations and sharp gradients in history-dependent materials. This work is influenced by similar transfer routines developed at Sandia, such as MAPVAR [16]. MAPVAR is designed for mesh adaption, but the current version has limitations that prevent transfers for geometry adaptive crack modeling. The following are the most substantial currently known limitations:

1. Solid elements are limited to 8-noded hexahedra, 8-noded tetrahedra, or 4-noded tetrahedra. Geometry adaptive crack modeling codes, such as FRANC3D, rely on unstructured tetrahedral meshing, for which a 10-noded tetrahedral element formulation is required to obtain accurate solutions. Therefore, a mapping capability is needed for 10-noded tetrahedra.

2. Element quantities in the source mesh are assumed to be volume-averaged values. Modeling of gradients within elements, such as those present in crack modeling, requires multi-point integration to get an accurate solution. Therefore, a capability to map from integration points is required.

3. Coincedent crack surfaces are not uniquely considered. To accurately map fields at crack surfaces, especially discontinuities, the mapping must interpret which nodes and elements lie on each of the coincident surfaces in both the source and target meshes.

4. Identical element types are required between the source and target meshes. Modeling applications commonly use hexahedral meshes, in which cases crack insertion and propagation will replace hexahedral meshes with tetrahedral meshes. Therefore, the mapping must be extended to process different element types between the source and target meshes.

One avenue for this work could have focused on extending MAPVAR to eliminate these limitations. However, writing a new code was determined to be more efficient and flexible. The initial implementation of the new code, as described here, does not address the last two items, but it is structured to efficiently include and test these capabilities in the future. The authors would like to thank the developers of MAPVAR and its precursors, MERLIN [5] and MERLIN II [6], for the valuable insight. The authors acknowledge that transfer routines are also available in SIERRA [3], but a stand-alone code was deemed more appropriate for this project.

The remainder of this chapter describes and exemplifies the new transfer code, with particular focus on variable mapping. Section 3.1 details the mapping approach, Section 3.2 describes features that are uniquely considered during the transfer process, and Section 3.3 gives examples of the current transfer capabilities.

3.1 Mapping Approach

A flowchart of the steps generally required for transfers with variable mapping is shown in Figure 3.1. The mapping routine begins by reading finite element databases for both the source and target meshes, the **OPEN SOURCE FEM** and **OPEN TARGET FEM** steps, respectively. The source database has the source finite element model and the analysis state(s) necessary for a restart. The target database has the target finite element model. The **INITIALIZE TRANSFER** step transfers all information that does not have to be mapped, *i.e.* everything except the nodal and element variables. The **CREATE SEARCH TREE** step creates a spatial search tree of the source mesh, which greatly reduces the computational cost of the spatial searches performed during the mapping. Finally, the field variables are mapped in the **MAP NODE VARIABLES** and **MAP ELEMENT VARIABLES** steps.

Specifics of this procedure and high-level code structures are given in the following three sections. The code is developed in C++ and in an object-oriented manner to allow for efficient development of future capabilities and portability of individual components into other finite element codes. Section 3.1.1 describes the finite element model class, including the driver methods for all of the steps shown in Figure 3.1. Section 3.1.2 describes the spatial search tree and Section 3.1.3 describes the element classes and the methods implemented therein to perform the mapping.

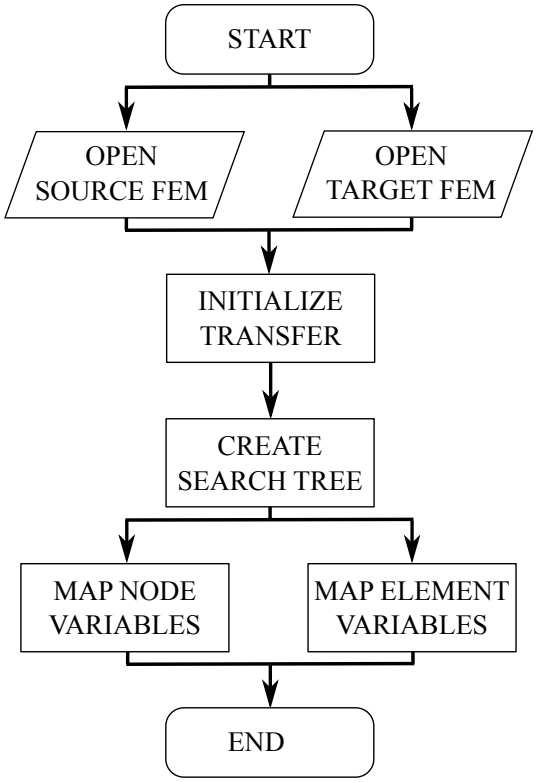

Figure 3.1. Flowchart of the framework for variable mapping after mesh adaption.

3.1.1 Finite Element Model

A high-level class diagram of the finite element model data structure used by the mapping code is shown in Figure 3.2. Only public methods are shown for brevity; all member variables and private methods are omitted. The construct consists of a base class, `fem`, and derived classes. Only one derived class is currently available, `exoFem`, which is compatible with EXODUS II files through calls to the EXODUS II C++ API. The idea, however, is to allow for multiple derived classes, so that other finite element database formats can be interpreted. The methods shown here for the `fem` base class are virtual, because each is redefined by the derived classes.

An `exoFem` has two constructors, the first for reading the source model and the second for reading the target model. The source's constructor has one input, the name of the EXODUS II file to which it is stored. The target's constructor has two inputs, the name of the EXODUS II file to which its mesh is stored and the name of the new EXODUS II file that will be created after the transfer.

After the source and target `exoFem` classes are constructed, the `readMtrlsMap`, `copyAppropriate`,

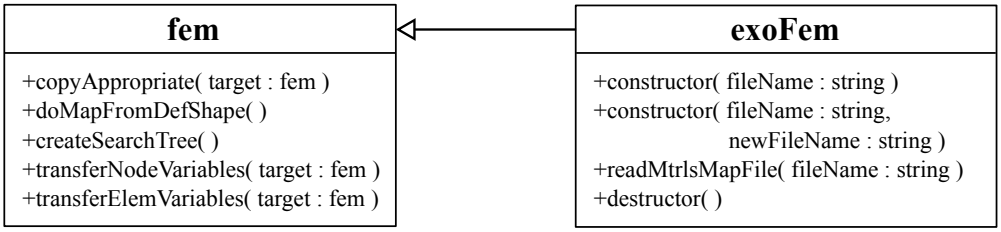

Figure 3.2. Class diagram of the finite element model data structure implemented for the mapping code.

and `doMapFromDefShape` methods are called. These three steps combined represent the **INITIALIZE TRANSFER** step in Figure 3.1. The first method reads further information about the model required for mapping that is not accessible in the EXODUS II database. This is further explained in Section 3.2.1. The next method, `copyAppropriate`, performs all of the steps of the transfer that do not require mapping. For EXODUS II databases, the `copyAppropriate` method copies the following components from the source to the target: QA records, information records, numbers of variables, variable truth tables, variable names, time steps, and global variables. The reader is referred to the EXODUS II manual [10] for descriptions of each component. The `doMapFromDefShape` method simply sets a flag indicating that the mapping is to be performed in the deformed configuration. This means that the displacements from the last time step stored in the source database will be applied prior to creating the spatial search tree and mapping variables. The target mesh is assumed to have been created on the equivalent deformed configuration.

The `createSearchTree` method is only called on the source `fem`. This method creates a member variable that stores a spatial search tree, which is described in Section 3.1.2.

The `transferNodeVariables` method is then called to map all node variables defined on the source mesh over to the target mesh. The call is made on the source `fem`, with the target `fem` as the input. The intention is to allow for multiple algorithms to be available for performing this mapping. Currently, however, only an inverse isoparametric mapping is available. For every node in the target, the search tree is called to get a list of source elements in the neighborhood of the target node. Each of these elements is then queried to determine which element actually contains the node. Also returned by this query are the source element's shape functions evaluated at the coordinates of the target node. The process for determining these element shape functions follows an inverse isoparametric mapping, *i.e.* Newton's method is applied to the isoparametric mapping,

$$X = \sum_{i=1}^{n} N_i(\xi, \eta, \zeta) X_i \tag{3.1}$$

to solve for the natural coordinates (ξ, η, ζ) of the global coordinates X, where n is the

number of nodes per element, N_i are the nodal shape functions evaluated at (ξ, η, ζ), and X_i are the nodal global coordinates. Nodal variables, U_i, are then transferred from source to target by the isoparametric mapping,

$$U = \sum_{i=1}^{n} N_i(\xi, \eta, \zeta) U_i \tag{3.2}$$

where U is the interpolated value at the global coordinates X. Note that this is an element-by-element mapping.

The `transferElemVariables` method is similar to `transferNodeVariables` method in both the way it is called and the type of mapping currently programmed – inverse isoparametric. Prior to mapping, however, an additional step is required to extrapolate element variables to the nodes. For every integration point in the target, the search tree and inverse isoparametric mapping are called to find the source element and its shape functions at the integration point's global coordinates. For each element variable, its values stored at the integration points of the source element are then extrapolated to that element's nodes. Element quantities that are stored on the source mesh as a volume-averaged value are assumed to be stored at one integration point, the centroid of the element, in both the source and target. The extrapolation procedure is further explained in Section 3.1.3. The extrapolated values are interpolated to the target integration points by Equation 3.2. Consideration of variables stored at multiple integration points within the element is a capability that is currently unavailable in MAPVAR, which treats all element variables as volume-averages. Specifically, the capability added here is a consideration of element variable gradients within each element. This is particularly important for mapping within regions known to have element field gradients within the elements, such as regions near crack fronts.

Admittedly, an element-by-element extrapolation procedure has its limitations, some of which are given in the MAPVAR manual [16]. The current method was chosen as a starting point; future research and development efforts are expected.

3.1.2 Search Tree

A high-level class diagram of the spatial search tree data structure is shown in Figure 3.3. The `femElemTree` is the class called and stored by the `createSearchTree` method of the `fem` class. Either of the two `femElemTree` constructors can be called to create a search tree. The first constructor builds a tree directly from the `fem` class instance. The second constructor builds a tree from vectors of: element identifiers, nodal connectivities of each element, and the global coordinates of each node. The latter constructor is required to create a search tree from the deformed configuration of a mesh, by passing in the deformed coordinates of each node. The first constructor only processes the model coordinates of each node. Each constructor creates bounding boxes of each element and stores them to an efficient search tree, the `adt`. The search tree is queried by passing a query point, *i.e.* the global coordinates

of a node or integration point in the target mesh, to the `query` method of the `femElemTree`. The `query` returns the ID's of all elements whose bounding boxes contain the query point.

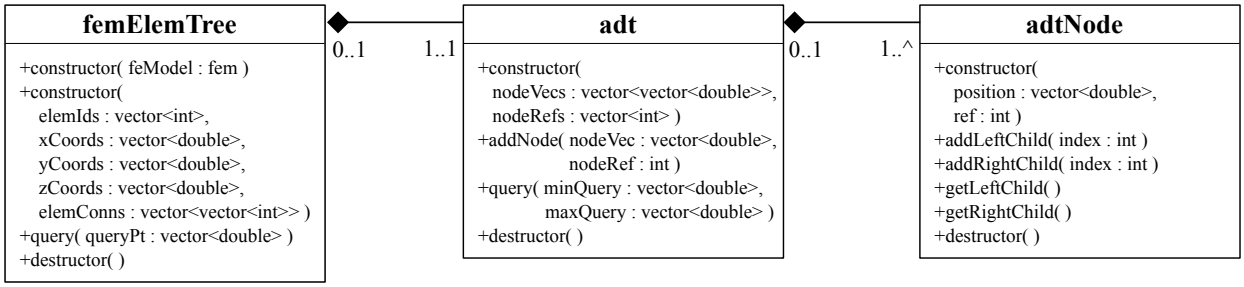

Figure 3.3. Class diagram of the spatial search tree data structure implemented for the mapping code.

The search tree algorithm adopted here is an alternating digital tree, following from the work of Bonet and Peraire [2]. This algorithm, when implemented for spatial searches, has a computational cost of $O(n(log(n)))$ where n is the number of items (elements). The methods shown in Figure 3.3 for the `adt` and `adtNode` classes are not described here since they follow the algorithms in Reference [2]. A `adtNode` instance is a node in the alternating digital tree, which represents the bounding box of a single element.

3.1.3 Element Types

A high-level class diagram of the data structure used to represent elements, and the various types thereof, is shown in Figure 3.4. Some of the methods shown for the `element` base class are defined or re-defined for each element type, but the details of which are skipped here for brevity. Each derived class is a singleton – they are called by the `instance` method instead of a `constructor`.

The following methods perform essentially as suggested by their names:

- `numNodes` returns the number of element nodes.
- `numGaussPts` returns the number of element integration points.
- `numModelCoordDims` returns the number of dimensions of the element, *i.e.* 2 or 3.
- `integrationWeights` returns the weights applied at each integration point.
- `gaussPtShapeFunction` returns the nodal shape functions evaluated at each integration point.

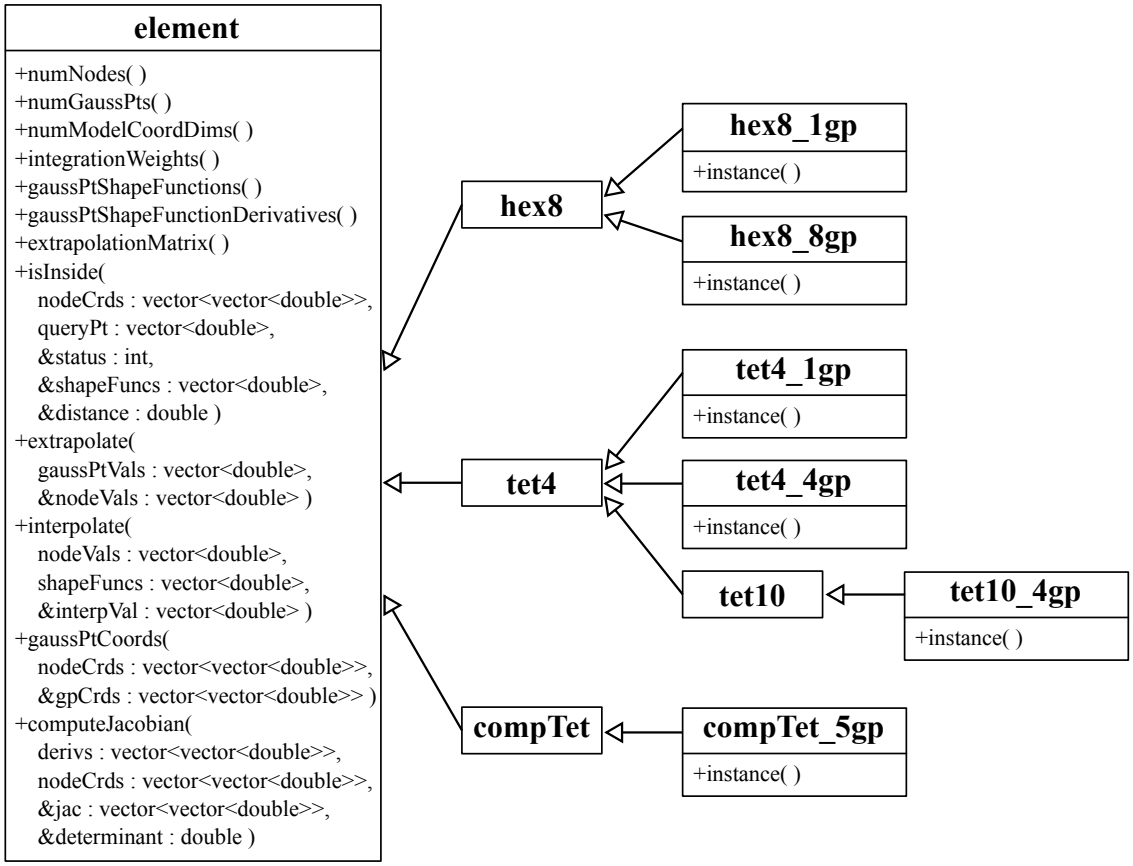

Figure 3.4. Class diagram of the element data structures implemented for the mapping code.

- `gaussPtShapeFunctionDerivatives` returns the nodal shape function derivatives evaluated at each integration point.
- `extrapolationMatrix` returns the extrapolation matrix. See subsequent discussion of the `extrapolate` method.
- `gaussPtCoords` returns the global coordinates of an element's gauss points for the input global nodal coordinates.

The `isInside` method performs the iterative routines for the inverse isoparametric mapping, solving Equation 3.1 for the natural coordinates (ξ, η, ζ). The inputs are the global coordinates of an element's nodes and the global coordinates of the query point. The routine returns a status binary, with 1 indicating that the query point is inside or on the surface of the element, and 0 indicating otherwise. Also returned by the routine are the shape function values at that query point and the distance from the element to the query point. This distance is zero if the query point is inside the element.

The `interpolate` method performs the interpolation by solving Equation 3.2 for given shape function values and nodal values.

The `extrapolate` method extrapolates element variables to the nodes. For volume-averaged element variables or single integration point elements, this is simply a copy of a scalar value to each of the nodes. For integration point variables stored in multiply integrated elements, an extrapolation matrix has to be constructed to project variables out to the nodes. Each element type has an extrapolation matrix which is returned by the `extrapolationMatrix` method. This matrix is constructed by virtually creating an element that has nodes at the natural coordinates of each of the real element's integration points. The matrix is the shape functions of this virtual element evaluated at each of the real element's nodes. Thus, the matrix is as follows,

$$E = \begin{bmatrix} N^v_{11} & N^v_{12} & \cdots & N^v_{1n} \\ N^v_{21} & N^v_{22} & \cdots & N^v_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ N^v_{m1} & N^v_{m2} & \cdots & N^v_{mn} \end{bmatrix} \tag{3.3}$$

where the superscript v indicates the virtual element, m is the number of element integration points, n is the number of element nodes, and N^v_{ji} is the jth shape function of the virtual element evaluated at the ith node of the real element. The extrapolation equation is then

$$U_i = \sum_{j=1}^{m} E_{ij} U_j \tag{3.4}$$

where U_j are the integration point values and U_i are the extrapolated values. Note, once again, that this is an element-by-element extrapolation, without averaging at or projection to the nodes. This technique can be less accurate than other techniques [16], but it is robust and easy to implement as an initial approach. A primary intention of this code is to have a framework in place for efficient implementation and research of mapping techniques that promise to be more accurate.

The `computeJacobian` method is used mostly by the Newton iterations performed in the `isInside` method, but it is also made public for computing some element variables, as explained in Section 3.2.2 and Section 3.2.3. This method computes the Jacobian matrix, J, and its determinant, $|J|$, for

$$J = \begin{bmatrix} \sum_{i=1}^{n} \frac{\delta N_i}{\delta \xi} x_i & \sum_{i=1}^{n} \frac{\delta N_i}{\delta \eta} x_i & \sum_{i=1}^{n} \frac{\delta N_i}{\delta \zeta} x_i \\ \sum_{i=1}^{n} \frac{\delta N_i}{\delta \xi} y_i & \sum_{i=1}^{n} \frac{\delta N_i}{\delta \eta} y_i & \sum_{i=1}^{n} \frac{\delta N_i}{\delta \zeta} y_i \\ \sum_{i=1}^{n} \frac{\delta N_i}{\delta \xi} z_i & \sum_{i=1}^{n} \frac{\delta N_i}{\delta \eta} z_i & \sum_{i=1}^{n} \frac{\delta N_i}{\delta \zeta} z_i \end{bmatrix} \tag{3.5}$$

where n is the number of nodes per element, $\frac{\delta N_i}{\delta \xi}$, $\frac{\delta N_i}{\delta \eta}$, and $\frac{\delta N_i}{\delta \zeta}$ are the partial derivatives of the nodal shape functions with respect to the natural coordinates evaluated at a point, and $(x_i, y_i, z_{i)}$ are the global nodal coordinates, X_i.

The following element types are currently programmed into the mapping code:

- `hex8_1gp`: The mean quadrature, 8-noded hexahedral element.
- `hex8_8gp`: The standard 8-point integration, 8-noded hexahedal element, which can be used for both the selective deviatoric and fully-integrated formulations.
- `tet4_1gp`: The mean quadrature, 4-noded tetrahedral element.
- `tet4_4gp`: The fully-integrated, 4-noded tetrahedral element.
- `tet10_4gp`: The fully-integrated, 10-noded tetrahedral element.
- `compTet_5gp`: The composite tetrahedral finite element formulation from reference [12]. Further explanation of this element, including motivation for including it in this study, is given in Section 3.2.3.

3.2 Unique Considerations

The implemented mapping code follows largely the aforementioned code structure. However, some modifications were necessary to properly handle the structure of the EXODUS II finite element model database and element variables that had to be computed rather than mapped. The following three sub-sections highlight these nuances.

3.2.1 Exodus II Databases

EXODUS II databases created from SIERRA/SM do not have a construct for integration point variables. Integration point variables are stored as an element variable. SIERRA/SM does use a consistent naming convention to indicate unique integration points, by adding an '`_i`' to the end of a variable name, where '`i`' is an integer representing the integration point. For example, the xx-component of the unrotated stress stored on an element with four integration points will have four separate variable names: `unrotated_stress_xx_1`, `unrotated_stress_xx_2`, `unrotated_stress_xx_3`, and `unrotated_stress_xx_4`. However, this nomenclature of adding '`_i`' at the end of a variable name is not unique to integration point variables. To overcome this ambiguity, the names of all volume-averaged element variables are stored in the `exoFem` class, using the nomenclature from SIERRA/SM. This obviously relies on the mapping code having a complete list of volume-averaged element variable names for the version of SIERRA/SM used during finite element analyses. Currently, this list is known to be complete for the 4.24 release version of SIERRA/SM and the following material models: elastic, multi-linear elastic plastic failure, and BCJ_mem.

Material model names are not required to be stored in an EXODUS II database, so the user must provide a separate text file for both the source and target meshes that indicate

the material model name and the number of integration points per element for each block in the mesh. The name of this text file is passed through the `readMtrlsMapFile` command, as shown in Figure 3.2. For example, if a mesh has two blocks, blocks 1 with 4-point integration and an elastic material model and block 11 with 8-point integration and a BCJ_mem material model, then the text file would have the following two lines:

```
1 4 ELASTIC
11 8 BCJ_MEM
```

A more permanent, robust solution should be implemented in the future to overcome this problem.

3.2.2 Element Volumes

One of the element variables stored in an EXODUS II database for a SIERRA/SM restart file is the volume of the element in the undeformed configuration. Mapping this variable is obviously incorrect. Instead, the target element volumes are computed as

$$V_{elem} = \sum_{j=1}^{m} w_j \, |J_j| \tag{3.6}$$

where m is the number of element integration points, w_j are the weights of each integration point, and $|J_j|$ are the determinants of the Jacobian matrices at each integration point computed by Equation 3.5. The public `element` class methods `computeJacobian` and `integrationWeights` (see Figure 3.4) are called to compute $|J_j|$ and w_j, respectively.

3.2.3 Composite Tetrahedral Elements

The composite tetrahedral element from Thoutireddy *et al.* [12] is implemented in this study because of its purported advantages. Previous discussions have emphasized that only tetrahedral meshes can be reliably generated for the arbitrary geometries of explicitly represented cracks. Higher order tetrahedral elements are preferred over simplicial 4-noded tetrahedra due to the volumetric locking of the latter. Furthermore, reference [12] suggests that the composite tetrahedral elements behave better in the near incompressible regime and have well-defined nodal masses as compared to simplicial 10-noded tetrahedral elements. All of the aforementioned tetrahedral element formulations are implemented herein with the future intent to investigate their relative performance in geometry adaptive, ductile fracture modeling.

Unlike the other commonly utilized element formulations implemented in SIERRA/SM, the composite tetrahedral element is implemented with the total Lagrangian approach[1].

[1]The authors would like to thank Jake Ostien for implementing the composite tetrahedral element in SIERRA/SM and giving us valuable insight into its formulation.

Therefore, the Jacobian determinant and the gradient operator are only computed once, during initialization, and then stored as element variables. For restarts, these variables are read from the database rather than being recomputed. Therefore, the mapping code must be responsible for supplying the correct values for these variables in the target mesh. Obviously, mapping is incorrect for both of these variables, so they must be computed for each target element. The Jacobian determinant is already available from the element volume calculation, as described in the previous section. The gradient operator, $\bar{L}_{aJ}(X)$ follows from [12],

$$\bar{L}_{aJ}(X) = \sum_{b=1}^{4}\sum_{c=1}^{4} \lambda_c(X) M_{cb}^{-1} \int_{\Omega_0} \lambda_b N_{a,J} dV_0 \tag{3.7}$$

where λ_a is the barycentric coordinates of integration point a, Ω_0 is the undeformed domain of the element, $N_{a,J}$ are the shape function derivatives with respect to the global coordinates, and M is a mass-like matrix evaluated by

$$M_{bc} = \int_{\Omega_0} \lambda_b \lambda_c dV_0; \tag{3.8}$$

3.3 Examples

The mapping code is intended for ongoing research and development to properly address, among other applications, mapping for geometry adaptive crack modeling. In its current state, the code is functioning for some common applications, as shown in the following three examples. All analyses were performed by SIERRA/SM.

3.3.1 Mapping in the Undeformed Configuration

One common application is arbitrary mesh refinement, where the modeler is required to create a new mesh that is structured independently from the source mesh. Figure 3.5 shows two examples of this application where the mapping is performed in the undeformed configuration. The model, as shown in Figure 3.5(a), is two cubes, each with a length $L = 1$. The top cube is elastic, isotropic with a Young's modulus $E_1 = 10000$ and a Poisson's ratio $\nu = 0.33$. The bottom cube is also elastic, isotropic with a Young's modulus $E_2 = 100000$ and a Poisson's ratio $\nu = 0.33$. The model is loaded in uniaxial tension in the y-direction to 25% strain, and then the mapping is performed. The first example, Figure 3.5(b) and (c), is a mapping between meshes with selective deviatoric hexes – 8 nodes, 8 integration points. The second example, Figure 3.5(d) and (e), is a mapping between meshes with composite tetrahedral elements – 10 nodes, 5 integration points. Mapping to (and from) 10-noded tetrahedral elements is important for accurately modeling complex evolving geometries, such as cracks, and it is a capability that is unavailable in MAPVAR. Nodal y-displacements are shown for both examples. Mapping on the undeformed configuration works well for smaller

deformations, but as the region becomes highly deformed, it becomes increasingly difficult, if not impossible, to create a new target mesh on the undeformed body that has desired element quality in the deformed configuration. In these two examples, it is evident that the target mesh elements are distorted in the y-direction in the deformed configuration. One can logically conclude that mapping in the undeformed configuration will not suffice for modeling the large deformations near the crack front in many ductile fracture problems. This motivates remeshing and mapping on the deformed configuration, as shown in the next example.

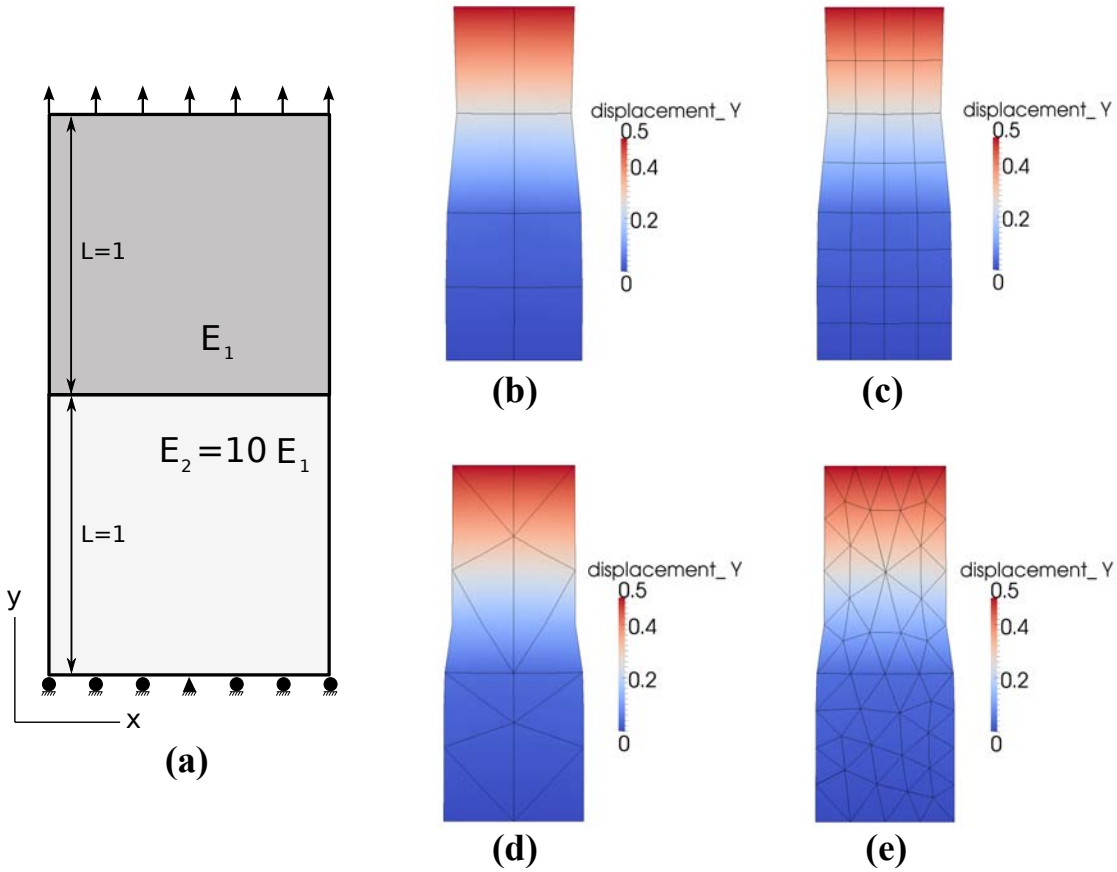

Figure 3.5. Example of nodal variable mapping in the undeformed configuration. (a) Illustration of model setup. (b) Computed displacements on source mesh with selective deviatoric hexahedral elements. (c) Mapped displacements on target mesh with selective deviatoric hexahedral elements. (d) Computed displacements on source mesh with composite tetrahedral elements. (e) Mapped displacements on target mesh with composite tetrahedral elements.

3.3.2 Mapping in the Deformed Configuration

This example, shown in Figure 3.6, is the same problem setup as the previous examples, but here remeshing and mapping are performed in the deformed configuration. The selective deviatoric hexahedral element is shown in this example. A comparison of this mapping, Figure 3.6(b), to mapping on the undeformed configuration, Figure 3.5(e), immediately reveals two differences:

1. The elements for the deformed configuration mapping are less distorted in the y-direction, because the model was remeshed in the deformed configuration; and,
2. The displacements are zero for the deformed configuration mapping. For obvious reasons, the mapping code does not map over nodal displacements when the mapping is performed on the deformed configuration[2]. All other variables, although not shown here, are mapped.

Figure 3.6(c) shows that the target mesh was restarted and loaded out to an additional 25% strain. A hypo-elastic material model was used for all examples shown here, so this problem demonstrates the ability to successfully continue an analysis after mapping history-dependent variables. Similar success was achieved for this problem setup with the multi-linear elastic-plastic failure (MLEPF) and BCJ_mem material models.

3.3.3 Mapping Element Variables at Integration Points

There are many applications that have stress gradients within elements, such as the beam-bending example shown in Figure 3.7. The model, illustrated in Figure 3.7(a), is a rectangular beam, with length $L = 10$, and equal base and height, $b = h = 1$. The material is elastic, isotropic with a Young's modulus $E = 100000$ and a Poisson's ratio $\nu = 0.33$, and the element type is the selective deviatoric hexahedron – 8 nodes, 8 integration points. The model is fixed at one end and displaced downward at the other, to $\Delta_y = 0.1$ when mapping is performed. Figure 3.7(b) shows the xx-component of the unrotated stress on the source mesh, and Figure 3.7 (c) shows the mapping of this solution onto the target mesh. For plotting purposes, the unrotated stress values at each integration point within an element were averaged by a filtering tool in the visualization software; the mapping actually operated on individual integration points. The target mesh shown here was created by dividing each source element into eight equally sized elements. Close inspection of regions with sharp stress gradients reveals that the mapped solution varies among target elements that lie within the same source element. This results directly from stress gradients within the source elements.

[2]The user must beware that boundary conditions applied in their restart immmediately after mapping should be updated accordingly to reflect that the displacements begin from zero.

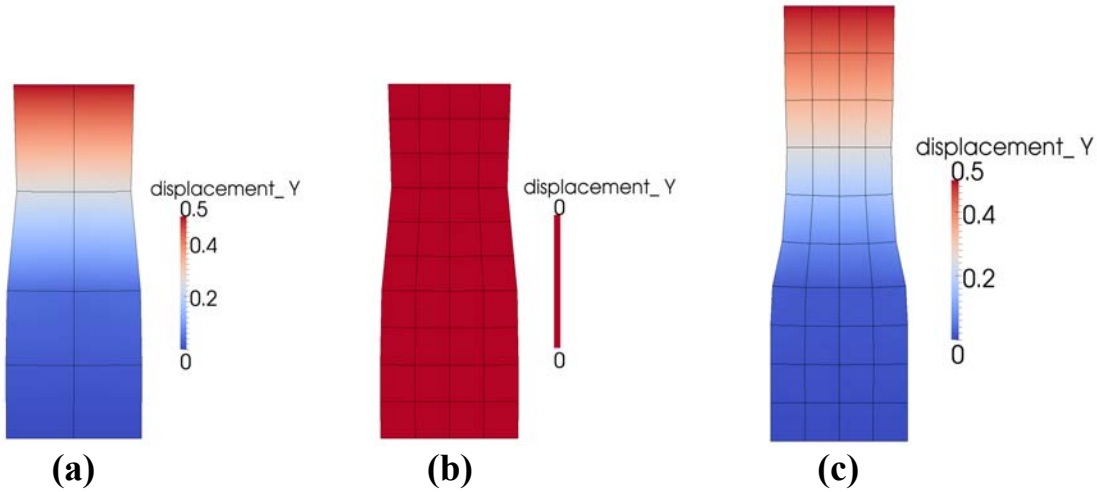

Figure 3.6. Example of nodal variable mapping in the deformed configuration. (a) Computed displacements on source mesh with selective deviatoric hexahedral elements. (b) Mapped displacements on target mesh with selective deviatoric hexahedral elements. (c) Computed displacements after further loading subsequent to mapping.

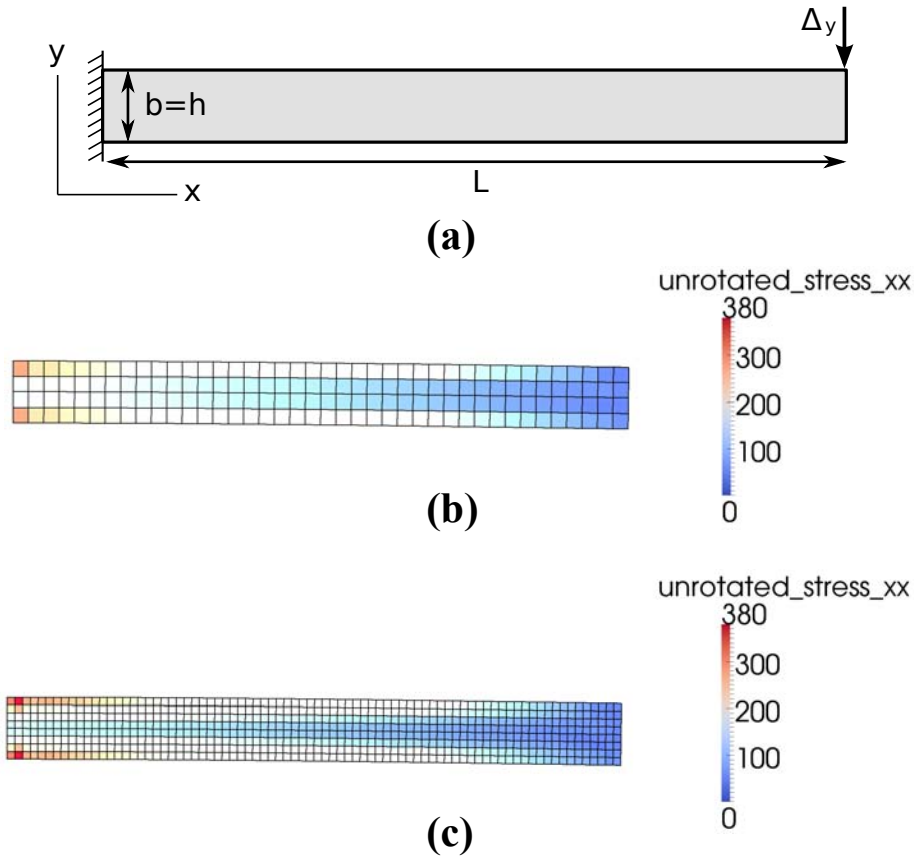

Figure 3.7. Example of element variable mapping from elements with multiple integration points. (a) Illustration of model setup. (b) Computed xx-component of the unrotated stress on the source mesh with selective deviatoric hexahedral elements. (c) Mapped xx-component of the unrotated stress on the target mesh with selective deviatoric hexahedral elements.

Chapter 4

Summary and Future Work

This work is motivated by the need to accurately quantify the cause and uncertainty of failure for applications that involve the nucleation and propagation of dominant cracks. Two capabilities have been developed: a semi-automated, geometry adaptive crack modeling procedure; and, a variable mapping code designed for large deformations and element field gradients.

The geometry adaptive crack modeling capability is implemented as a loose coupling between a commercial geometry adaptive crack representation code, FRANC3D, and the SIERRA/SM finite element analysis code. A FRANC3D application is kept in memory during incremental calls to the finite analysis code to evolve fields that drive crack evolution. Finite element model data are transferred to and from the FRANC3D application by accessing EXODUS II databases. FRANC3D updates the geometry of a model and remeshes to represent the nucleation and evolution of arbitrarily non-planar crack surfaces. The accomplishments of this project for geometry adaptive crack modeling procedure include:

1. An interface, all in memory, of FRANC3D with EXODUS II databases.
2. Extension of the crack growth capabilities in FRANC3D to correctly compute crack growth increments and directions for models that do not have FRANC3D's standard template of crack front elements since the template cannot be modeled in SIERRA/SM.
3. A semi-automated, scripting interface for geometrically modeling crack evolution, from nucleation to failure.
4. Demonstration of capabilities for examples with well-known crack paths and for an example of a potential user application.

This approach is currently ready for some user applications, such as brittle fracture modeling. However, future work is still intended, including:

1. Research and development of methods for accurately predicting crack nucleation and propagation in both the brittle and ductile fracture regimes. The implemented geometry adaptive approach should aid to improve calibration and validation for such R&D efforts.

2. Inclusion of strain localization elements with physically justified damage models along geometrically adapted crack paths.

3. Improvement of the ability to control mesh size at the crack front and evaluation of appropriate mesh sizes for convergence.

4. Full coupling between the geometry adaption algorithms and the finite element analysis.

5. Evaluation and extension of the approach's limits, *e.g.* crack branching and coalescense.

6. Development of the ability to map fields after crack adaption and continue an analysis with history-dependent material response.

The latter is known as one of the largest, most risky tasks to address for this modeling approach.

Accordingly, this project also focused on variable mapping for history-dependent materials. A new variable mapping procedure has been implemented that is motivated by the need to overcome some of the limitations of existing mapping capabilities. The current procedure is structured to:

1. Interface with multiple finite element model formats, with the proof-test format being EXODUS II.

2. Perform efficient spatial searches via an Alternating Digital Tree (ADT) algorithm.

3. Operate on an entire suite of element types. Current element types include: mean quadrature, fully-integrated, and selective deviatoric 8-noded hexahedra; mean quadrature and fully-integrated 4-noded tetrahedra; and, fully-integrated and composite 10-noded tetrahedra.

4. Map field gradients within elements by mapping element variables to and from the appropriate integration points.

5. Map on both the deformed and undeformed configurations.

6. Be easily adaptable for research, development, and testing of multiple mapping approaches. The initial approach is an inverse isoparametric mapping from the nodes; element variables are first extrapolated to the nodes.

Several areas of future work are intended for variable mapping, including:

1. Proper mapping of discontinuous fields, *e.g.* displacements, at crack surfaces.

2. Development and testing of more robust mapping approaches.

3. Mapping between meshes with dissimilar element types.

4. Demonstration of capabilities for geometry adaptive crack modeling.
5. Full coupling with crack adaption and finite element analysis procedures.

References

[1] D.J. Bamman, M.L. Chiesa, and G.C. Johnson. Modeling large deformation and failure in manufacturing processes. In T. Tatsumi, E. Watanabe, and T. Kambe, editors, *Proceedings of the 19th International Congress of Theoretical and Applied Mechanics*, pages 359–376, Amsterdam, 1997. Elsevier Science Publishers.

[2] J. Bonet and J. Peraire. An alternating digital tree (ADT) algorithm for 3D geometric searching and intersection problems. *International Journal for Numerical Methods in Engineering*, 31:1–17, 1991.

[3] H.C. Edwards and J.R. Stewart. SIERRA: A software environment for developing complex multi-physics applications. In K.J. Bathe, editor, *First MIT Conference on Computational Fluid and Solid Mechanics*, pages 1147–1150, Amsterdam, 2001. Elsevier Science Publishers.

[4] Python Software Foundation. Python 2.7.1, April 2011.

[5] D.K. Gartling. MERLIN – A computer program to transfer between finite element meshes. Technical Report SAND81-0463, Sandia National Laboratories, Albuquerque, NM, 1981.

[6] D.K. Gartling. MERLIN – A computer program to transfer between finite element meshes. Technical Report SAND89-2989, Sandia National Laboratories, Albuquerque, NM, 1991.

[7] A.R. Ingraffea. Computational fracture mechanics. In E. Stein, R. de Borst, and T.J.R. Hughes, editors, *Encyclopedia of Computational Mechanics*, volume 2. John Wiley & Sons, Ltd., 2007.

[8] L.F. Martha, P.A. Wawrzynek, and A.R. Ingraffea. Arbitrary crack representation using solid modeling. *Engineering with Computers*, 9:63–82, 1993.

[9] G.D. Sjaardema, L.A. Schoof, and A.R. Yarberry. EXODUS II 5.22, August 2012. Sandia National Laboratories.

[10] G.D. Sjaardema, L.A. Schoof, and V.R. Yarberry. EXODUS II – A finite element data model. Technical Report SAND92-2137, Sandia National Laboratories, Albuquerque, NM, November 2006. Draft.

[11] SIERRA Solid Mechanics Team. SIERRASolidMechanics 4.24, March 2012. Sandia National Laboratories.

[12] P. Thoutireddy, J.F. Molinari, E.A. Repetto, and M. Ortiz. Tetrahedral composite finite elements. *International Journal for Numerical Methods in Engineering*, 53:1337–1351, 2002.

[13] P.A. Wawrzynek, B.J. Carter, and A.R. Ingraffea. FRANC3D Command Language & Python Extensions, Version 6, November 2011. Fracture Analysis Consultants, Inc.

[14] P.A. Wawrzynek, B.J. Carter, and A.R. Ingraffea. FRANC3D Reference Manual, Version 6, November 2011. Fracture Analysis Consultants, Inc.

[15] P.A. Wawrzynek, B.J. Carter, and A.R. Ingraffea. FRANC3D V7.0(beta), August 2012. Fracture Analysis Consultants, Inc.

[16] G.W. Wellman. MAPVAR – A computer program to transfer solution data between finite element meshes. Technical Report SAND99-0466, Sandia National Laboratories, Albuquerque, NM, March 1999.

[17] Q. Yang, A. Mota, and M. Ortiz. A class of variational strain-localization finite elements. *International Journal for Numerical Methods in Engineering*, 62:1013–1037, 2005.

DISTRIBUTION:

1	MS 0372	N. Breivik, 1524 (electronic copy)
1	MS 9042	A. Brown, 8259 (electronic copy)
1	MS 0346	R. Chambers, 1526 (electronic copy)
1	MS 9042	M. Chiesa, 8259 (electronic copy)
1	MS 9042	S. English, 8259 (electronic copy)
1	MS 0372	E. Fang, 1524 (electronic copy)
1	MS 9042	M. Gonzales, 8250 (electronic copy)
1	MS 1321	G. Hansen, 1444 (electronic copy)
1	MS 0380	M. Heinstein, 1542 (electronic copy)
1	MS 9042	J. Foulk, 8256 (electronic copy)
1	MS 0380	J. Jung, 1542 (electronic copy)
1	MS 9042	T. Kostka, 8259 (electronic copy)
1	MS 9042	C. Moen, 8256 (electronic copy)
1	MS 9404	A. Mota, 8256 (electronic copy)
1	MS 1398	M. Neilsen, 1526 (electronic copy)
1	MS 9042	J. Ostien, 8256 (electronic copy)
1	MS 0380	K. Pierson, 1542 (electronic copy)
1	MS 0840	D. Rader, 1513 (electronic copy)
1	MS 0346	D. Reedy, 1524 (electronic copy)
1	MS 0380	D. Womble, 1540 (electronic copy)
1	MS 0899	Technical Library, 8944 (electronic copy)
1	MS 0359	D. Chavez, LDRD Office, 1911

Sandia National Laboratories