OBJECT ORIENTED PLATFORM FOR SEISMIC ANALYSIS OF
PRECAST CONCRETE BUILDINGS

A Thesis in
Civil Engineering
by
Serdar Astarlioglu

© 2005 Serdar Astarlioglu

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2005
The thesis of Serdar Astarlioglu has been reviewed and approved* by the following:

Andrew Scanlon  
Professor of Civil Engineering  
Thesis Adviser  
Chair of Committee  
Head of the Department of Civil and Environmental Engineering

Ali M. Memari  
Associate Professor of Architectural Engineering

Andrea Schokker  
Associate Professor of Civil Engineering

Ashok D. Belegundu  
Professor of Mechanical Engineering

Maria Lopez de Murphy  
Assistant professor of Civil Engineering

*Signatures are on file in the Graduate School
Abstract

The development of finite element analysis program named SNAPS3D (Seismic Nonlinear Analysis of Precast Structures) for nonlinear analysis of precast concrete structures under seismic loads is described. The advantages of the Microsoft .NET Framework and C# programming language, which are used to develop the source code, over traditional procedure-based FORTRAN finite elements in terms of maintainability and extensibility of software is discussed. Since .NET Framework does not contain out-of-the-box numerical libraries, the classes for matrix and vector operations are developed and packaged in an assembly named Snaps3D.Algebra. The finite element concepts such as nodes, elements, solvers, and loads are abstracted into C# types (classes and interfaces) and packaged in an assembly named Snaps3D.Core. Included in this assembly are specialized classes to model precast concrete members (shell element to model wall panels and floors, beam element to model beams, columns, and horizontal connections) and their behavior (elastic material to model the response of precast members and nonlinear materials to model the connections). New material models, elements, solvers can be developed and incorporated into the program either by implementing the appropriate interface (IElement interface to add a new element class) or by inheriting from an existing class with similar behavior (Inherit from Solid class to develop Shell element). Neither case would require the existing program to be modified. The program is used to solve several example problems to verify its operation and to
demonstrate how precast building components can be modeled using the element and matrix classes that are currently implemented in SNAPS3D.
# Table of Contents

List of Figures ...................................................................................................................... x  
List of Tables .......................................................................................................................... xvi  
Acknowledgements .............................................................................................................. xvii  
Chapter 1. Introduction .......................................................................................................... 1  
  1.1 General .......................................................................................................................... 1  
  1.2 Objective and Scope ...................................................................................................... 3  
  1.3 Thesis Layout .............................................................................................................. 4  
Chapter 2. Precast Concrete Buildings .............................................................................. 5  
  2.1 General ........................................................................................................................ 5  
  2.2 Performance of Precast Concrete Buildings in Earthquakes ....................................... 5  
  2.3 Precast Concrete Shear Wall Systems ....................................................................... 8  
     2.3.1 Building Configurations ......................................................................................... 8  
     2.3.2 Shear Wall Configurations .................................................................................. 10  
     2.3.3 Shear Wall Connection Details .......................................................................... 13  
     2.3.4 Behavior of Shear Walls ....................................................................................... 15  
     2.3.5 Code Provisions .................................................................................................... 19  
  2.4 Precast Concrete Frame Systems ............................................................................... 26  
     2.4.1 Details of Ductile Connections ............................................................................ 28  
     2.4.2 Behavior of Precast Concrete Frames with Ductile Connections .................... 29  
     2.4.3 Emulative Design of Precast Concrete Frames ..................................................... 36  
  2.5 Concluding Remarks ................................................................................................... 40  
Chapter 3. Snaps3D: A Platform for Analysis of Precast Concrete Structures .................. 41
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 General</td>
<td>41</td>
</tr>
<tr>
<td>3.2 Description of Base Platform</td>
<td>42</td>
</tr>
<tr>
<td>3.2.1 The .NET Framework</td>
<td>42</td>
</tr>
<tr>
<td>3.2.2 C# Programming Language</td>
<td>51</td>
</tr>
<tr>
<td>3.2.3 Structure of .NET Applications</td>
<td>53</td>
</tr>
<tr>
<td>3.3 Snaps3D.Algebra Assembly</td>
<td>54</td>
</tr>
<tr>
<td>3.3.1 Concept Description</td>
<td>54</td>
</tr>
<tr>
<td>3.3.2 DoubleVector Class</td>
<td>56</td>
</tr>
<tr>
<td>3.3.3 DoubleMatrix Class</td>
<td>56</td>
</tr>
<tr>
<td>3.3.4 DiagonalDoubleMatrix Class</td>
<td>58</td>
</tr>
<tr>
<td>3.3.5 BoolVector Class</td>
<td>59</td>
</tr>
<tr>
<td>3.3.6 ObjectVector Class</td>
<td>59</td>
</tr>
<tr>
<td>3.3.7 ObjectMatrix Class</td>
<td>59</td>
</tr>
<tr>
<td>3.3.8 DiagonalObjectMatrix Class</td>
<td>60</td>
</tr>
<tr>
<td>3.4 Snaps3D Core Assembly</td>
<td>60</td>
</tr>
<tr>
<td>3.4.1 Concept Description</td>
<td>65</td>
</tr>
<tr>
<td>3.4.2 Interfaces</td>
<td>65</td>
</tr>
<tr>
<td>3.4.2.1 IMember Interface</td>
<td>66</td>
</tr>
<tr>
<td>3.4.2.2 IInteractiveMember Interface</td>
<td>68</td>
</tr>
<tr>
<td>3.4.2.3 IMaterial Interface</td>
<td>68</td>
</tr>
<tr>
<td>3.4.2.4 IIntegrationPoint Interface</td>
<td>70</td>
</tr>
<tr>
<td>3.4.2.5 ISection Interface</td>
<td>72</td>
</tr>
<tr>
<td>3.4.2.6 IElememt Interface</td>
<td>72</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>3.4.2.7</td>
<td>ISolver Interface</td>
</tr>
<tr>
<td>3.4.2.8</td>
<td>ITraceable Interface</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Classes</td>
</tr>
<tr>
<td>3.4.3.1</td>
<td>Member Class</td>
</tr>
<tr>
<td>3.4.3.2</td>
<td>Domain Class</td>
</tr>
<tr>
<td>3.4.3.3</td>
<td>InteractiveMember Class</td>
</tr>
<tr>
<td>3.4.3.4</td>
<td>Node Class</td>
</tr>
<tr>
<td>3.4.3.5</td>
<td>Boundary Class</td>
</tr>
<tr>
<td>3.4.3.6</td>
<td>LumpedMass Class</td>
</tr>
<tr>
<td>3.4.3.7</td>
<td>Material Class</td>
</tr>
<tr>
<td>3.4.3.8</td>
<td>IntegrationPoint Class</td>
</tr>
<tr>
<td>3.4.3.9</td>
<td>Section Class</td>
</tr>
<tr>
<td>3.4.3.10</td>
<td>Element Class</td>
</tr>
<tr>
<td>3.4.3.11</td>
<td>Loading Class</td>
</tr>
<tr>
<td>3.4.3.12</td>
<td>Solver Class</td>
</tr>
<tr>
<td>3.5</td>
<td>Summary</td>
</tr>
<tr>
<td>4.1</td>
<td>General</td>
</tr>
<tr>
<td>4.2</td>
<td>Review of Previous Models</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Shear Walls</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Frame Models</td>
</tr>
<tr>
<td>4.3</td>
<td>Element Models</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Solid Element</td>
</tr>
</tbody>
</table>
List of Figures

Figure 2.1 Plan of a cross-wall system (Precast/Prestressed Concrete Institute 1999) ................................................................. 9

Figure 2.2 Elevation (longitudinal) (Precast/Prestressed Concrete Institute 1999) ................................................................................................................................. 9

Figure 2.3 Elevation (transverse) (Precast/Prestressed Concrete Institute 1999) ................................................................. 9

Figure 2.4 Simple shear wall ................................................................................................................................. 11

Figure 2.5 Coupled shear wall ................................................................................................................................. 11

Figure 2.6 Shear wall with vertical connections ......................................................................................................................... 12

Figure 2.7 Platform connection ................................................................................................................................. 14

Figure 2.8 Platform connection (at exterior wall) ......................................................................................................................... 14

Figure 2.9 Rocking and slip behavior of a simple wall ................................................................................................. 16

Figure 2.10 Seismic response of a coupled shear wall ................................................................................................. 18

Figure 2.11 Seismic response of a wall system with vertical connections ................................................................. 18

Figure 2.12 Unbonded post-tensioned wall with vertical connections (Stanton and Nakaki 2002) ......................................................................................................................... 20

Figure 2.13 Seismic response of an unbonded post-tensioned wall with vertical connections (Stanton and Nakaki 2002) ......................................................................................................................... 20

Figure 2.14 Connection with welded reinforcement ......................................................................................................................... 22

Figure 2.15 Connection with mechanical splice ......................................................................................................................... 22

Figure 2.16 Connection with mechanical splice (at exterior wall). ......................................................................................................................... 23

Figure 2.17 Test sequence for precast wall modules ......................................................................................................................... 25

Figure 2.18 Acceptable hysteretic behavior ......................................................................................................................... 27
Figure 4.2 Shear-slip element force-displacement relationship (Schricker and Powell 1980)................................. 101
Figure 4.3 Gap element dof (Schricker and Powell 1980)......................... 102
Figure 4.4 Gap element force-displacement relationship (Schricker and Powell 1980)............................................. 102
Figure 4.5 Concrete stress-strain model (Kianoush and Scanlon 1986)......... 105
Figure 4.6 Frame sub-assembly model with spring elements (El-Sheikh et al. 1999)................................................................. 107
Figure 4.7 Frame sub-assembly model with fiber elements (El-Sheikh et al. 1999)................................................................. 107
Figure 4.8 Fiber beam-column element for DRAIN2DX (Powell 1993)........ 109
Figure 4.9 Class inheritance diagram for Element classes......................... 110
Figure 4.10 Solid element with eight nodes........................................... 112
Figure 4.11 Shell element with eight nodes........................................... 116
Figure 4.12 Beam element with eight nodes ......................................... 119
Figure 4.13 Axial behavior of linear-elastic material ............................ 121
Figure 4.14 Shear behavior of linear-elastic material ............................ 121
Figure 4.15 Axial behavior of rebar material ........................................ 124
Figure 4.16 Axial behavior of concrete material .................................... 129
Figure 4.17 Shear-slip behavior of concrete ........................................ 130
Figure 5.1 Finite element mesh of L-shaped beam (Chandrupatla and Belegundu 1997) .......................................................... 133
Figure 5.2 Displacement history with different damping coefficients. ......... 135
Figure 5.3 Abaqus model of the wall in Example 2 .................................................. 136
Figure 5.4 Roof displacement history ........................................................................ 138
Figure 5.5 Shear wall subassembly with horizontal connection .......................... 139
Figure 5.6 Finite element mesh of the shear wall .................................................... 142
Figure 5.7 Comparison of elastic and nonlinear connection materials ............. 143
Figure 5.8 Applied force vs. roof displacement ..................................................... 144
Figure 5.9 Hysteresis at connection integration point ........................................... 146
Figure 5.10 Displacement history of cantilever beam .......................................... 148
Figure 5.11 Mesh of the 15-story building model (Kianoush and Scanlon 1986) ......................................................................................................................... 150
Figure 5.12 Displacement history of 15-story building model .............................. 151
Figure 5.13 Isometric view of U-shaped shear wall building ............................... 153
Figure 5.14 Plan view of U-shaped shear wall building ....................................... 154
Figure 5.15 Elevation of U-shaped shear wall building ....................................... 154
Figure 5.16 Loading case 1: Triangular pulse in x-direction .............................. 156
Figure 5.17 Loading case 2: General seismic load in x and y-directions ......... 156
Figure 5.18 Displacement history of northeast tip of the roof under the pulse 157
Figure 5.19 Deformed shape of Wall 1 at t=100 ms ........................................... 157
Figure 5.20 Deformed shape of Wall 1 at t=120 ms ........................................... 158
Figure 5.21 Deformed shape of Wall 1 at t=300 ms ........................................... 158
Figure 5.22 Axial vs. shear strain history of the east end of Wall 1 at base 159
Figure 5.23 Displacement history of the roof under seismic load in x-direction 161
Figure 5.24 Displacement history of the roof under seismic loading in y-direction.
................................................................................................................................. 161

Figure B.1 Input screen for L-Shaped beam analyzed in Chapter 5. ................. 177
List of Tables

Table 3.1 The components of the .NET Framework ................................................. 43
Table 3.2 C# class members ...................................................................................... 52
Table 3.3 Partial list of C# access modifiers ............................................................... 52
Table 3.4 Member class members .............................................................................. 79
Table 3.5 Domain class members .............................................................................. 81
Table 3.6 InteractiveMember class members ............................................................. 81
Table 3.7 Node class members ................................................................................. 83
Table 3.8 Boundary class members .......................................................................... 84
Table 3.9 LumpedMass class members .................................................................... 85
Table 3.10 Material class members .......................................................................... 87
Table 3.11 IntegrationPoint class members ............................................................... 88
Table 3.12 Section class members ............................................................................. 91
Table 3.13 Element class members .......................................................................... 93
Table 3.14 Loading class members .......................................................................... 96
Table 3.15 Solver class members .............................................................................. 96
Table A.1 System.Object class members ................................................................. 175
Table A.2 System.MarshalByRefObject class members .......................................... 176
Acknowledgements

This thesis has been a very long journey for me.

Although many things have changed, including the very topic, one thing remained constant: Dr. Scanlon’s support and confidence in me. Without him, this thesis probably would never have been completed. It is an honor for me to express my gratitude to my thesis advisor Dr. Andrew Scanlon for his belief in my actions and approaches related to my thesis throughout this research. As a great role model, he always provided me encouragement and inspired me numerous times with his valuable guidance.

During my studies, I also had the privilege to work with Dr. Ali Memari, whom I also would like to express my gratitude for motivating me to find new directions in this research along with providing me valuable comments.
Chapter 1

Introduction

1.1 General

Precast concrete has been successfully used in variety of building structures including residential buildings, office buildings, and parking garages. Compared to cast-in-place reinforced concrete, precast concrete can be erected much faster, has lower on-site labor requirements, requires little or no formwork, and offers better quality control because of plant production of precast components in a controlled environment.

Currently, the only design approach that is validated by building codes for construction of precast buildings in regions of moderate to high seismic risk is the “emulative approach”. The emulative approach requires connections that join the precast members to be stronger and stiffer than the precast members themselves. This approach assumes that by using stronger and stiffer connections a precast concrete structure would deform in a manner similar to a cast-in-place reinforced concrete structure during a destructive earthquake. Therefore, the connections would not exhibit a nonlinear behavior that is exhibited in “ductile connections”.

1
The “ductile connections” are typically weaker and more flexible than precast members that they join and they are not permitted in regions of moderate to high seismic risk under the current building codes. This is due to the lack of sufficient information on behavior of precast structures with such connections under three-dimensional seismic loads. The majority of the available experimental data is limited to two-dimensional seismic loads based on laboratory tests where only in-plane loading cases are considered and the computer programs, such as DRAIN2D (Kanaan and Powell 1973), are used for correlation studies or for testing new connection concepts. These programs are typically coded in FORTRAN using procedure based programming approach and adding new modeling capabilities in the form of new materials or elements is very difficult. Using the .NET Framework and the C# programming language introduced by Microsoft in 2002 (Microsoft 2005), it is possible to develop a new finite element architecture, which is very flexible and easy to maintain, using object-oriented programming approach. Because the .NET Framework supports multiple programming languages, new modeling capabilities for this finite element architecture can be done in any .NET compliant programming language and can be used without having to recompile the original source code. The C# programming language provides new features such as the support for XML code comments, Unicode (ability to use non-ASCII characters in code), and jagged arrays that are not available in FORTRAN or C++.
1.2 Objective and Scope

Finite element analysis software used to study the behavior of precast concrete buildings, such as DRAIN2D family of programs, are based on programming languages and practices that are outdated and represent a great challenge to researchers who want to extend the functionality of these software by adding new materials, elements, or both. The objective of this study is to develop a set of finite element class libraries suited for analysis of precast concrete buildings utilizing software technologies that result in computer code that is maintainable and extensible. The contribution of this study to the Civil Engineering profession will be a flexible and extensible analytical tool to aid in assessing of performance of precast concrete components and structures under seismic loads.

The scope of this study is summarized as follows:

1. Perform a review of precast concrete building systems and their components.

2. Develop a finite element analysis platform that is easy to maintain and modify using state-of-the-art technologies for distributed software development.

3. Develop a variety of finite elements and materials to model precast concrete members and their connections.

4. Demonstrate the application of the program using various examples.
1.3 Thesis Layout

This thesis is divided into six chapters. The objective and the scope of this study are described in Chapter 1. Configuration of common precast concrete shear wall and moment frame type structures, their connections, and the current available information on their performance in earthquakes are reviewed in Chapter 2 along with code provisions governing the design of such structures.

Chapter 3 describes the computer program SNAPS3D - the acronym for Seismic Nonlinear Analysis of Precast Structures. The chapter starts with a brief introduction to the Microsoft .NET Framework and C# programming language, which are used to develop SNAPS3D. The description of all base components, their purpose, and examples on their use are also presented in this chapter.

Chapter 4 discusses the techniques used to model the behavior of precast structures. The specialized SNAPS3D elements and material classes, developed to model various precast concrete building components, are described.

Chapter 5 presents several examples used to verify the program and demonstrate its application on precast concrete structures.

Finally, Chapter 6 summarizes the major conclusions reached from the study and outlines the recommendations for future study.
Chapter 2

Precast Concrete Buildings

2.1 General

This chapter reviews literature related to details and behavior of precast concrete buildings. In Section 2.2, a summary of the observations on behavior of precast concrete buildings under severe earthquakes is presented. Sections 2.3 and 2.4 describe the details and behavior precast concrete shear wall and moment frame systems, respectively. Available experimental data is used to highlight the response of these types of buildings and the connections utilized in precast concrete buildings under dynamic loading. A review of the code provisions governing the design of these buildings is also provided.

2.2 Performance of Precast Concrete Buildings in Earthquakes

The performance of a structural system under seismic loads depends on many factors. While the type of construction (precast concrete, cast-in-place concrete, steel, or masonry) plays an important role; it does not fully explain how the structure will perform under an earthquake. Other important factors are the distance of the building to epicenter, the underlying soil type, and the dominant period of the earthquake and its interaction with the structure. The framing
system also plays a very important role since it affects the period and the ductility of the structure. Below are the major earthquakes that put the precast concrete structures to the ultimate test in the last thirty years.

The 1977 earthquake of Bucharest, Romania, which measured 7.2 on Richter scale, was the first significant seismic event to test the performance of precast concrete buildings of various configurations. Majority of the precast buildings affected were shear wall structures with precast concrete slabs or precast concrete shear walls but not both. The precast members in these buildings were interconnected through strong connections where the continuity of structure was ensured by welding and cast-in-place jointing. Both precast frame and shear wall structures performed very well and exhibited little or no structural damage (Fintel 1995).

The September 19, 1985 earthquake at west coast of Mexico and the aftershock that followed it a day later measured 8.1 and 7.5 on Richter scale, respectively. Of the 265 concrete structures that collapsed or demolished, five contained precast concrete components. Many buildings and parking garages with precast concrete elements survived the earthquakes with little or no damage. Large industrial precast concrete buildings, located only 30 km from the epicenter, performed very well (Fintel 1986).

The 1988 earthquake of Armenia measured 6.9 on Richter scale. There were 78 precast concrete large panel buildings in the area and none of them collapsed or demolished. On the other hand, more than one-third of the precast concrete frame buildings and masonry (composite frame or loadbearing wall)
buildings collapsed or had to be demolished due to extensive damage (Fintel 1995).

Most buildings with precast concrete elements performed well in the 1994 Northridge earthquake, which measured 6.8 on Richter scale. However, parking garages, regardless of the structural system used, did not perform well. While the vertical components of the load resisting system in these buildings performed adequately, the diaphragms in these buildings, especially those with large areas, performed very poorly (Iverson and Hawkins 1994).

Precast concrete wall panel buildings also performed well in the 1995 earthquake of Kobe, Japan, which measured 6.9 on Richter scale. These structures were generally two to five stories tall and typically utilized mechanical reinforcing bar couplers and cast-in-place concrete at precast panel connections. In these buildings, the precast elements showed no damage while the cast-in-place concrete at splice locations showed minor spalling or cracking at a few locations (Ghosh 1995).

The overall performance of precast structures under these earthquakes suggests the viability of precast concrete construction in regions of moderate to high seismicity. However, it should be noted that some of this good performance can be attributed to such factors as the dominant period of the earthquake being very different than that of the precast structures in the area, which according to Fintel (1995) may have been the case in Romania earthquake. It is also notable that in most of the above cases, the precast concrete structures were
constructed using emulative design approach, which, as described in the following section, is still the norm today.

2.3 Precast Concrete Shear Wall Systems

Precast concrete large panel construction is used to refer to structural systems consisting of precast wall panels and precast floor slabs as the primary vertical and horizontal load carrying systems, respectively. The wall panels are typically one story tall and interconnected to each other and the slabs through the horizontal connections to form a box type structure.

Large panel buildings gained popularity on Eastern Europe following the Second World War since they were able to meet the demand for a fast and economic means of construction. Large panels are mass-produced at plants and assembled on-site using cast-in-place connections regardless of weather conditions (Olivia et al. 1988). The panels offer better acoustic and thermal isolation, durability, and fire resistance than other types of building enclosures, and consequently are ideal for use in residential structures.

2.3.1 Building Configurations

The most common arrangement of large panels in a building is the cross-wall system in which bearing walls are placed in the transverse (short) direction and hollow core slabs span in the longitudinal (long) direction. Figure 2.1 shows the plan of a typical cross-wall building whereas Figure 2.2 and Figure 2.3 show the elevation of the same building (Precast/Prestressed Concrete Institute 1999). Other large panel wall configurations include the long-wall and the two-way
Figure 2.1 Plan of a cross-wall system (Precast/Prestressed Concrete Institute 1999)

Figure 2.2 Elevation (longitudinal) (Precast/Prestressed Concrete Institute 1999)

Figure 2.3 Elevation (transverse) (Precast/Prestressed Concrete Institute 1999)
systems (Kianoush and Scanlon 1986). The long-wall (or spline-wall) system contains walls spanning the longitudinal direction and the slabs spanning the transverse direction and the two-way (or mixed) system contains walls placed in both directions.

2.3.2 Shear Wall Configurations

Shear walls spanning in one direction can be configured in three ways:

1. Simple Shear Wall: This is the type of the wall which has horizontal connections only. Figure 2.4 shows a typical “uncoupled” shear wall. As will be discussed later, if platform style “weak connections” are used, the horizontal connections may exhibit rocking and shear-slip type behavior which may lead to the instability of the structure.

2. Coupled Shear Walls: Two shear walls are connected to each other through coupling beams. They might be used for architectural reasons such as door or window openings, corridors, etc. In this type of walls, shown in Figure 2.5, the coupling beams are typically cast-in-place and serve as energy dissipaters.

3. Shear Wall with Vertical Connections: This type of shear wall, shown in Figure 2.6, contains two or more panels connected to each other at vertical connections. Similar to coupling beams, the purpose of the vertical connections is to dissipate energy.

In addition to one-way shear walls, two-way wall systems can be utilized around the elevator cores, stairs, and other building services.
Figure 2.4 Simple shear wall

Figure 2.5 Coupled shear wall
Figure 2.6 Shear wall with vertical connections
2.3.3 Shear Wall Connection Details

Connections in precast concrete shear walls can be classified as horizontal or vertical based on their orientation. Horizontal connections are responsible for transmitting the axial and shear forces coming from the upper panel and the slab to the lower panel. Vertical connections are used to connect wall panels to each other and primarily transmit shear.

Figure 2.7 and Figure 2.8 show the details of “platform” type horizontal connections commonly used in practice in North America. The floor slabs are placed on plastic bearing strips, and the space between the slab panels on each side is filled with grout. Finally, dry-pack is used between the wall panel above and the slab panels (Kianoush and Scanlon 1986). These connections may be unreinforced or reinforced. If reinforcement is used, the continuity of vertical reinforcement is achieved by welding, bolting, or using mechanical connectors such as the splice sleeve at splice locations (Soudki et al. 1995b). Alternatively, bonded (where post-tensioning ducts are filled with grout after post-tensioning) or unbonded (where ungrouted plastic ducts are used) post-tensioning may be used to achieve vertical continuity instead of or along with reinforcing bars (Soudki et al. 1995a).

While most horizontal connections are constructed using “wet” joints as described above, vertical connections are typically constructed using “dry” connections. This connection detail involves welding or bolting steel plates that are anchored to the wall panels on each side.
Figure 2.7 Platform connection

Figure 2.8 Platform connection (at exterior wall)
2.3.4 Behavior of Shear Walls

The behavior of a jointed precast concrete shear wall under a severe seismic load is very different than a cast-in-place reinforced concrete shear wall of similar configuration. In precast walls, the connections are the planes of weakness in terms of strength and stiffness and may exhibit nonlinear behavior at even very low levels of base accelerations (Caccese and Harris 1987; Clough et al. 1989).

The nonlinear behavior is localized at horizontal connections in the form of rocking and shear slip as shown in Figure 2.9. The rocking is a term used to describe the opening and closing of the horizontal connections under seismic load and this mechanism causes high compressive stresses at the ends of the wall panels which may lead to crushing of the concrete and result in instability of the structure. While rocking does not result in accumulated plastic displacements, it also dissipates very little energy (Clough et al. 1989). The shear-slip is a term used to describe the transfer of shear forces across the connection through friction due to axial stresses. In unreinforced wet joints, these axial stresses are due to gravity loads. This type of mechanism dissipates considerable amount of energy and may result if instability if the slip starts accumulating primarily in one direction (Becker and Llorente 1979). The shear resistance of the horizontal connection can be increased by employing reinforcing across the horizontal connection or using post-tensioning. In reinforced wet connections, shear resistance is increased by dowel and clamping action of the reinforcement (Kianoush and Scanlon 1986). Post tensioning the
Figure 2.9 Rocking and slip behavior of a simple wall
connection increases the axial stresses, and thus increases the slip resistance and overall structural integrity. Particularly, the use of unbonded post-tensioning as means to ensure vertical continuity has the effect of reducing the final accumulated deformations of the structure since these structures exhibit almost nonlinear-elastic behavior (Kurama et al. 1999).

Figure 2.10 shows the response of a shear wall system with coupling beams to lateral loads. The nonlinear behavior primarily takes place at the coupling beams away from the horizontal connections. This behavior is much desirable since the coupling beams are not crucial to the structure’s stability. The strength and stiffness of the coupling beams also have an effect on the overall response of the wall system. Low strength coupling beams may yield early and impose high ductility demands on themselves and increase gap opening across horizontal connections while stiffer coupling beams require lower amount of reinforcing and ductility (Kianoush and Scanlon 1986).

Figure 2.11 shows the deformation characteristics of a shear wall with vertical connections. In this case, the vertical connections serve a similar purpose in terms of energy dissipation as the beams in the coupled shear wall described above. The vertical friction type connections are ineffective when coupled with unreinforced horizontal connections, since this type of walls tend to dissipate energy through shear-slip mechanism which reduces the seismic base shear but may result in large accumulated slip as described earlier. However, they are very effective in reducing the permanent deformations when coupled with reinforced or post-tensioned horizontal connections (Pekau and Hum 1991).
Figure 2.10 Seismic response of a coupled shear wall

Figure 2.11 Seismic response of a wall system with vertical connections
The compressive stresses due to rocking is also much smaller compared to the simple wall composed of a single panel at each level because each panel between the vertical connections rock about its edge.

Figure 2.12 shows the elevation of the unbonded post-tensioned split wall which was tested in the final stage of the PRESSS program (Priestley et al. 1999). In this system, the wall panels are dimensioned to ensure that the primary mode of inelastic behavior is rocking, not sliding as shown in Figure 2.13. The unbonded post-tensioning ensures vertical continuity and provides an elastic restoring force to limit final story drifts. The vertical connections dissipate energy and reduce the peak drifts during an earthquake. These connections may be omitted to reduce the construction costs, however, this would result in a less stiff shear wall (Stanton and Nakaki 2002).

2.3.5 Code Provisions

In countries such as New Zealand, the design of precast concrete structures using “emulative design” approach has long been the norm (Restrepo et al. 1995). The term “emulative design” is used to refer to the design approach where the connections are designed to be stronger and stiffer than the precast members they are connecting. As a result, the connections remain linear-elastic, the wall panels yield, and the structure behaves in a manner very similar to a cast-in-place reinforced concrete structure during an earthquake.
Figure 2.12 Unbonded post-tensioned wall with vertical connections (Stanton and Nakaki 2002)

Figure 2.13 Seismic response of an unbonded post-tensioned wall with vertical connections (Stanton and Nakaki 2002)
The design of precast structures using emulative design approach is relatively new in the United States. Figure 2.14 and Figure 2.15 show two of the horizontal connection details recommended by ACI (ACI-ASCE Committee 550 and American Concrete Institute 2002). The connections utilize a combination of grout and splicing using welding or mechanical connectors located at least two times the floor thickness from the face of the wall to ensure vertical continuity, and the code requires them to be designed to remain elastic under seismic loads, emulating the behavior of cast-in-place reinforced concrete. Figure 2.16 shows the detail of such a connection for exterior wall panels. It is possible to use the same design practices developed for cast-in-place reinforced concrete structures in design of precast concrete structures if these types of details are utilized.

“Jointed design” approach, as the name suggests, recognizes the jointed nature of precast concrete construction, and takes advantage of the positive behavioral characteristics of these types of buildings. For example, in the unbonded post-tensioned split wall described in the previous section, the connections are the locations where nonlinear behavior takes place and the panels remain linear-elastic. Although, this type of behavior is completely unacceptable in structures designed using the “emulative design” approach, there are some advantages in this type of a design. These include the self-restoring feature of post-tensioned walls and ease of repair since little or no damage occurs on the panels. Recognizing this, the 2003 edition of National Earthquake Hazards Reduction Program (NEHRP) contains acceptance criteria for “jointed design” of precast structural walls based on validation testing.
Figure 2.14 Connection with welded reinforcement

Figure 2.15 Connection with mechanical splice
Figure 2.16 Connection with mechanical splice (at exterior wall).
(Building Seismic Safety Council 2004). Although the NEHRP recommended provisions is not an actual building code, it serves as a precursor to ASCE 7-05 and the 2006 edition of the International Building Code (IBC) (Ghosh 2004).

These recommended provisions define a minimum acceptance criterion for coupled or uncoupled precast concrete shear walls designed for regions of high seismicity based on experimental and analytical evidence. The experimental validation procedure requires the testing of at least two modules containing two or more panels by an independent testing agency. The modules are required to be large enough to represent the behavior of the materials and the load transfer system of the original structure. These modules are subjected to displacement-controlled fully reversed cycles which increase in magnitude every fourth cycle as shown in Figure 2.17. The drift ratio, which is defined as the angle the roof rotates relative to the base, is increased till it reaches 1.5 times the drift ratio or the value obtained from Equation (2.1), whichever is larger.

\[
0.80 \leq 0.67 \left[ \frac{h_w}{l_w} \right] + 0.5 \leq 2.5
\]  

(2.1)

where,

\[ h_w = \text{Height of entire wall prototype structure, in.} \]

\[ l_w = \text{Length of entire wall in direction of shear, in.} \]

For a wall system to be acceptable, the peak strength of the module should not be less than 90% of the calculated lateral resistance, the module should not experience failure in the reinforcement, the coupling beams, or other sudden degradations in strength. Figure 2.18 and Figure 2.19 show examples of acceptable and unacceptable hysteretic behavior, respectively.
Figure 2.17 Test sequence for precast wall modules
2.4 Precast Concrete Frame Systems

Precast concrete frames have been successfully used in construction of residential, office, industrial and parking garage type buildings (Precast/Prestressed Concrete Institute 1999). Coupled with hollow core slabs, buildings with precast concrete frames can be constructed very quickly and economically. There are three very different design methodologies for construction of buildings with precast frames: hybrid, emulative, and jointed design approaches.

In current practice, most precast concrete frame buildings are designed to utilize cast-in-place reinforced concrete members (shear walls or moment frames) to provide lateral load resistance. While this “hybrid design” approach where cast-in-place and precast members are mixed is an acceptable solution in terms of satisfying the building codes, the cost of such buildings typically exceed those which use just one type of system (cast-in-place or precast) alone for resisting both lateral and gravity loads (Nakaki et al. 1994).

The “emulative design” approach described in Section 2.3.5 for precast concrete shear wall buildings is also applicable to precast concrete frames. If the precast beam to column connection is designed to be stronger and stiffer than the beam itself, the plastic hinge forms away from the connection under a severe earthquake loading. These types of connection details will be discussed further in Subsection 2.4.3.
Figure 2.18 Acceptable hysteretic behavior

Figure 2.19 Unacceptable hysteretic behavior
Of particular interest in this study are precast concrete frames constructed using “jointed design” approach where the connections are the locations that exhibit nonlinear behavior during a seismic event. These types of connections are referred to as “ductile connections” and were developed and tested as a part of the PRESSS research program (Priestley 1991; Priestley et al. 1999). The next section describes the details and behavior of precast concrete frames with these types of ductile connections.

2.4.1 Details of Ductile Connections

The following ductile connections for precast concrete frames were incorporated into the PRESSS five-story test building (Priestley et al. 1999):

1. Unbonded post-tensioned connection with damping
2. Unbonded pre-tensioned connection
3. Tension-compression yield (TCY) connection
4. TCY-gap connection

Figure 2.20 shows the detail of an unbonded post-tensioned frame connection with damping. Frame bays utilizing this type of connection is post-tensioned for the entire width of the frame at beam mid-height. Additional reinforcement is provided at connections at top and bottom by inserting them into ducts in beams. These ducts are then grouted except for a small distance at the face of the interface and throughout the column. The post-tensioning is not grouted (Stanton and Nakaki 2002).

The unbonded pre-tensioned connection shown in Figure 2.21 utilizes pre-tensioning tendons whose centroid is also located at beam mid-height. Unlike
the post-tensioned case, this connection does not contain additional beam reinforcement passing through the column. The continuity of the reinforcement between the columns above and below the connection is achieved by using mechanical connectors.

Figure 2.22 shows the detail of a TCY connection. This connection closely resembles the unbonded post-tensioned connection. However, it only contains the top and bottom reinforcement, not the post-tensioning.

The TCY-Gap connection shown in Figure 2.23 contains a grout pad at the bottom of the beam. Post-tensioning tendons pass through the grout pad and compress the interface. Reinforcing bars are passed through the ducts present at the top of the beam and then grouted. There is a small gap at the top of the connection because of the pad at the bottom.

In all these connections the columns are post-tensioned using threaded bars. The columns are continuous and the beams are single-span except for the case of unbonded pre-tensioned connection, where the beams are continuous throughout the entire frame. Each of these connections responds differently to lateral loads, which will be described in the next section.

2.4.2 Behavior of Precast Concrete Frames with Ductile Connections

The connection types described in the previous section have been incorporated into a five story tall, 60% scale building and tested under simulated seismic loads that are 50% higher than the design forces required by Uniform Building Code.
Figure 2.20 Unbonded post-tensioned connection (Stanton and Nakaki 2002)

Figure 2.21 Unbonded pre-tensioned connection (Stanton and Nakaki 2002)
Figure 2.22 TCY connection (Stanton and Nakaki 2002)

Figure 2.23 TCY-Gap connection (Stanton and Nakaki 2002)
for design of buildings in seismic zone four (Priestley et al. 1999). The building, whose plan is shown in Figure 2.24 contained two frames designed to resist lateral loads. The first frame utilized unbonded post-tensioned connections with damping in the lower three floors and unbonded pre-stressed connections in the upper two floors. The second frame utilized TCY-Gap connections in the lower three floors and TCY connections in the top two floors.

The published report shows that, the overall damage experienced by the structure was much less than an equivalently configured cast-in-place reinforced concrete building. The prestressed frame performed exceptionally with very minor spalling of the concrete cover close to the interfaces and crushing of grout pads, and sustained very small final drift. The non-prestressed frame experienced more damage and bigger residual displacements; however it was also able to dissipate more energy.

Figure 2.25 and Figure 2.26 show the deformation characteristics of the unbonded post-tensioned connection with damping and unbonded pre-tensioned connection without damping, respectively. On both connections, the post or pre-tensioning strands provide the elastic restoring force which is the reason for the small residual drift the prestressed frame experienced. The top and bottom mild reinforcing bars crossing the interface in the post-tensioned connection provide some damping to the structure as they yield in tension and compression in each loading cycle.

Figure 2.27 and Figure 2.28 show the deformation characteristics of the TCY and TCY-gap connections, respectively. Unlike the other two, these
Figure 2.24 Floor plan of the PRESSS test building (Nakaki et al. 1999)
Figure 2.25 Deformed unbonded post-tensioned connection (Stanton and Nakaki 2002)

Figure 2.26 Deformed unbonded pre-tensioned connection (Stanton and Nakaki 2002)
Figure 2.27 Deformed TCY connection (Stanton and Nakaki 2002)

Figure 2.28 Deformed TCY-Gap connection (Stanton and Nakaki 2002)
connections do not re-center after the loading is removed. The TCY connection dissipates energy through yielding of the partially debonded continuity reinforcement in tension and compression in each load reversal. In the TCY-Gap connection, the energy is dissipated by yielding of the continuity reinforcement at the top of the beam in compression as the gap closes, and yielding in tension as the gap opens. The length of the post-tensioning tendons at the bottom of the beam remains constant, and consequently experiences no change in stress. Furthermore, in a frame with TCY-Gap connections, the span length of each bay does not change due to rigid rotations of frame members. This prevents the potential problem of loss of support for precast floor elements located at the ends of the frame (Stanton and Nakaki 2002).

### 2.4.3 Emulative Design of Precast Concrete Frames

The ductile connections for precast systems described in the previous section are designed to be the components which exhibit nonlinear behavior under seismic loads and limit the deformations that the precast columns and beams will experience to the linear-elastic range. A precast concrete building with ductile connections would respond to a base motion differently than a monolithic structure. As a result, both structures would experience different base shears. While the design methodologies specifically suited for jointed construction is slowly making their way into building codes, as is the case in 2003 edition of NEHRP recommended provisions described in Section 2.3.5 for precast concrete shear walls, the only way precast concrete frames can
participate in lateral load resisting system is if they are designed utilizing strong connections.

The connections in a precast frame constructed using “emulative design” approach remain linear-elastic forcing the other frame members to experience nonlinear behavior. Consequently, the seismic response of these types of structures is the type of response that is expected from a monolithic structure. Any code provision that is valid for a cast-in-place reinforced concrete structure is also applicable to these types of precast frames.

One of the requirements of emulative design is the locations where connections can be placed. Instead of using the beam ends, as was the case in ductile frame connections, the strong connections are typically placed at locations where the point of inflection is expected to occur. This necessitates the use of horizontal or vertical cruciform, T-shaped, or H-shaped frame elements. Figure 2.29 shows various configurations utilizing these precast elements and strong connections. Figure 2.30 shows typical beam-to-beam connection details where the connection is located at the point of inflection. Figure 2.31 shows the details of beam to column connections if typical single span beam and column members have to be used.
Figure 2.29 Typical framing arrangement for emulative design (ACI-ASCE Committee 550 and American Concrete Institute 2002)
Figure 2.30 Strong beam-to-beam connection details (ACI-ASCE Committee 550 and American Concrete Institute 2002)

Figure 2.31 Strong column-to-column connections (ACI-ASCE Committee 550 and American Concrete Institute 2002)
2.5 Concluding Remarks

Buildings containing precast members have generally performed adequately in past earthquakes. The experiments carried out during the PRESSS research program suggest that the jointed design approach using ductile connectors is a viable alternative to hybrid construction or precast frames designed using emulative approach. However, the building codes have been slow in adopting these new concepts and technologies. While these tests provide some insight into how these connections will behave under simulated seismic loads, their behavior in a building under a general seismic loading where three-dimensional effects are possible is still unknown.

The finite element method of analysis has a great potential as an analytical tool for filling this gap in information on behavior of jointed precast structures. The next chapter describes the base components of the finite element framework developed for the purpose of analyzing the three-dimensional behavior of precast concrete structures with ductile connections.
Chapter 3

Snaps3D: A Platform for Analysis of Precast Concrete Structures

3.1 General

In the previous chapter it was shown that analytical modeling and analysis of jointed precast building structures is needed to fully understand behavior under seismic loading. Existing software such as DRAIN2D (Kanaan and Powell 1973) and ABAQUS (ABAQUS Inc. 1998) have been developed using procedure-based programming approach and coded in FORTRAN 77, which is outdated by today’s standards. Although these programs allow new capabilities to be added through the use of user subroutines, this process requires the users to have some understanding of the overall structure of the program and how data is passed between different subroutines (Archer 1996). Furthermore, features such as inheritance, which promotes code reuse, is not available in FORTRAN. With each new addition the software becomes more complex. Emerging software development technologies such as the .NET Framework and the C# programming language allow distributed applications to be written in a way which is much less restrictive than the procedure based approach supported by FORTRAN.
SNAPS3D is a set of matrix and finite element class libraries developed to perform nonlinear dynamic analysis of precast concrete building structures. In Section 3.2 the Microsoft .NET Framework and the C# programming language, used for development of SNAPS3D, and their suitability for use in scientific computing are discussed.

Section 3.3 and Section 3.4 describe .NET assemblies which contain the matrix classes and the core finite element classes, respectively.

3.2 Description of Base Platform

Since its introduction in 2002, the Microsoft .NET Framework has revolutionized the way windows and web applications are developed (Microsoft 2005). While .NET provides the developers an object-oriented platform to deliver reusable code and supports cross-language programming, its applications in developing high-performance numerical code such as finite element applications remains largely unexplored. This section aims to provide an introduction to the .NET Framework, followed by an introduction to C# which is selected as the primary programming language for SNAPS3D code development, and demonstrate how the technologies incorporated into this platform can be used to develop a set of reusable class libraries for finite element analysis.

3.2.1 The .NET Framework

The .NET Framework consists of an execution engine called the Common Language Runtime (CLR) and a set of Base Class Libraries (BCL) as seen in Table 3.1 and allows seamless integration between different programming
languages. The CLR handles services such as code loading and execution, automatic memory management (providing memory for new objects and disposing of objects when they are no longer referenced by the code), type safety (i.e. a string type can not be cast as an integer type), and cross-language integration. Any computer code written specifically for CLR is referred to as “managed code”.

<table>
<thead>
<tr>
<th>Table 3.1 The components of the .NET Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>XML-Web Services</td>
</tr>
<tr>
<td>ASP.NET</td>
</tr>
<tr>
<td>Data and XML Classes</td>
</tr>
<tr>
<td>Base Framework Classes</td>
</tr>
<tr>
<td>Common Language Runtime</td>
</tr>
</tbody>
</table>

The BCL contain hundreds of .NET types that provide services from such basic functionality as input/output and text manipulation to more complex ones such as databases and user interface design. These provide developers a common set of libraries to base their work on instead of developing those classes themselves or obtaining them through a third party. All these classes are sorted into different namespaces to avoid any naming conflicts. At the top of the BCL is the `System.Object` class, where the System prefix signifies that the class is located under the System namespace. Every type in the .NET Framework, including the `System.Array` class, ultimately derives from this class.

Performance, portability, reusability, and support for distributed computing are the most important issues related to finite element application development. The evaluation of the .NET Framework on these issues requires the
understanding of how this platform works. A compiler that targets the CLR does not produce native code. Instead, the compiler emits a CPU-independent intermediate language (IL) code. During run-time the IL is converted to native code by the Just-In-Time (JIT) compiler provided by the CLR. Figure 3.1 shows the steps involved in the compilation procedure.

Figure 3.2 shows the performance comparison between C# (which is strictly a .NET language) and C in a variety of numerically intensive code, reported by Gilani (2004). The author attributes the significant performance difference between C# and C in benchmarks such as dense matrix factorization and sparse matrix multiplication to the use of non-optimized implementation of the benchmark suite for C#. Even if this is the case, managed code (the code that is written for the .NET Framework) will always have an overhead because of the conversion that takes place from IL to native code during runtime. However, with each new release of the .NET Framework, this overhead is expected to get smaller. It is also important to realize that unmanaged code (non-.NET code) needs to be recompiled to take advantage of the features provided by newer CPU’s, whereas, this is not the case with managed code. Managed code will always take advantage of all the features of the specific CPU it is running on since the CLR provides an optimized JIT for the hardware platform the application is running on.
Figure 3.1 Steps in compiling source code for the .NET Framework (Richter 2000)
Figure 3.2 SciMark results between C# and C (Gilani 2004)
Portability for an unmanaged code means that the source code can be taken from one platform to another and compiled there without any code changes provided that there is a compiler which will do the job for that platform. With managed code, once the code is compiled into a .NET assembly (.NET executable or dynamic-link library) it can be run in any platform regardless of CPU or operating system, provided that the operating system hosts the .NET Framework CLR.

Reusability has always been one of the most crucial factors related to design and development of general purpose finite element codes. Most finite element codes available today are designed using procedure-based programming approach and typically coded in FORTRAN 77. For example, a typical procedure-based explicit finite element code has specialized subroutines to read mesh data and properties, generate global mass matrix, carry out time-integration, and write output as shown in Figure 3.3. The information regarding nodal positions, element connectivity, material properties, and other information related to the model is generally stored in global arrays, since these are information that are needed to be accessed frequently by the mentioned subroutines. The task of implementing a new element type or a new material behavior to such a program is quite difficult and requires most, if not all, of the subroutines in Figure 3.3 to be modified. In addition to this, the global arrays may have to be modified to accommodate the information related to the new type that is being implemented. Modifications of this magnitude not only require the implementer to have an understanding of the entire program, but have a risk of
Figure 3.3 A typical procedure-based explicit FEM flowchart (Owen and Hinton 1980)
breaking the existing program. After the modifications are complete the program not only needs to be validated for the new feature, but also be checked to see if older input files still work.

The severe restrictions posed on reusability by inherent limitations of FORTRAN lead many researchers to investigate C++ programming language, which supports object-oriented programming (OOP), as a possible candidate for finite element software development. In an object-oriented code, classes are the building blocks. The instances of classes, called objects, contain both data fields and methods (which are equivalent to FORTRAN subroutines and functions) to modify that data. In a sense, data processes itself. This is very different than a procedure-based code, where procedures are the building blocks of software and procedures process data. Finite element concepts such as nodes, elements, materials, and loads readily present themselves as possible candidates for abstraction into class definitions. The primary advantage of this is the elimination of global arrays such as the connectivity matrix to store information about the model. For example, if the element class is designed to have the element node array as a data field, when an element instance wants to calculate its mass matrix, it already knows what its nodes are and where they are positioned. Thus, there is no need to access to a global connectivity matrix.

OOP languages also provide two powerful features to promote code reuse: inheritance and polymorphism. These features are like the two faces of a coin: Inheritance lets developers build new classes (child) from existing classes (base). For example, the Solid child class derives from the Element base class.
Polymorphism lets the developers treat related objects the same way. For example, since a Solid is a child class of the Element class, any Solid object can be treated as an Element object. While the first ensures that the new type has all the features of the base types since it derives from it, the latter ensures that the rest of the code will not need to be modified since for an object responsible for controlling the solution procedure all that matters is that it can still refer to the new type as if it was a member of the base type.

Unlike C++ which is an OOP language, .NET Framework is an object oriented platform. It features over twenty programming languages, including C#, Managed C++ (a specific C++ compiler that targets the CLR, i.e. creates code that is “managed” by the CLR), Visual Basic .NET, and even Fortran for .NET (a proprietary FORTRAN compiler which adheres to the Fortran 2003 Draft Standard and supports cross-language inheritance). This means that classes developed for the .NET Framework can be extended using any .NET aware programming language, not just the one it was written in. Furthermore, access to the source code of the existing program is not required: All that is required is the .NET assembly of the original program. Once this assembly is referenced, any classes, interfaces, etc. are available for reuse (provided that the original developer allowed code reuse).

Distributed computing for CPU-intensive analysis using the .NET Framework does not require any expensive hardware and/or software. A feature called .NET Remoting enables developers to build applications where the application components may be distributed among different computers on the
network or even different parts of the world. An example of using .NET Remoting for a CPU-intensive application has been published (D'Anna 2005).

Of all the programming languages that can be used for developing .NET applications, the C# programming language is considered to be the best in terms of exposing the inner workings of the CLR and the BCL. The next section is devoted to providing a simple introduction to C# and its syntax.

### 3.2.2 C# Programming Language

While the CLR supports many programming languages, the C# programming language, which was specifically designed by Microsoft to develop code managed by the CLR, stands out with the features it presents to developers (Petzold 2003). It belongs to a family of programming languages which have their roots in the C programming language. However, unlike C++, C# is not a superset of C.

C/C++ features such as pointers are not permitted under normal circumstances and C# does not require any header files (Trupin 2000). C# is touted to be the first component-oriented programming language in the C-family of languages because component concepts such as properties and events are first-class code members as shown in Table 3.2. C# also features integrated documentation using XML code comments. From these XML code comments, it is possible to create a complete technical documentation of the code including namespaces contained in each assembly, class hierarchy, class members and their definitions.
Table 3.2 C# class members

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field</td>
<td>A data variable that represents the state of the object.</td>
</tr>
<tr>
<td>Method</td>
<td>A function that performs an operation on the object usually changing the state of the object.</td>
</tr>
<tr>
<td>Property</td>
<td>Specialized methods generally used to modify the data variables. Can provide full access or read or write only access to a field.</td>
</tr>
<tr>
<td>Event</td>
<td>These provide notification mechanism between an object and other interested objects.</td>
</tr>
</tbody>
</table>

C# contains a wide range of modifiers to improve how objects are accessed. Table 3.3 shows some of these type and/or type member modifiers and what type of access restrictions they provide. Like all OOP languages, C# encourages encapsulation of data. Because of this fields are typically marked as private, internal, or protected to avoid giving other objects the ability to change an object's state directly. Properties, on the other hand, are typically marked public since they control how other objects can change the state of an object.

Table 3.3 Partial list of C# access modifiers

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public</td>
<td>There is no restriction on access to the type or type member.</td>
</tr>
<tr>
<td>private</td>
<td>Type members with this modifier can only be accessed within the body of the class or structure.</td>
</tr>
<tr>
<td>internal</td>
<td>Types and type members with this modifier can only be accessed from files in the same assembly.</td>
</tr>
<tr>
<td>protected</td>
<td>The type members with this modifier are accessible within the class in which it is declared and any class derived from it.</td>
</tr>
<tr>
<td>abstract</td>
<td>Indicate that a class is intended only to be a base class of other classes.</td>
</tr>
<tr>
<td>virtual</td>
<td>Declare a method or an accessor whose implementation can be changed by an overriding member in a derived class.</td>
</tr>
<tr>
<td>override</td>
<td>Provide a new implementation of a virtual member inherited from a base class.</td>
</tr>
<tr>
<td>static</td>
<td>Declare a member that belongs to the type itself rather than to a specific object.</td>
</tr>
</tbody>
</table>
Abstract and virtual are two modifiers that are the key to developing reusable code and without these modifiers polymorphism would not be possible. Any method marked as “virtual” in a class can be overridden with a new method in a child class marked as “override”. The major difference between abstract (when used with methods) and virtual modifiers is: Abstract methods do not provide any implementation and can only be used in classes that are marked as abstract. Any class that inherits from an abstract class needs to implement the method as well. Virtual methods on the other hand can be used in concrete (non-abstract) classes and they provide an implementation. If a child class does not override that particular method, then it uses that method in its original form.

Snaps3D Algebra assembly, which will be described in the next section, is a matrix and vector class library for the .NET Framework. It has been coded with the C# programming language.

3.2.3 Structure of .NET Applications

Programs developed using .NET Framework contain one or more binaries called .NET assemblies. To avoid naming conflicts and to aid in organizing the code, each assembly can contain multiple namespaces, which serve as folders for storing classes that are designed to perform similar tasks. The current version of Snaps3D contains three .NET assemblies named Snaps3D.Algebra, Snaps3D.Core, and Snaps3D.Forms. As the names of these files imply, they each contain types related to basic matrix and vector classes, the core types related to finite element analysis, and the graphical user interface that enables users to create finite element objects, respectively. While the Snaps3D.Algebra
assembly contains a single namespace which carries the same name, the Snaps3D.Core assembly contains multiple namespaces including one named Snaps3D.Core and another named Snaps3D.Core.Materials, under which all material classes are located.

3.3 Snaps3D.Algebra Assembly

Unlike Fortran 90, C# programming language does not have built-in support for matrix operations. However, it has all the facilities, such as indexers and operator overloading, to allow development of very comprehensive matrix and vector classes. The Snaps3D.Algebra assembly contains a single namespace named Snaps3D.Algebra. Figure 3.4 shows the matrix and vector classes under the Snaps3D.Algebra namespace.

3.3.1 Concept Description

The main purpose of the Snaps3D.Algebra assembly is to provide the necessary matrix and vector classes for finite element analysis. Classes such as DoubleVector and DoubleMatrix are very similar to FORTRAN arrays with double precision floating point numbers as elements. The classes ObjectVector and ObjectMatrix, on the other hand, do not have any FORTRAN equivalents. These are vector of vectors or matrix of matrices, respectively. For example, the stiffness matrix of an 8-noded solid element would be an 8x8 ObjectMatrix. Each of the elements of this matrix would be 3x3 DoubleMatrix objects.
Figure 3.4 Snaps3D.Algebra namespace
3.3.2 DoubleVector Class

This class allows the creation and manipulation of one-dimensional arrays. It contains operators for performing addition, multiplication, etc., and methods for calculating cross and dot products of two vectors.

3.3.3 DoubleMatrix Class

A DoubleMatrix is a two-dimensional matrix whose elements are double-precision floating point numbers. It contains operators for performing addition, multiplication, etc., and methods for calculating the transpose of the matrix and its determinant and inverse if applicable.

Figure 3.5 demonstrates how instances of the DoubleMatrix class can be declared, initialized, and assigned to. It also shows the matrix multiplication operator at work in the last statement.

```java
// 1. Define a matrix:
DoubleMatrix m1;

// 2. Initialize it as a 3x3 matrix
m1 = new DoubleMatrix(3, 3);

// 3. Assign values to some of the elements:
  m1[0, 0] = 1.5;
  m1[1, 1] = 2.0;
  m1[2, 2] = 4.5;

// 4. Define another matrix:
DoubleMatrix m2;

// 5. Use it to store the multiplication below:
  m2 = m1 * m1;
```

Figure 3.5 DoubleMatrix class example

The first statement in Figure 3.5 tells the compiler that the variable m1 will be used to refer to a DoubleMatrix type object. The second statement initializes a new 3x3 DoubleMatrix instance, whose elements are zero by default, and then
associates it with m1. The third statement assigns values to the diagonal of the matrix referenced by m1. The fourth statement tells the compiler that m2 will be used to refer to a DoubleMatrix type object and the fifth statement evaluates the multiplication of the matrix referenced by m1 by itself, and associates the resulting matrix with m2.

Although C# matrix classes look and behave like FORTRAN matrices, they are (just like any other C# class) reference type objects. Consider the example shown in Figure 3.6. If these statements were written in FORTRAN, the value of m3(0,0) would be 1 after the last statement. This is not the case with C#, and m3(0,0) would return 2. This is because of the second statement: In FORTRAN this statement would have caused all the elements of m3 to be copied to a new matrix called m4, and anything that was done on m4 would have no effect on m3. In C#, this statement causes the matrix which was referenced by m3 to be also referenced by m4.

```csharp
// 1. Define and initialize the matrix:
DoubleMatrix m3 = new DoubleMatrix(2, 2);

// 2. Assign the element at indices 0,0 a value:
m3[0, 0] = 1;

// 2. Define another matrix and set it equal to the first one:
DoubleMatrix m4 = m3;

// 3. Change the value at indices 0,0:
m4[0, 0] = 2;
```

Figure 3.6 Another DoubleMatrix class example

Figure 3.7 shows the implementation of the multiplication operator which takes two DoubleMatrix objects as input and returns a “new” DoubleMatrix object. The DoubleMatrix class also has multiplication operators for multiplication of a
matrix and a vector, multiplication of a scalar and a matrix. The compiler automatically determines which method to use by checking the types of the variables on each side of the operator sign.

```java
public static DoubleMatrix operator *(DoubleMatrix m1, DoubleMatrix m2)
{
    if (m1 != null && m2 != null)
    {
        DoubleMatrix m3 = new DoubleMatrix(m1.Rows, m2.Cols);
        double tmp;

        for (int row = 0; row < m1.Rows; row++)
        {
            for (int col = 0; col < m2.Cols; col++)
            {
                tmp = 0;
                for (int k = 0; k < m1.Cols; k++)
                {
                    tmp += m1[row, k] * m2[k, col];
                }
                m3[row, col] = tmp;
            }
        }
        return m3;
    }
    else
    { return null; }
}
```

Figure 3.7 Multiplication operator for DoubleMatrix objects

### 3.3.4 DiagonalDoubleMatrix Class

This is the last class to use double-precision floating point numbers as its elements, which can only be located on the diagonal. This type of objects forms the sub-matrices of element or model level mass matrices.
3.3.5 **BoolVector Class**

This is similar to the DoubleVector class, except all its members are Boolean.

3.3.6 **ObjectVector Class**

ObjectVector class allows the creation of one-dimensional array of DoubleVector objects. All element and model level vectors are this type.

3.3.7 **ObjectMatrix Class**

This is a two-dimensional matrix whose elements are DoubleMatrix objects instead of double-precision floating point numbers. It contains operators for performing addition, subtraction, multiplication, etc.

Figure 3.8 shows an example of how ObjectMatrix objects are defined and accessed. Although, the DoubleMatrix and ObjectMatrix classes look very similar except their element types, there is a significant difference in how memory is allocated for them by the C# compiler. Both DoubleMatrix and ObjectMatrix instances are reference types as described before. However, the elements of a DoubleMatrix object are double-precision floating point numbers (System.Double or double in C# syntax), which are **value type** objects. Creating a 3x3 DoubleMatrix instance would automatically reserve enough space to hold nine double numbers and initializes them to zero. Creating a 3x3 ObjectMatrix would reserve enough space to store nine references which are all initialized to “null” which means “not an object reference”. For example, following the 3rd step in Figure 3.8, a(0,0) and a(0,1) refer to the DoubleMatrix objects defined in step 1.
However, \(a(1,0)\) and \(a(1,1)\) were not assigned to, and as a consequence do not refer to anything. As far as the compiler is concerned, they are null.

```java
// 1. Define two DoubleMatrix objects (to serve as sub-matrices):
DoubleMatrix a00 = new DoubleMatrix(2, 2);
DoubleMatrix a01 = new DoubleMatrix(2, 2);

// 2. Define a ObjectMatrix object:
ObjectMatrix a = new ObjectMatrix(2, 2);

// 3. Assign the previously defined sub-matrices to it:
a[0, 0] = a00;
a[0, 1] = a01;

// 4. Set some values:
a[0, 0][1, 1] = 10; // same as setting a00[1, 1] = 10;
a01[0, 0] = 20; // same as setting a[0, 1][0, 0] = 20;
```

**Figure 3.8 ObjectMatrix class example**

### 3.3.8 DiagonalObjectMatrix Class

This is the type of ObjectMatrix where all the elements are located on the diagonal and they are of DiagonalDoubleMatrix type. An object of this type is used to store the mass matrix of the model and its inverse. While this class resembles a vector rather than a matrix, it has a method for getting the inverse, and a multiplication operator, which takes a DiagonalObjectMatrix object and multiplies with an ObjectVector object.

### 3.4 Snaps3D Core Assembly

Snaps3D.Core assembly contains all the types for creating and manipulating finite element related objects. Unlike the other two assemblies which were meant to be replaceable with better ones when or if available, this assembly is meant to be extensible. For ease of navigation, this assembly
contains multiple namespaces as shown in Figure 3.9, and the following subsections describe the types that are contained under each namespace.

Snaps3D.Core namespace is located at the top of all the namespaces in this assembly and contains all the types that are crucial to execution and extensibility of Snaps3D. Figure 3.10 shows all the types that are located in this namespace. Of particular interest for extensibility of Snaps3D are the interfaces which this namespace contains. Interfaces are reference types just like classes and are considered to be similar to abstract classes. C# interfaces cannot contain any implementation, whereas even an abstract C# class is permitted to have some non-abstract methods and fields. Furthermore, a C# class can only inherit from a single base class (unlike C++), but a C# interface can inherit from multiple interfaces. The only type members that are permitted in an interface are properties and methods and the access to these type members can only be public.

Figure 3.11 through Figure 3.16 show the remaining namespaces in the Snaps3D.Core assembly and the types they contain. For example, the Snaps3D.Core.Materials assembly contains the classes responsible for storage of material properties. The Material class, which will be described later, is the base class for all the other classes in this namespace. The Elastic class derives from the Material class, and adds properties such as Poisson’s ratio and modulus of elasticity. The Concrete and Rebar classes are derived from the Elastic class and add new properties to define the types of materials they were named after.
Figure 3.9 Namespaces in Snaps3D.Core assembly

```plaintext
Snaps3D.Core {}
Snaps3D.Core.Materials {}
Snaps3D.Core.Secions {}
Snaps3D.Core.IntegrationPoints {}
Snaps3D.Core.Elements {}
Snaps3D.Core.Loads {}
Snaps3D.Core.Solvers {}
```

Figure 3.10 Snaps3D.Core namespace
Figure 3.11 Snaps3D.Core.Materials namespace

Figure 3.12 Snaps3D.Core.Sections namespace

Figure 3.13 Snaps3D.Core.IntegrationPoints namespace
Figure 3.14 Snaps3D.Core.Elements namespace

Figure 3.15 Snaps3D.Core.Loads namespace

Figure 3.16 Snaps3D.Core.Solvers namespace
In the following subsections, the finite element related types (interface or class) which form the Snaps3D.Core assembly are described.

### 3.4.1 Concept Description

The names of the interfaces and classes described in this section are selected based on the function they provide. The most primitive type in Snaps3D is the Member class, which has properties to assign a name and a domain, and serves as the base class for more complex Snaps3D classes. The top-level type is the Domain class which allows the entire model information to be represented by a single object. Any class that derives from the Member class, such as Node, Element, and IntegrationPoint objects, can be added to the “member list” of a Domain object. A Node object is used to define a point in space, an Element object is used to define a volume in space, and an IntegrationPoint object is used to define a material point inside an Element object.

### 3.4.2 Interfaces

Interfaces can be considered as “contracts” between the original developer and anyone who is interested in extending the code or developing “components” for it. The programming approach where interfaces are the keys to reusability is called component-oriented programming or component-based software design (CBSD). The OOP approach, which has been common in FEM software written in C++, encourages code reuse through inheritance from an existing class. This is very advantageous if the implementer of the new type has a “white-box” understanding on the class he is inheriting from and knows which
methods to override and what their dependencies are. The CBSD approach encourages code reuse through interface implementation. This is referred to as the “black-box” approach to software design, since it does not require any knowledge from the developer about the class hierarchy or how methods are implemented in a class: as long as the software component implements all the methods and properties which are required by the interface, it satisfies the contract and can be plugged to the base code. The interfaces IMember, IInteractiveMember, IMaterial, ISection, IElememt, ISolver, and IIntegrationPoint in this namespace are provided for this type of black-box extensibility. For example any class that implements the IElememt interface is considered to be an element as far as Snaps3D is concerned and treated as such by other parts of the program. The IMember interface serves as the base type for all the other interfaces and is described in the next subsection.

### 3.4.2.1 IMember Interface

This interface, shown in Figure 3.17, sets the minimum standard for methods and properties a component should implement to be considered a Snaps3D domain member. This means that, any class that implements this interface must have name and domain properties, and should contain methods to create a copy of itself, check to see if it refers to any other object which implements IMember and remove that reference if requested. Finally, it should allow itself to be deleted from the domain by setting all references to itself to null.
using System;

namespace Snaps3D.Core
{
    public interface IMember
    {
        string Name
        {
            get;
            set;
        }

        Domain Domain
        {
            get;
            set;
        }

        IMember GetCopy();

        bool ContainsReferenceTo(IMember target, bool removeReference);

        void RemoveReferenceTo(IMember target);

        void Delete();
    }
}

Figure 3.17 IMember interface
3.4.2.2 IInteractiveMember Interface

This interface shown in Figure 3.18 derives from the IMember interface which means that a class that implements IInteractiveMember interface also has to implement the properties and methods described in IMember. The classes that implement IInteractiveMember interface should be designed to interact with the user which is accomplished by two methods. The first method GetListViewItem() returns a ListViewItem object which allows it to be displayed in a table style window. The second method GetForm() returns a windows form which lets the user edit its properties. Both ListViewItem and Form are BCL classes for user-interface development and are located under System.Windows.Forms namespace.

Generally speaking, any developer who would like to extend Snaps3D classes would not use either the IMember or the IInteractiveMember interfaces. The next set of interfaces: IMaterial, ISection, IElemen, ISolver, and IIntegrationPoint are designed for this purpose.

3.4.2.3 IMaterial Interface

Figure 3.19 shows the code for IMaterial interface whose primary purpose is to hold the material properties and when asked create an integration point appropriate for that material. Any class that implements IMaterial interface should specify a density and should have a method to create an integration point to represent its behavior for a given type of an element. The IMaterial interface and the IIntegrationPoint interfaces are intimately tied since one provides the properties of the material and the other applies the material law for a specific
using System;
using System.Windows.Forms;

namespace Snaps3D.Core
{
    public interface IInteractiveMember : IMember
    {
        ListViewItem GetListViewItem();
        Form GetForm();
    }
}

Figure 3.18 IInteractiveMember interface

using System;
using Snaps3D.Core.Elements;
using Snaps3D.Core.IntegrationPoints;

namespace Snaps3D.Core
{
    public interface IMaterial : IInteractiveMember
    {
        double Density
        {
            get;
            set;
        }

        IIntegrationPoint CreateIntegrationPointInstance(IElement sender);
    }
}

Figure 3.19 IMaterial interface
type of element. The classes that implement this interface are grouped under Snaps3D.Core.Materials namespace and include the base class Material, and specialized classes Elastic, Rebar, and Concrete.

3.4.2.4 IIntegrationPoint Interface

Figure 3.20 shows the code for the IIntegrationPoint interface. The classes that implement this interface are grouped under Snaps3D.Core.IntegrationPoints namespace, and responsible for defining the coordinates of an integration point in the natural coordinate system, the weight of the integration point and its material. This interface is implemented in IntegrationPoint class, which is the base class to implement the interface, and other classes specialized classes such as ElasticSolid, ElasticShell, ElasticBeam, ConcreteBeam, and KinematicBeam.

The only method an integration point is required to have is the GetStressVector() method. This method calculates the stress vector resulting from a prescribed strain vector and returns it to the requesting object, which is typically an element.

Unlike IMaterial, ISection, and IElement interfaces which derive from IInteractiveMember, this interface derives directly from IMember. This is because objects of this type are created automatically during analysis runtime by the element that owns it and do not need a windows form for user interaction. The IIntegrationPoint interface also implements the ITraceable interface to ensure that the stress and strain values can be saved for later output at specific time intervals.
using System;
using System.Data;
using Snaps3D.Algebra;
using Snaps3D.Core.Elements;
using Snaps3D.Core.Materials;

class IntegrationPoint
{
}

namespace Snaps3D.Core
{
    public interface IIntegrationPoint : IMember, ITraceable
    {
        int Index
        {
            get;
            set;
        }

        DoubleVector Position
        {
            get;
        }

        Double Weight
        {
            get;
        }

        IMaterial Material
        {
            get;
            set;
        }

        IEElement Owner
        {
            get;
            set;
        }

        Double Djb
        {
            get;
            set;
        }

        DoubleVector GetStressVector(DoubleVector strain);
    }
}

Figure 3.20 IIntegrationPoint interface
Each integration point belongs to a single element and storing the coordinates and weights of every integration point might put unnecessary demands on memory. To avoid this, the ISection interface, which will be described next section, stores the actual coordinates and weights.

3.4.2.5 ISection Interface

Instances of ISection interface store the positions, weights, and the materials of the integration points that an element can have as shown in Figure 3.21. Unlike an IIntegrationPoint instance, which is referenced by only one element, ISection instances can be referenced by multiple elements. During the analysis, each element object requests its integration points to be created by calling GetElementIntegrationPoint() method.

Classes that implement this interface are located in the Snaps3D.Core.Sections namespace and include the Section class, which represents the most general case where all integration points can have a different material, the FullyIntegrated class, which creates 8 integration points per element and assign coordinates to them that represent the 2x2x2 integration rule, the Layered class, where any number of layers made up of different materials can be defined and each layer is integrated using the 2x2 integration rule, and the Fibered class, which defines a beam type section with reinforcement in it.

3.4.2.6 IEElement Interface

The IEElement interface defines the properties and methods an element object should have. Classes that implement this interface are located under
using System;
using Snaps3D.Algebra;
using Snaps3D.Core.Materials;
using Snaps3D.Core.Elements;
using Snaps3D.Core.IntegrationPoints;

namespace Snaps3D.Core
{
    public interface ISection : IInteractiveMember
    {
        double ShearFactor
        {
            get;
            set;
        }
        DoubleVector[] PositionArray
        {
            get;
            set;
        }
        double[] WeightArray
        {
            get;
            set;
        }
        IMaterial[] MaterialArray
        {
            get;
            set;
        }
        IIntegrationPoint[] GetElementIntegrationPoints(IElement sender);
    }
}

Figure 3.21 ISection interface
Snaps3D.Core.Elements namespace and include the base class Element, Solid, Shell, and Beam. The basic requirements to satisfy the element criteria are to have a node array, section, and type properties as shown in Figure 3.22. Of these, type refers to the type of element (i.e. solid, shell, or beam), since each type would require a different type of integration point.

The InitializeElement() method provides each element object a method to perform final initializations at the start of the time-history analysis, before the time-integration starts. Currently, this method is used to initialize the integration points of each element object. The GetLumpedMass() method tells the element object to return its lumped mass matrix and the GetInternalForce() method tells the element to calculate the internal force vector due to prescribed nodal displacements. The GetExternalForce() method gets called only if an element belongs to the list of an ElementLoad object.

Currently, the methods are limited to the ones that are necessary for performing explicit time integration such as GetLumpedMass() and GetInternalForce(). However, it is possible to define a new interface derived from this one and add methods that are appropriate for implicit time-integration in the future.

3.4.2.7 ISolver Interface

The ISolver interface, shown in Figure 3.23, represents the properties and methods a solver class should implement. While this is a very short interface, the actual class that implements it is one of the most complex in Snaps3D since it is responsible for controlling creation of strong-typed arrays, such as the
using System;
using Snaps3D.Algebra;
using Snaps3D.Core.Sections;
using Snaps3D.Core.IntegrationPoints;

namespace Snaps3D.Core
{
    public interface IElement : IInteractiveMember
    {
        ElementType Type
        {
            get;
        }

        Node[] NodeArray
        {
            get;
            set;
        }

        ISection Section
        {
            get;
            set;
        }

        void InitializeElement();
        DiagonalObjectMatrix GetLumpedMass();
        ObjectVector GetExternalForce(DoubleVector bodyForce);
        ObjectVector GetInternalForce();
    }
}
using System;
using System.ComponentModel;
using System.Windows.Forms;

namespace Snaps3D.Core
{
    public interface ISolver : IInteractiveMember
    {
        double Time
        {
            get;
        }

        double TimeStepSize
        {
            get;
            set;
        }

        int TimeStepCount
        {
            get;
            set;
        }

        void PerformTimeSteps(BackgroundWorker worker, DoWorkEventArgs e);
    }
}
element and node arrays, calculate global mass matrix, force matrices, and perform time integration. While the Domain class, which will be described later, is the type which stores all the model information, the solver is the type which actually controls the solution procedure.

### 3.4.2.8 ITraceable Interface

The ITraceable interface, shown in Figure 3.24, is meant to be implemented any class that will gather data during the time-history analysis. Types such as integration points and nodes are required to implement this interface. Each instance of this type should have a switch to enable or disable tracing. An instance of the DataTable class which is a member of the BCL and is located in System.Data namespace is the database where all the time-history information is saved for later review.

### 3.4.3 Classes

#### 3.4.3.1 Member Class

The Member class is the base class for most of the classes in Snaps3D.Core namespace and the namespaces located under it. This abstract class also implements the IMember interface. Table 3.4 lists the members of this class.
using System;
using System.Data;

namespace Snaps3D.Core
{
    public interface ITraceable
    {
        bool IsTraced
        {
            get;
            set;
        }

        int TraceInterval
        {
            get;
            set;
        }

        DataTable TraceTable
        {
            get;
        }

        void SaveTrace();
    }
}

Figure 3.24 ITraceable interface
Table 3.4 Member class members

Public Instance Constructors

| Member Constructor | Creates a new Member instance and assigns it a default name. |

Public Instance Properties

| Domain | Gets or sets the domain this Member instance belongs to. |
| Name   | Gets or sets the name of the Member instance. |

Public Instance Methods

| ContainsReferenceTo | Overloaded. Checks to see if this IMember instance contains reference to the target object. |
| Delete              | Removes any references to this object from all the objects contained in the domain, removes this object from the memberlist of the domain and prepares the object for GC. |
| GetCopy             | Creates a new Member instance which belongs to the same domain as the original. |
| RemoveReferenceTo   | Removes the reference to target object from the properties of the current instance (if applicable). |
| ToString           | (Overridden) Returns the name of the Member instance. |

Protected Instance Methods

| GetDetails | Creates a string which represents all the properties of the Member instance. |
3.4.3.2 Domain Class

Domain class is the top-level class in Snaps3D. Table 3.5 shows the list of the Domain class members (the list of the members inherited from base classes are not included). It contains a Damping object, a Solver object, and a collection type object called MemberList, to which any object which implements IMember interface can be added. Although both Damping and Solver classes implement IMember interface, there can only be one of each per Domain instance. All other Snaps3D types, such as nodes, elements, and loads, are added to the “member list”. The task of sorting and filtering the objects to their appropriate arrays is carried out by the solver object. The reason for not having strong-typed collections such as “node list” or “element list” is to leave some leeway for future “members” such as sub-domains or super-elements.

3.4.3.3 InteractiveMember Class

InteractiveMember class is the abstract base class that implements the IInteractiveMember interface. The classes which derive from this class have a windows form, which allow the state of the object to be changes during runtime. Table 3.6 shows the list of the InteractiveMember class members. Since this class inherits from the Member class, it also has access to the methods and properties listed in Table 3.4. These members were not included in Table 3.6 to avoid repetition.
Table 3.5 Domain class members

<table>
<thead>
<tr>
<th>Public Instance Constructors</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Domain Constructor</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Public Instance Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Damping</strong></td>
</tr>
<tr>
<td><strong>MemberList</strong></td>
</tr>
<tr>
<td><strong>Solver</strong></td>
</tr>
<tr>
<td><strong>State</strong></td>
</tr>
</tbody>
</table>

Table 3.6 InteractiveMember class members

<table>
<thead>
<tr>
<th>Public Instance Constructors</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>InteractiveMember Constructor</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Public Instance Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GetForm</strong></td>
</tr>
<tr>
<td><strong>GetListViewItem</strong></td>
</tr>
</tbody>
</table>
3.4.3.4 Node Class

A Node object represents a point in space. It not only stores the coordinates of the point, but its displacement, velocity, and acceleration vectors for the current time step of the time-history analysis, as well. The Node class also implements the ITraceable interface, which allows the displacement, velocity, and acceleration vectors to be saved to a DataTable at user-defined intervals. Currently, the default length of each of the mentioned vectors is three. However, this might be changed easily if elements with rotational dof are added to Snaps3D. The members of the Node class are shown in Table 3.7.

3.4.3.5 Boundary Class

The Boundary class, shown in Table 3.8, allows displacement boundary conditions over a number of Node objects to be defined. A Boundary object contains a BoolVector name DOFVector (typically with 3 vector elements) to set which degrees of freedoms (dofs) are restrained. After a Boundary object is created, any number of nodes that have this type of restraints can be added to its node list. A fixed boundary condition can be represented by creating an new Boundary object and setting all the values in the DOFVector property to “true”.

3.4.3.6 LumpedMass Class

Table 3.9 shows the members of the LumpedMass class. This class provides the facility to assign additional lumped nodal masses to the global mass matrix of the model. Once a LumpedMass object is created and its LumpedMassMatrix property is set, it can be assigned to a range Node objects.
### Table 3.7 Node class members

#### Public Static Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GetNodeAt</strong></td>
<td>Gets a node at the given coordinates (or close enough that the distance is less than the proximity) in the given domain.</td>
</tr>
<tr>
<td><strong>GetNodeWithName</strong></td>
<td>Searches the given Domain instance for a Node with a specified name property.</td>
</tr>
</tbody>
</table>

#### Public Instance Constructors

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Node Constructor</strong></td>
<td>Creates a new Node instance with default size position, displacement, velocity, and acceleration arrays. Also sets trace flag to false, interval to zero.</td>
</tr>
</tbody>
</table>

#### Public Instance Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Acceleration</strong></td>
<td>Gets or sets the acceleration vector of the Node instance.</td>
</tr>
<tr>
<td><strong>Displacement</strong></td>
<td>Gets or sets the displacement vector of the Node instance.</td>
</tr>
<tr>
<td><strong>Index</strong></td>
<td>Gets or sets the index of the Node instance. Set by NodeArray class during solver initialization.</td>
</tr>
<tr>
<td><strong>IsTraced</strong></td>
<td>Gets or sets a flag indicating whether the Node instance is traced or not. Also, depending on the flag, the trace table is either initialized or nullified.</td>
</tr>
<tr>
<td><strong>Position</strong></td>
<td>Gets or sets the position vector of the Node instance.</td>
</tr>
<tr>
<td><strong>TraceInterval</strong></td>
<td>Gets or sets the interval at which the nodal values will be saved to the trace table.</td>
</tr>
<tr>
<td><strong>TraceTable</strong></td>
<td>Gets the trace table which stores the history information. A null DataTable is returned if the Node instance is not traced.</td>
</tr>
<tr>
<td><strong>Velocity</strong></td>
<td>Gets or sets the velocity vector of the Node instance.</td>
</tr>
</tbody>
</table>

#### Public Instance Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GetCopy</strong></td>
<td>Overloaded. (Overridden) Creates a new Node instance which belongs to the same domain, and has the same position and trace settings as the original.</td>
</tr>
<tr>
<td><strong>GetForm</strong></td>
<td>(Overridden) Gets a Windows form which enables the user to view/edit the Node instance properties.</td>
</tr>
</tbody>
</table>
If the internal counter matches the trace interval, this method creates a DataRow from the displacement, velocity, and acceleration vectors for the current time step and adds it to the trace table. This method gets called if the solver’s `tracedMemberArray` field contains the Node instance. field.

### Protected Instance Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SaveTrace</code></td>
<td></td>
</tr>
</tbody>
</table>

### Table 3.8 Boundary class members

#### Public Instance Constructors

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Boundary Constructor</code></td>
<td>Creates a new boundary with 3 free dofs and zero sized node array.</td>
</tr>
</tbody>
</table>

#### Public Instance Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DOFVector</code></td>
<td>Gets or sets the dof’s that are restrained.</td>
</tr>
<tr>
<td><code>NodeArray</code></td>
<td>Gets or sets the array of nodes this boundary condition applies to.</td>
</tr>
</tbody>
</table>

#### Public Instance Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ContainsReferenceTo</code></td>
<td>(Overridden). Checks to see if the current Member instance contains reference to the specified IMember object in its properties.</td>
</tr>
<tr>
<td><code>GetForm</code></td>
<td>Gets a Windows form which enables the user to view/edit the Member instance properties.</td>
</tr>
</tbody>
</table>

#### Protected Instance Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GetDetails</code></td>
<td>Creates a string which represents all the properties of the Boundary instance.</td>
</tr>
</tbody>
</table>
Table 3.9 LumpedMass class members

**Public Instance Constructors**

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LumpedMass Constructor</strong></td>
<td>Creates a new LumpedMass instance, and assigns it a default (3 dof) mass matrix.</td>
</tr>
</tbody>
</table>

**Public Instance Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LumpedMassMatrix</strong></td>
<td>Gets or sets a DiagonalDoubleMatrix which represents the lumped masses at a node.</td>
</tr>
<tr>
<td><strong>NodeArray</strong></td>
<td>Gets or sets an array of Nodes which will have the prescribed lumped masses added to their position in the global mass matrix.</td>
</tr>
</tbody>
</table>

**Public Instance Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ContainsReferenceTo</strong></td>
<td>(Overridden) Checks to see if this IMember instance contains reference to the target object.</td>
</tr>
<tr>
<td><strong>GetForm</strong></td>
<td>(Overridden) Gets a Windows form which enables the user to view/edit the LumpedMass instance properties.</td>
</tr>
</tbody>
</table>

**Protected Instance Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GetDetails</strong></td>
<td>(Overridden) Creates a string which represents all the properties of the LumpedMass instance.</td>
</tr>
</tbody>
</table>
After the solver assembles the global lumped mass matrix from the element lumped mass matrices, it loops over the LumpedMass objects present in the model. For each LumpedMass object, the specified LumpedMassMatrix is added to the global mass matrix at the indices of the nodes referred by the NodeArray property.

### 3.4.3.7 Material Class

Table 3.10 shows the properties and methods contained in the Material base class, which is responsible for storing material properties. The classes that inherit from this class directly or indirectly, shown in Figure 3.11, all define new material properties. For example, the elastic class adds modulus of elasticity and poisson’s ratio properties to the density property that already exists in the base class.

This class is also responsible for creating integration points when the time-history analysis starts. The CreateIntegrationPoint(IElement sender) method ties integration point creation to a specific element type.

### 3.4.3.8 IntegrationPoint Class

The IntegrationPoint class is the base class that implements the IIntegrationPoint interface. The members of this class are listed in Table 3.11. This class and its child classes, shown in Figure 3.13, are responsible for the application of material laws (i.e. calculate and return the strains that correspond to the given strains, update material state, etc). Since solid, shell, and beam elements have different material matrices due to enforcement of zero out-of-
Table 3.10 Material class members

<table>
<thead>
<tr>
<th>Public Static Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GetMaterialWithName</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Public Instance Constructors</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Material Constructor</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Public Instance Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Density</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Public Instance Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CreateIntegrationPointInstance</strong></td>
</tr>
<tr>
<td><strong>GetForm</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Protected Instance Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GetDetails</strong></td>
</tr>
</tbody>
</table>
Table 3.11 IntegrationPoint class members

**Public Instance Constructors**

| IntegrationPoint Constructor | Initializes the integration point. Creates a default size (6) stress vector. (The strain vector is created by the owner element. |

**Public Instance Properties**

| Djb | Gets or sets the determinant of the Jacobian at the integration point. |
| Index | Gets or sets the Owner.Section.PositionArray, WeightArray, and MaterialArray index this IntegrationPoint instance’s position, weight, and material resides. |
| IsTraced | Gets or sets a flag indicating whether this integration point will be traced or not. Currently, if the owner element is traced, all of its integrations are set to be traced via Element.InitializeElement() method. This also initializes or nullifies the trace table depending on the flag. |
| Material | Gets the material of the integration point. In subclasses, this should set the specific material class field as well to avoid unnecessary unboxing during time history analysis. |
| Owner | Gets or sets the element this integration point belongs to. |
| Position | Gets the position vector of the integration point. |
| TraceInterval | Gets or sets the trace interval. |
| TraceTable | Gets the trace table. |
| Weight | Gets the weight of the integration point. |

**Public Instance Methods**

| GetStressVector | Gets the stress vector caused by given strains. |
| SaveTrace | If the internal counter matches the trace interval, this method creates a DataRow from the stress and strain vectors for the current time step and adds it to the trace table. This method gets called if the solver’s _tracedMemberArray field |
contains the owner element of this instance.

**Protected Instance Fields**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>strain</td>
<td>The strain vector at a given time.</td>
</tr>
<tr>
<td>stress</td>
<td>The stress vector at a given time.</td>
</tr>
<tr>
<td>traceCounter</td>
<td>The trace counter, zero means save every time step.</td>
</tr>
<tr>
<td>traceTable</td>
<td>The datatable for storing time history of the member.</td>
</tr>
</tbody>
</table>

**Protected Instance Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InitializeTraceTable</td>
<td>Initializes the trace table and adds the necessary columns to it.</td>
</tr>
</tbody>
</table>
plane stress conditions, as will be described in Chapter 4, the integration points are element type specific.

3.4.3.9 Section Class

This is the base class that implements the ISection interface. Unlike the other base classes in Snaps3D, such as the Material, Element, or IntegrationPoint, the Section class is not abstract. In fact, it represents the most general case, where the user is responsible for creating an array of material objects, an array of position vectors (in isoparametric coordinate system), and a double precision floating point number array which contains the integration point weights. It can be assigned to multiple elements. During time-history analysis, each element calls the GetElementIntegrationPoints(IElement sender) method, which creates an array of IntegrationPoint objects and assigns them to the “sender”.

The primary purpose of this class, the Material class, and their child classes are to save memory and streamline the input process. For example, in a model with four fully-integrated solid elements with same material properties, there are 32 integration points. Instead of storing 32 position vectors, only eight are stored in the PositionArray property of the Section instance. The index property of each integration point tells it which element of the PositionArray to use to get its coordinates, and returns a reference to it when asked. Similarly, instead of storing its own material properties, each IntegrationPoint instance references a Material object which holds those values for it.
# Table 3.12 Section class members

## Public Instance Constructors

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Section Constructor</strong></td>
<td>Creates a new section with default shear factor and zero sized position, weight and material arrays.</td>
</tr>
</tbody>
</table>

## Public Instance Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MaterialArray</strong></td>
<td>Gets or sets the materials of the integration points that will be attached to every element which has this section.</td>
</tr>
<tr>
<td><strong>PositionArray</strong></td>
<td>Gets or sets the position vectors (in the isoparametric coordinate system) of the integration points that will be attached to every element which has this section.</td>
</tr>
<tr>
<td><strong>ShearFactor</strong></td>
<td>Gets or sets the shear factor for the section.</td>
</tr>
<tr>
<td><strong>WeightArray</strong></td>
<td>Gets or sets the weights of the integration points that will be attached to every element which has this section.</td>
</tr>
</tbody>
</table>

## Public Instance Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ContainsReferenceTo</strong></td>
<td>(Overridden) Checks to see if the current IMember instance contains reference to the specified IMember object in its properties.</td>
</tr>
<tr>
<td><strong>GetElementIntegrationPoints</strong></td>
<td>Gets an array of integration points that are appropriate for the element type which calls this method.</td>
</tr>
<tr>
<td><strong>GetForm</strong></td>
<td>Gets a Windows form which enables the user to view/edit the Member instance properties.</td>
</tr>
</tbody>
</table>

## Protected Instance Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GetDetails</strong></td>
<td>Creates a string which represents all the properties of the Member instance.</td>
</tr>
</tbody>
</table>
The FullyIntegrated class can be used with any of the three element types currently incorporated to Snaps3D and creates an array with eight integration points that all have the same material. The position array holds the coordinates of the 2x2x2 gauss integration rule. Although, the Layered and Fibered classes can also be used with all three element types, they are suited for shell and beam type elements, respectively. The Layered class creates user specified number of layers each having point coordinates reflecting 2x2 gauss integration rule and one material. The Fibered class creates integration points using the rebar positions and material along with meshing the cross-section into a predetermined fibers along the width and height of the section.

3.4.3.10 Element Class

The Element class implements the IElememt interface and serves the base class for the continuum type integrated elements residing in the Snaps3D.Core.Elements namespace, shown in Figure 3.14. The primary responsibilities of the Element class and its child classes are to calculate the strains from the nodal displacements at each time step, and use the stress vector returned by each integration to calculate the internal force vector. Each element has a NodeArray property, as shown in Table 3.13, that references the Node objects that define its connectivity, a Section property that references the Section object that defines the integration point coordinates and materials, and IntegrationPointArray property that referenced the collection of IntegrationPoint objects the element owns.
Table 3.13 Element class members

**Public Instance Constructors**

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Element Constructor</strong></td>
<td>Creates a new Element with a null node array.</td>
</tr>
</tbody>
</table>

**Public Instance Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IntegrationPointArray</strong></td>
<td>Gets or sets the array of integration points this element has.</td>
</tr>
<tr>
<td><strong>IsTraced</strong></td>
<td>Gets or sets the time-history trace flag.</td>
</tr>
<tr>
<td><strong>NodeArray</strong></td>
<td>Gets or sets the array of nodes this element has.</td>
</tr>
<tr>
<td><strong>SaveBMat</strong></td>
<td>Gets or sets the flag for saving strain displacement matrices.</td>
</tr>
<tr>
<td><strong>Section</strong></td>
<td>Gets or sets the section which has the material and integration point coordinate/weight information for the element.</td>
</tr>
<tr>
<td><strong>TraceInterval</strong></td>
<td>Gets or sets the tracing interval (0=save every time step, 1=save every other time step).</td>
</tr>
<tr>
<td><strong>TraceTable</strong></td>
<td>Gets a null datatable (Element tracing not implemented).</td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>Gets the element type (to aid in appropriate integration point creation).</td>
</tr>
</tbody>
</table>

**Public Instance Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ContainsReferenceTo</strong></td>
<td>(Overridden) Checks to see if the current IMember instance contains reference to the specified IMember object in its properties.</td>
</tr>
<tr>
<td><strong>GetCopy</strong></td>
<td>(Overridden) Creates a new Element instance which belongs to the same domain, has the same tracing options and section as the original except the NodeArray is not set.</td>
</tr>
<tr>
<td><strong>GetExternalForce</strong></td>
<td>Gets the external force vector from the input body force vector.</td>
</tr>
<tr>
<td><strong>GetForm</strong></td>
<td>Gets a Windows form which enables the user to view/edit the Element instance properties.</td>
</tr>
<tr>
<td><strong>GetInternalForce</strong></td>
<td>Gets the internal force vector based on material law.</td>
</tr>
<tr>
<td><strong>GetLumpedMass</strong></td>
<td>Gets the lumped mass matrix of the element for the solver to use.</td>
</tr>
<tr>
<td><strong>InitializeElement</strong></td>
<td>Creates the integration points, if element is to be traced, this method reflects the</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>SaveTrace</strong></td>
<td>Prompts all the integration points in the element to save their time history information.</td>
</tr>
<tr>
<td><strong>Protected Instance Fields</strong></td>
<td></td>
</tr>
<tr>
<td>bMat</td>
<td>Saves strain displacement matrices at each integration point to speed up solution.</td>
</tr>
<tr>
<td>section</td>
<td>The elements section field.</td>
</tr>
<tr>
<td>traceCounter</td>
<td></td>
</tr>
<tr>
<td><strong>Protected Instance Methods</strong></td>
<td></td>
</tr>
<tr>
<td>GetDetails (Overridden)</td>
<td>(Overridden) Creates a string which represents all the properties of the IMember instance.</td>
</tr>
<tr>
<td>GetJacobian</td>
<td>Gets the jacobian matrix at an integration point (using the shape function gradients evaluated at the integration point).</td>
</tr>
<tr>
<td>GetShapeFunctionCartesianGradientsAt</td>
<td>Gets the shape function gradients, $dN/dx$, $dN/dy$, and $dN/dz$ at a given integration point.</td>
</tr>
<tr>
<td>GetShapeFunctionNaturalGradientsAt</td>
<td>Gets the shape function gradients, $dN/dr$, $dN/ds$, and $dN/dt$ at a given integration point.</td>
</tr>
<tr>
<td>GetShapeFunctionsAt</td>
<td>Gets the shape function vector, $N$, at a given integration point.</td>
</tr>
<tr>
<td>GetStrainDisplacementAt</td>
<td>Gets the strain displacement matrix $B$ at a given node index using the shape functions</td>
</tr>
<tr>
<td>GetStrainDisplacementMatrices</td>
<td>Gets all the strain displacement matrices by calling GetStrainDisplacementAt() method for each node and integration point in the element.</td>
</tr>
</tbody>
</table>
Currently, Solid, Shell, and Beam are the three continuum based elements that derive from the Element class. These elements will be discussed in detail in Chapter 4.

3.4.3.11 Loading Class

The primary role of the Loading class is to provide a facility to specify the loads acting on the model. Each Loading object has a scale factor and a database which stores the time and x, y, and z components of the load, as described in Table 3.14. The child classes of Loading class, shown in Figure 3.15, define where the loads are applied. NodalLoad class represents body loads that are applied to a set of Node objects, while the ElementLoad class represents body loads (per volume) that are applied to a set of Element objects. Seismic type objects are applied to the entire model and represent base acceleration.

3.4.3.12 Solver Class

The Solver class, shown in Table 3.15, currently incorporated into Snaps3D uses explicit central difference scheme to carry out time-integration. As mentioned in 3.4.3.2 the Domain class is the top level class in Snaps3D and stores all the model related information, such as nodes, elements, sections, loads, etc., in a collection accessible through the MemberList property. When the time-history analysis is started by calling the PerformTimeSteps method, partially shown in Figure 3.25, of the Solver class, the first task that is accomplished is to create specialized arrays.
### Table 3.14 Loading class members

#### Public Instance Constructors

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Loading Constructor</strong></td>
<td>Creates a new Loading with default scale factor, initializes the spreadsheet columns.</td>
</tr>
</tbody>
</table>

#### Public Instance Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Count</strong></td>
<td>Gets the number of points in the loading history.</td>
</tr>
<tr>
<td><strong>ScaleFactor</strong></td>
<td>Gets or sets factor to scale the load vector by.</td>
</tr>
<tr>
<td><strong>Table</strong></td>
<td>Gets or sets the spreadsheet which has the time vs loading history.</td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>Gets the type of the loading. The solver uses this property to determine where to use this loading.</td>
</tr>
</tbody>
</table>

#### Public Instance Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GetForce</strong></td>
<td>Gets the force vector at a given time.</td>
</tr>
<tr>
<td><strong>GetForm</strong></td>
<td>Gets a Windows form which enables the user to view/edit the Loading instance properties.</td>
</tr>
</tbody>
</table>

#### Protected Instance Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GetDetails</strong></td>
<td>Creates a string which represents all the properties of the IMember instance.</td>
</tr>
</tbody>
</table>

### Table 3.15 Solver class members

#### Public Instance Constructors

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Solver Constructor</strong></td>
<td>Initializes a new Solver with zero time steps and 0.001 time step size.</td>
</tr>
</tbody>
</table>

#### Public Instance Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time</strong></td>
<td>Gets the current time.</td>
</tr>
<tr>
<td><strong>TimeStepCount</strong></td>
<td>Gets or sets the number of time steps.</td>
</tr>
<tr>
<td><strong>TimeStepSize</strong></td>
<td>Gets or sets the time step size.</td>
</tr>
</tbody>
</table>

#### Public Instance Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GetForm</strong></td>
<td>Gets a Windows form which enables the user to view/edit the Solver instance properties.</td>
</tr>
<tr>
<td><strong>PerformTimeSteps</strong></td>
<td>Performs the time history analysis.</td>
</tr>
</tbody>
</table>
for (int timeStepNo = 1; timeStepNo <= TimeStepCount; timeStepNo++)
{
    // Increment time to n + 1
    _time += _timeStepSize;

    // Calculate global displacement vector at n + 1
    _nodeArray.Displacement += _timeStepSize * _nodeArray.Velocity
                           + (0.5 * _timeStepSize * _timeStepSize) * 
                     _nodeArray.Acceleration;

    // Enforce boundary conditions on the nodal displacement vectors:
    EnforceBoundaryConditions();

    // Get the body force at n + 1
    _f = GetBodyForce();

    // Calculate the global acceleration vector at n + 1
    _mf = _im * _f;

    // Calculate the global velocity vector at n+1 (_a is the global
    // acceleration vector at n)
    _nodeArray.Velocity += (_nodeArray.Acceleration + _mf) * (0.5 * _timeStepSize);

    // Update the global acceleration vector at n+1
    _nodeArray.Acceleration = _mf;

    SaveTracedMembers();
    percentComplete = (int)((double)(timeStepNo + 1) / 
                           (double)TimeStepCount * 100);
    worker.ReportProgress(percentComplete);
}

Figure 3.25 Solver class time-history integration
by filtering the information on the domain’s MemberList. These specialized arrays include a node array and a boundary array, which contain Node objects and Boundary objects only, respectively. Using the number of the nodes in the domain, the global mass matrix is created. Next, specialized element array, which contains all the objects that implement the IElement interface in the MemberList. Each element in the element array is initialized (their integration points created), its mass matrix is added to the global mass matrix. Other initial tasks include filtering the LumpedMass objects from the MemberList and adding them to the global mass matrix, creating a specialized array for objects that are type of Loading (this includes the child classes of Loading as well), and creating an array containing all the traced members (currently, nodes, elements, and integration points only).

3.5 Summary

In this chapter, the suitability of .NET framework and C# programming language for development of finite element libraries were discussed. Module, object and component oriented programming approaches were described. The organization of the Snaps3D matrix and core class libraries were presented and major interfaces and classes, their derivation, and their properties and methods were described.
Chapter 4

Mathematical Models for Precast Members and Structures

4.1 General

The finite element modeling techniques that are used in cast-in-place reinforced concrete buildings under lateral loads are generally not applicable to precast concrete buildings, unless the emulative design approach is used as described in Chapter 2. In a typical precast concrete building, the connections are the planes of weakness in terms of strength and stiffness.

This study uses various prominent models that analyze behavior of the precast concrete structures while a consideration is given to their limitations. In this chapter, the modeling strategies used for analysis of precast concrete structures are discussed. The finite elements and the material models used for modeling the components such as wall panels, beams, columns, and connections are described.

4.2 Review of Previous Models

There have been previous attempts to model the response of precast building structures under seismic activity. Three main weaknesses of these models are 1) they generally use two-dimensional analysis, 2) wall systems or
frames are isolated from the remainder of the structure with the assumption of rigid diaphragms, and 3) concentration of nonlinear action occurs only at the connections.

Section 4.2.1 describes the finite element modeling techniques used for precast concrete shear wall buildings. In Section 4.2.2, the models for frame type structures are discussed.

### 4.2.1 Shear Walls

Previous analytical studies by other researchers have been limited to the analyses of simple walls or coupled shear walls with various horizontal and vertical connection models (Caccese and Harris 1987; Kianoush et al. 1996; Kianoush and Scanlon 1986; Pekau and Hum 1991; Schricker and Powell 1980). The wall panels are modeled using plane-stress elements with linear-elastic material properties. In some cases, modified beam elements are used instead of plane stress elements to achieve greater computational efficiency at the expense of some accuracy (Schricker and Powell 1980).

In the study by Schricker and Powell (1980), several elements to model the behavior of the connections under shear forces, axial forces, or both are developed and incorporated into the finite element program called DRAIN2D (Kanaan and Powell 1973). Figure 4.1 and Figure 4.2 show the degree of freedom (DOF) and force-displacement relationships of the shear-slip element, respectively. Figure 4.3 and Figure 4.4 show the DOF and force-displacement relationships of the gap element, respectively. One of the primary characteristics of these elements is that they all are zero-thickness elements. It is also
Figure 4.1 Shear-slip element dof (Schricker and Powell 1980)

Figure 4.2 Shear-slip element force-displacement relationship (Schricker and Powell 1980)
Figure 4.3 Gap element dof (Schricker and Powell 1980)

Figure 4.4 Gap element force-displacement relationship (Schricker and Powell 1980)
significant that these elements are not integrated and their material properties are provided as force displacement relationships. This requires the stiffness values to be calculated based on an effective area using the stress-strain curves. As shown in Figure 4.4, the behavior of the gap element under compression represents a tri-linear curve when the tensile resistance of the connection material is ignored.

In the study by Kianoush and Scanlon (1986), the connection models are based on continuum type elements which require stress-strain type relationships, as opposed to force-displacement in the previous case. The wall panels are modeled using linear-elastic plane stress elements and the connections are modeled using four-node quadrilaterals, where the normal stress component parallel to the connection orientation is ignored as shown in Equation (4.1). This results in the 2x2 material matrix shown in Equation (4.2). Furthermore, the shear stress is assumed to remain constant over the connection thickness. \[
\{ \varepsilon \} = \begin{bmatrix} \varepsilon_x \\ \gamma_{xy} \end{bmatrix} \quad (4.1)
\]
\[
[D] = \begin{bmatrix} E & 0 \\ 0 & G \end{bmatrix} \quad (4.2)
\]
where,
\[
E = \text{Axial tangent modulus per unit thickness.}
\]
\[
G = \text{Shear tangent modulus per unit thickness}
\]

The material models developed for the connection element include 1) a linear elastic model, where the axial response is linear-elastic, 2) a zero-tension model, where the axial resistance was zero under tensile forces, and 3) a more
complex multi-linear model based on a modified version of the concrete model that is developed by Darwin and Pecknold (1974). The parabola representing the ascending part of the curve is replaced with a bilinear relationship in favor of computational efficiency. Figure 4.5 shows the modified concrete curve and unloading/reloading paths under cyclic loading. The stress and tangent modulus for a given compressive stress is as follows:

For $0 < \varepsilon' \leq 0.425 \varepsilon_c$:

$$\sigma' = E_0 \varepsilon' \left( \frac{1}{1 + \frac{E_0}{2 E_s}} + \frac{1}{2} \right)$$  \hspace{1cm} (4.3)$$

$$E_1 = E_0 \left( \frac{1}{1 + \frac{E_0}{2 E_s}} + \frac{1}{2} \right)$$  \hspace{1cm} (4.4)$$

For $0.425 \varepsilon_c < \varepsilon' \leq \varepsilon_c$:

$$\sigma' = \sigma_c - 0.425 \varepsilon_c E_1 \left( \frac{1 - \frac{\varepsilon'}{\varepsilon_c}}{0.575} \right)$$  \hspace{1cm} (4.5)$$

$$E_2 = \frac{\sigma_c - 0.425 \varepsilon_c E_1}{0.575 E_c}$$  \hspace{1cm} (4.6)$$

where,

$\sigma'$ = Stress at time $t$.

$\varepsilon'$ = Strain at time $t$.

$\varepsilon_c$ = Strain corresponding to peak compressive stress.

$E_0$ = Initial tangent modulus.
Figure 4.5 Concrete stress-strain model (Kianoush and Scanlon 1986)
\[ E_s = \text{Secant modulus.} \]

The shear stress-strain behavior is represented using three different models: 1) a linear–elastic model, 2) a shear-slip model, where it is assumed that slip will occur when the shear stresses exceed the stress value calculated as a function of the compressive stress acting on the section and coefficient of friction, and 3) a shear-friction model, where the reinforcement crossing the connection plane provides clamping action as it develops tension.

The coupling beams are modeled using different approaches based on the depth to span ratios. Slender beams are modeled using beam elements with plastic hinges at the ends and deep beams are modeled using inelastic trusses in a x-bracing type configuration.

### 4.2.2 Frame Models

As described in the previous section, in precast concrete shear walls nonlinear behavior takes place in the horizontal and vertical connections unless the emulative design approach is used. In the case of precast concrete frames with PRESSS style ductile connections, the emphasis is placed on modeling the nonlinear behavior of beam-column sub-assemblages (Priestley 1996).

Figure 4.6 and Figure 4.7 show the finite element meshes used to model a post-tensioned precast frame sub-assembly using spring and fiber elements, respectively (El-Sheikh et al. 1999). In the former, the nonlinearity in the column-beam connection region is modeled using a rotational spring element tied to the center of the panel zone using rigid links. The column and beam segments
Figure 4.6 Frame sub-assembly model with spring elements (El-Sheikh et al. 1999)

Figure 4.7 Frame sub-assembly model with fiber elements (El-Sheikh et al. 1999)
outside the connection region are modeled using linear-elastic beam elements.
In the latter, the spring elements are replaced with truss and fiber beam-column
elements. The truss elements are used to model the post-tensioning steel. The
column and beam segments inside the connection region are modeled using
nonlinear fiber beam-column elements.

Both the spring and fiber elements are developed as a part of the
DRAIN2DX computer program (Powell 1993). The spring element is a zero-
length two-node element with one DOF/node that can be used to model a
translational or rotational spring. The force-displacement or moment-rotation
relationship can be bilinear inelastic, bilinear elastic, or bilinear inelastic with gap.
The fiber element contains multiple layers across the cross-section as shown in
Figure 4.8. Each of these layers can be assigned a concrete stress-strain
relationship (different stress-strain points to define the compression and tension
parts of the curve) or a steel stress-strain relationship (compression and tension
curves are same).

4.3 Element Models

The element models incorporated to SNAPS3D are all derived from the
Element class described in Section 3.4.3.10, and are used to implement the
IElement interface described in Section 3.4.2.6. Figure 4.9 shows the complete
inheritance diagram, including the Object and MarshalByRefObject classes that
are a part of the .NET BCL. The solid, shell, and beam elements described
below are all eight-node continuum elements with three translational DOF at
each node.
Figure 4.8 Fiber beam-column element for DRAIN2DX (Powell 1993)
Figure 4.9 Class inheritance diagram for Element classes
Element models can be grouped into solid, shell, and beam elements as described in Sections 4.3.1, 4.3.2, and 4.3.3, respectively.

4.3.1 Solid Element

The derivation of element matrices for an eight-node solid element is well established (Bathe 1996; Chandrupatla and Belegundu 1997; Hughes 1987). The geometry interpolation for the solid element shown in Figure 4.10 can be written as:

\[ x = x_i N_i(\xi, \eta, \zeta) \]  
(4.7)

where the Lagrange shape functions are:

\[ N_i(\xi, \eta, \zeta) = \frac{1}{8} (1 + \xi_i \xi)(1 + \eta_i \eta)(1 + \zeta_i \zeta) \]  
(4.8)

using the chain rule:

\[
\begin{bmatrix}
\frac{\partial N_i}{\partial \xi} \\
\frac{\partial N_i}{\partial \eta} \\
\frac{\partial N_i}{\partial \zeta}
\end{bmatrix} =
\begin{bmatrix}
\frac{\partial N_i}{\partial x} & \frac{\partial N_i}{\partial y} & \frac{\partial N_i}{\partial z}
\end{bmatrix}
\begin{bmatrix}
\frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} & \frac{\partial x}{\partial \zeta}
\end{bmatrix}
\begin{bmatrix}
\frac{\partial N_i}{\partial x} \\
\frac{\partial N_i}{\partial y} \\
\frac{\partial N_i}{\partial z}
\end{bmatrix}
\]  
(4.9)

or

\[
\begin{bmatrix}
\frac{\partial N_i}{\partial \xi} \\
\frac{\partial N_i}{\partial \eta} \\
\frac{\partial N_i}{\partial \zeta}
\end{bmatrix} = \mathbf{J} \begin{bmatrix}
\frac{\partial N_i}{\partial x} \\
\frac{\partial N_i}{\partial y} \\
\frac{\partial N_i}{\partial z}
\end{bmatrix}
\]  
(4.10)

where, the Jacobian matrix is:
Figure 4.10 Solid element with eight nodes
The displacement interpolation for an eight node solid element is:

\[ u = u_i N_i(\xi, \eta, \zeta) \]  

The strain-displacement relations can be written as:

\[ \varepsilon = B u \]  

or

\[ \varepsilon = B_i u_i \]

where

\[ B_i = \begin{bmatrix} \frac{\partial}{\partial x} & 0 & 0 \\ 0 & \frac{\partial}{\partial y} & 0 \\ 0 & 0 & \frac{\partial}{\partial z} \end{bmatrix} N_i \]

and

\[ J = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} & \frac{\partial z}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} & \frac{\partial z}{\partial \eta} \\ \frac{\partial x}{\partial \zeta} & \frac{\partial y}{\partial \zeta} & \frac{\partial z}{\partial \zeta} \end{bmatrix} \]
Once the stress vector is calculated using the material law, the internal force vector can be calculated from:

\[ \mathbf{f}_{\text{int}} = \int \mathbf{B}^T \mathbf{\sigma} \, d\Omega \]  \hspace{1cm} (4.17)

where

\[ \mathbf{\sigma} = \begin{bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{zz} \\ \sigma_{xy} \\ \sigma_{yz} \\ \sigma_{zx} \end{bmatrix} \]  \hspace{1cm} (4.18)

The consistent mass matrix is:

\[ \mathbf{M} = \int \rho \mathbf{N}^T \mathbf{\mathbf{N}} \, d\Omega \]  \hspace{1cm} (4.19)

The lumped mass matrix used by the solver is obtained by lumping the rows of the consistent mass matrix in Equation (4.19). Since the current version of SNAPS3D has an explicit solver, none of the element classes has a method to calculate the stiffness matrix.

The solid element is incorporated to SNAPS3D as a general purpose element. While it can be used to model any component of a precast structure, its primary purpose is to serve as a stepping stone between the base Element class
and the Shell and Beam classes. Solid elements with more nodes can easily be
derived from this class provided that new classes override the methods which
return the shape functions and shape function gradients with their versions
suitable for the new element type. Furthermore, these higher order solid
elements can use the same type of integration points as the basic Solid element
with eight nodes which are described here.

### 4.3.2 Shell Element

The eight-node shell element, shown in Figure 4.11, incorporated to
SNAPS3D is based on the solid shell formulation that is used in a program called
LS-DYNA (Hallquist 1998). It uses the same shape functions and strain-
displacement relations as the solid element described in Section 4.3.1. However,
it introduces a local coordinate system at the mid-plane between the top shell
surface (denoted by nodes 5, 6, 7, and 8) and the bottom shell surface (denoted
by nodes 1, 2, 3, and 4). The position vector to the corner of the mid-plane:

\[ \mathbf{r}_i = \frac{1}{2} (\mathbf{x}_i + \mathbf{x}_{i+4}) \quad (i = 1, 4) \]  

(4.20)

The normal to the mid-plane is:

\[ \mathbf{s}_3 = \mathbf{r}_{31} \times \mathbf{r}_{42} \]  

(4.21)

where,

\[ \mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j \]  

(4.22)

The unit outward normal can be obtained using Equation (4.21) as:

\[ \mathbf{e}_3 = \frac{\mathbf{s}_3}{|\mathbf{s}_3|} \]  

(4.23)
Figure 4.11 Shell element with eight nodes
The other unit vectors defining the local coordinate system can be selected as:

\[
e_1 = \frac{s_1}{|s_1|}
\]  

(4.24)

and,

\[
e_2 = e_3 \times e_1
\]  

(4.25)

where,

\[
s_1 = r_{21} - (r_{21} \cdot e_3) e_3
\]  

(4.26)

The global-to-local transformation matrix is:

\[
q = \begin{bmatrix} e_{11} & e_{12} & e_{13} \\
e_{21} & e_{22} & e_{23} \\
e_{31} & e_{32} & e_{33} \end{bmatrix}^T
\]  

(4.27)

or

\[
q = \begin{bmatrix}
e_{11} & e_{12} & e_{13} \\
e_{21} & e_{22} & e_{23} \\
e_{31} & e_{32} & e_{33} 
\end{bmatrix}
\]  

(4.28)

The additional transformation step takes place when the strains calculated using Equation (4.14) are converted to a local coordinate system. The stresses are evaluated in the local coordinate system using the “reduced constitutive equation” (Hughes 1987), which will be described in Section 4.4.1. The shell element then transforms these stresses back into the global coordinate system to use in internal force calculations.

As described in 4.2.1, four-node plane-stress elements are used by other researchers to model wall panels using two-dimensional modeling approaches. The eight-node shell element presented in this section represents a natural three-dimensional substitute for the plane-stress element. When multiple through-the-thickness integration points are used, it can be used to model the
wall panels with reinforcement layers, floor planks, ramps, etc. It can also be used to model the horizontal and vertical connections if the necessary nonlinear material models are supplied.

4.3.3 Beam Element

The beam element, shown in Figure 4.12, is essentially the same as the shell element described in the previous section. The integration points owned by the beam elements enforce zero out-of-plane condition in all directions but the local x axis. The stresses in local y and z axes are assumed to be zero.

This element can be used to model beam and column elements, wall connections, post-tensioning steel when used alone, and frame connections when coupled with other beam elements. The nonlinear material models, which will be described in the next section, are developed for this type of element.

4.4 Material Models

In SNAPS3D, the material models are grouped into elastic, rebar, gap, and shear slip models as described in Sections 4.4.1, 4.4.2, 4.4.3, and 4.4.4, respectively.

4.4.1 Elastic Model

Linear elastic material is a fictitious material whose displacement response is proportional to the force acting on it by a constant factor, as shown in
Figure 4.12 Beam element with eight nodes

Figure 4.13 and Figure 4.14. Most materials behave linear elastically under small loads.
For the general three-dimensional case, the linear-elastic constitutive matrix can be written as:

$$\mathbf{D} = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & \nu & 0 & 0 & 0 \\ \nu & 1-\nu & \nu & 0 & 0 & 0 \\ \nu & \nu & 1-\nu & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.5 - \nu & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.5 - \nu & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.5 - \nu \end{bmatrix}$$ (4.29)

For shell elements, where zero out-of-plane stress condition (or plane stress condition) applies, Equation (4.29) has to be modified to reflect this condition. By solving the zz component of the strain in terms of other strain components (Hughes 1987), the reduced constitutive matrix (5x5) can be obtained as:

$$\mathbf{D} = \frac{E}{1-\nu^2} \begin{bmatrix} 1 & \nu & 0 & 0 & 0 \\ \nu & 1 & 0 & 0 & 0 \\ 0 & 0 & \frac{1-\nu}{2} & 0 & 0 \\ 0 & 0 & 0 & \frac{1-\nu}{2} & 0 \\ 0 & 0 & 0 & 0 & \frac{1-\nu}{2} \end{bmatrix}$$ (4.30)

The strain terms corresponding to the reduced constitutive matrix are:

$$\mathbf{\varepsilon} = \begin{Bmatrix} \varepsilon_{xx} \\ \varepsilon_{yy} \\ \varepsilon_{xy} \\ \varepsilon_{yz} \\ \varepsilon_{zx} \end{Bmatrix}$$ (4.31)
Figure 4.13 Axial behavior of linear-elastic material

Figure 4.14 Shear behavior of linear-elastic material
For beam elements, the yy component of stress is also assumed to be zero. By solving the yy component of strain in terms of other strain components in Equation (4.30) or alternatively, condensing both the yy and zz components of the strain in Equation (4.29) the reduced constitutive matrix linear elastic material can be used with beam elements can be obtained as:

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & \frac{1}{2(1+\nu)} & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & \frac{1}{2(1+\nu)}
\end{bmatrix}
\]

\[D = E \quad (4.32)\]

The strain terms corresponding to the reduced constitutive matrix are:

\[
\begin{bmatrix}
\varepsilon_{xx} \\
\varepsilon_{xy} \\
\varepsilon_{yz} \\
\varepsilon_{zx}
\end{bmatrix}
\]

\[\varepsilon = \begin{bmatrix}
\varepsilon_{xx} \\
\varepsilon_{xy} \\
\varepsilon_{yz} \\
\varepsilon_{zx}
\end{bmatrix} \quad (4.33)\]

As described in Chapter 2, when ductile connections are used as primary means for joining precast components, nonlinear behavior occurs at the connections. Since the wall panels, beams, and columns remain essentially linear-elastic, there is no reason to use computationally intensive nonlinear material models. Using a linear-elastic material model is suitable for these members.
4.4.2 Rebar Model

The axial behavior of reinforcement is represented using a bilinear stress-strain relationship as shown in Figure 4.15. The unloading/reloading behavior of the material depends on whether isotropic or kinematic hardening is selected.

For kinematic hardening, the yield strengths in the current time step are calculated using:

\[ f_{yt}^t = f_{y0} + H\varepsilon_p^{t-1} \]
\[ f_{yc}^t = -f_{y0} + H\varepsilon_p^{t-1} \]  \hspace{1cm} (4.34)

where,

- \( f_{yt}^t \) = Yield strength in tension at time \( t \).
- \( f_{yc}^t \) = Yield strength in compression at time \( t \).
- \( f_{y0} \) = Initial yield strength.
- \( \varepsilon_p^{t-1} \) = Plastic strain at time \( t-1 \).
- \( H \) = Hardening parameter \( H = \frac{EE_T}{(E - E_T)} \)
- \( E \) = Modulus of elasticity.
- \( E_T \) = Tangent modulus.

The trial stress is evaluated using:

\[ f_{trial}^t = E(\varepsilon^t - \varepsilon_p^{t-1}) \]  \hspace{1cm} (4.35)

If \( f_{trial}^t < f_{yc}^t \), yielding in compression
Figure 4.15 Axial behavior of rebar material
\[ f^t = f_{yc}^t + \left( \frac{E}{\varepsilon} \right) f_{\text{trial}}^t - f_{yc}^t \] 
\[ \varepsilon_p^t = \varepsilon^t - \frac{f^t}{E} \] 

Else if \( f_{\text{trial}}^t > f_{yt}^t \), yielding in tension

\[ f^t = f_{yt}^t + \left( \frac{E}{\varepsilon} \right) f_{\text{trial}}^t - f_{yt}^t \] 
\[ \varepsilon_p^t = \varepsilon^t - \frac{f^t}{E} \] 

Else, unloading or reloading

\[ f^t = f_{\text{trial}}^t \] 

For isotropic hardening, the calculation of the elastic trial stress is also carried out using Equation (4.35). However, the state determination is performed differently.

If \( |f_{\text{trial}}^t| > f_{y}^{t-1} \), the material yielded (in compression or tension)

\[ f^t = \text{Sign}(f_{\text{trial}}^t) f_{y}^{t-1} + \left( \frac{E}{\varepsilon} \right) (f - \text{Sign}(f_{\text{trial}}^t) f_{y}^{t-1}) \] 
\[ \varepsilon_p^t = \varepsilon^t - \frac{f^t}{E} \] 
\[ f_{y}^{t} = |f^t| \] 

Else, material is unloading/reloading. Use Equation (4.38) to calculate the stress.

The behavior of rebar material in shear is assumed to be linear-elastic and shear-axial stresses are not considered. This material is used to model the reinforcement in beam, post-tensioning steel, and unbonded mild reinforcement in beam-column connections.
4.4.3 Gap Model

The gap model used in this study is based on the uniaxial concrete law proposed by Saenz (1969) that is used in another study to represent the descending branch of the concrete stress-strain curve (Balan et al. 2001). The equation is:

\[
\sigma = f_c \frac{K \left( \frac{\varepsilon}{\varepsilon_c} \right)}{1 + A \left( \frac{\varepsilon}{\varepsilon_c} \right) + B \left( \frac{\varepsilon}{\varepsilon_c} \right)^2 + C \left( \frac{\varepsilon}{\varepsilon_c} \right)^3}
\]  

(4.40)

where,

\[
K = \frac{E_0 \varepsilon_c}{f_c}
\]

\[
K_e = \frac{\varepsilon_{control}}{f_c}
\]

\[
K_\sigma = \frac{f_c}{f_{control}}
\]

\[
A = C + K - 2
\]

\[
B = 1 - 2C
\]

\[
C = K \left( \frac{(K_\sigma - 1)}{(K_e - 1)^2} - \frac{1}{K_e} \right)
\]

(4.41)

\[
\varepsilon_c, f_c = \text{The strain and stress values corresponding to the peak point on the compression curve.}
\]

\[
\varepsilon_{control}, f_{control} = \text{The strain and stress values corresponding to a post-peak point on the compression curve and define how fast the material softens.}
\]

\[
E_0 = \text{Initial tangent modulus of concrete.}
\]
Figure 4.16 shows the unloading/reloading behavior of the concrete model used in this study. Unlike the concrete models used in previous studies which unload parallel to the initial tangent modulus, the concrete model used here is assumed to unload towards a focal point \((\varepsilon_f, f_f)\) as proposed by Lee and Willam (1997). The location of the focal point is normally determined based on experimental observations. However, for computational simplicity, the location of the focal point might be assumed to be at the peak point on the tension curve (Kwon 2000). If the stress and strain values denoting the focal point are selected to be very large numbers on the line emanating from the origin at a slope equal to initial tangent modulus, the unloading/reloading will be approximately parallel to the initial tangent modulus similar to the unloading rule that is used in previous studies (Kianoush and Scanlon 1986; Schricker and Powell 1980).

The equations governing hysteretic behavior are as follows:

1. Evaluate the trial stress at time \(t\):

   \[
   f_{trial}^t = E_T^{t-1} \left( \varepsilon^t - \varepsilon_p^{t-1} \right) \tag{4.42}
   \]

2. If \(f_{trial}^t \leq f_{envelope}^t\), loading takes place following the compression envelope determined by Equation (4.40). Update material properties as:
This model is used to describe the material behavior of connections in axial direction. The tensile resistance of concrete is ignored. As a result, if the strains become positive (tensile), the resistance is lost and a gap develops. The gap model described in this section is intimately linked to the shear-slip model described in the next section.

### 4.4.4 Shear-Slip Model

The shear-slip model used by Kianoush and Scanlon (1986) is adopted in this study. Figure 4.17 shows the cyclic behavior of the shear-slip model. If the gap is open, all the shear strain at time $t$ is assumed to directly add to the accumulated slip. If the gap is closed, the current yield strength in shear is determined as a factor of coefficient of friction and the current axial compressive stress determined from the concrete material model described in the previous section.
Figure 4.16 Axial behavior of concrete material
Figure 4.17 Shear-slip behavior of concrete.
The gap and shear-slip models are collectively called the “concrete model”. While the former models the behavior of connections under axial stresses, the latter models the behavior of connections under shear stresses.

4.5 Summary

In this chapter, the methodologies used in the past to model the behavior of precast concrete shear walls and precast concrete frames were reviewed. The elements and material models incorporated into SNAPS3D from these previous studies and the precast concrete components they represent were discussed.

While two dimensional modeling approaches would require frames or walls to be isolated from the rest of the structure, the elements presented here enable researchers to evaluate a building as a whole without the restrictions posed by two-dimensional modeling techniques.
Chapter 5

Application Examples

5.1 General

This chapter contains several example problems solved using SNAPS3D to verify the implementation of core finite element classes and to demonstrate how precast building components can be modeled using the element and material classes currently implemented in SNAPS3D. The examples include elastic analysis of an L-shaped beam under static load, elastic analysis of a simple wall under base acceleration, and nonlinear analysis of a precast concrete shear wall with horizontal connection under various loading conditions. Since the current version of SNAPS3D does not contain a static solver, the static loads in the L-shaped beam case or the gravity load in the precast concrete shear wall case are applied as a function of time under a high mass proportional damping before application of any lateral loads.

5.2 Elastic Analysis of an L-Shaped Beam under Static Load

Figure 5.1 shows the node and element numbering of the finite element mesh of an L-shaped beam loaded with a single concentrated load of 80 kN. The displacement at node 1 in z-direction is determined as -0.04098 mm using
Figure 5.1 Finite element mesh of L-shaped beam (Chandrupatla and Belegundu 1997)
the computer program HEXAFNT (Chandrupatla and Belegundu 1997). The mesh consists of four fully integrated eight-node solid elements and the material properties are:

\[ E = 200 \text{ GPa} \]
\[ \nu = 0.3 \]

Currently, the only solver incorporated into SNAPS3D is for carrying out dynamic analysis using explicit time-integration. To obtain a solution for a static loading case, either the load can be applied very slowly or a large artificial damping can be used.

In this case mass proportional damping is used and the loading is applied instantaneously. The mass-density of the material is assumed to be 7.85E-6 kg/mm\(^3\) and a time step of 0.0001 seconds is used. Figure 5.2 shows the displacement histories of node 1 in z-direction for undamped (\(\alpha=0\)) and damped (\(\alpha=3\) and \(\alpha=8\)). For the undamped case the solution oscillates about the displacement calculated using static analysis. For the damped cases, the solution converges to the displacement calculated using HEXAFNT. The speed of the converge depends on the value assumed for mass proportional damping coefficient.

5.3 Elastic Analysis of a Simple Wall Subjected to Base Acceleration

Figure 5.3 shows the finite element mesh of a simple wall-like structure which is subjected to a base acceleration in the form of a triangular pulse. The overall dimensions of the model are 100 mm by 200 mm by 300 mm. The
Figure 5.2 Displacement history with different damping coefficients.
Figure 5.3 Abaqus model of the wall in Example 2.
problem is solved by ABAQUS/Explicit (ABAQUS Inc. 2005) using selectively reduced integrated eight-node solid elements. The nodes located on the Y=0 plane is restrained in Y and Z directions, and base acceleration is applied to those nodes in X direction. The applied base acceleration decreased from 0.5g at start to zero at 0.2 milliseconds linearly. In the SNAPS3D model, fully integrated eight-node solid elements are substituted in place of selectively-reduced integrated eight node solid elements and a constant time step of 0.0001 milliseconds is used. The material properties of the solid elements are as follows:

\[
\begin{align*}
\rho &= 2400 \text{ kg/m}^3 \\
E &= 26.25 \text{ GPa} \\
\nu &= 0.2
\end{align*}
\]

Figure 5.4 shows the displacement history of the roof relative to the base during the base acceleration. During the application of the load, the results from both programs are very similar. However, once the load is removed, the period of the SNAPS3D model is slightly smaller which may be attributed to increased stiffness of fully integrated solids.

### 5.4 Nonlinear Analysis of a Simple Wall Subjected to Concentrated Loads

This example contains two precast wall panels joined by a horizontal connection as shown in Figure 5.5 and has been used in a previous study by Soudki et al. (1995a) The top panel is 2100 mm high, 1200 mm wide, 152 mm thick and the bottom panel is 600 mm high, 1200 mm wide, 152 mm thick. The concrete in both panels has a nominal strength of 40 MPa. The connection is 20
Figure 5.4 Roof displacement history
Figure 5.5 Shear wall subassembly with horizontal connection
mm high and is made of drypack with a nominal strength of 48 MPa. The bottom panel is restrained at the edges against lateral and vertical movement.

For the panels, an elastic type material is used with the following properties:

\[
\rho = 2400 \text{ kg/m}^3 \\
E = 32 \text{ GPa} \\
\nu = 0.15 \\
\]

For the connection, a concrete type material with the following properties is used. The high values are assigned to focal strain and strain values to force the unloading behavior to take place parallel to the initial tangent modulus.

\[
\rho = 2400 \text{ kg/m}^3 \\
\varepsilon_{cc, f_c} = -0.002, -48 \text{ MPa} \\
\varepsilon_{control, f_{control}} = -0.0035, -31 \text{ MPa} \\
\varepsilon_f, f_f = 1, 33 \text{ GPa} \\
E_0 = 33 \text{ GPa} \\
\nu = 0.15 \\
\mu = 0.5 \\
G_s = 1.434 \text{ GPa} \\
\]

The top and bottom panels are meshed in 28 and 8 shell elements measuring 300 mm by 300 mm and are 152 mm thick, respectively. Each shell element is assigned a section property with 5 equal thickness layers. Each layer was assigned the elastic type material described above. The connection is meshed into 4 beam elements measuring 300 mm wide, 152 mm high, and 20
mm long. Each beam element is assigned a section property with 20 fibers (5 along width x 4 along height). Each fiber is assigned the concrete type material described above. The mesh of shell and beam elements to model the panels and the connection is shown in Figure 5.6.

Concentrated vertical loads of 22.8 kN and 45.6 kN are applied to the corner nodes and edge nodes at the roof level in vertical direction to simulate gravity loads, respectively. The gravity loads are applied slowly at the start of the analysis and then kept constant afterwards. Several lateral loading cases are considered. In all these cases the lateral loads are applied at the 4 corner nodes 300 mm below the roof level.

Figure 5.7 shows the lateral displacement time-history of the roof level when lateral loads are applied in a monotonically increasing manner between 0.02 seconds and 0.04 seconds and kept constant at 80 kN afterwards. The mass proportional damping coefficient is assumed to be 900 during the application of gravity loads (t < 0.02 s) and 90 afterwards. The chart also includes the time-history of the same model when connection is assumed to remain linear elastic. Because of the high damping, the structure does not oscillate during the application of the gravity loads. However, when the damping coefficient is reduced, the oscillation of the structure becomes noticeable. While the displacements of the nonlinear connection model is only slightly larger than the elastic model, the period of the nonlinear model is about 30% bigger.

Figure 5.8 shows the applied force vs. roof lateral displacement when lateral load is applied in a cyclic manner and at a much larger magnitude (320
Figure 5.6 Finite element mesh of the shear wall
Figure 5.7 Comparison of elastic and nonlinear connection materials
The softening of the structure is apparent as the magnitude of the load is increased in positive direction. When the load is reversed, the stiffness of the structure is initially smaller due to open gap in the positive loading cycle. As the gap closes, the stiffness of the structure increases. The chart also shows that there is a small strength degradation after one loading cycle. Figure 5.9 shows the hysteretic behavior at the connection integration located closest to the tip of the wall. The stress-strain envelope defined by the Saenz equation described in Section 4.4.3 is plotted on the chart using a continuous line. Following the first load reversal, the plastic axial strain is 0.004, and after the second load reversal, the plastic axial strain becomes slightly larger than 0.005.

For comparison purposes Figure 5.8 also contains the applied force-lateral displacement history of a “reinforced” connection. The primary difference between the unreinforced and reinforced models is the additional beam element which is connected to the corner nodes of the model in the latter model to simulate unbonded reinforcement. This beam element has only two fibers with Grade 400 reinforcing bar properties. The diameter of each bar is 25.4 mm and the bars are located 150 mm from each side of the wall. The addition of the reinforcement bars increase the stiffness of the wall and reduce the magnitude of lateral displacements about 25%.

5.5 Accuracy of Continuum Based Beam Element

In this example, a cantilever beam loaded at one end with a concentrated load is modeled using 8-noded continuum-based beam elements. This example
Figure 5.9 Hysteresis at connection integration point
is used to verify the accuracy of the continuum-based beam implementation in
Snaps3D. The beam dimensions are as follows:

\[
\begin{align*}
b &= 200 \text{ mm} \\
h &= 500 \text{ mm} \\
L &= 5000 \text{ mm}
\end{align*}
\]

Each element is assigned 20 fibers along their height with the following
material properties:

\[
\begin{align*}
\rho &= 2.40 \cdot 10^{-6} 2.40 \text{ kg/mm}^3 \\
E &= 33 \text{ kN/mm}^2 \\
\nu &= 0.15
\end{align*}
\]

The time step size is 0.001 ms and the mass proportional damping
coefficient is 0.1. Figure 5.10 shows the displacement time history of the free
end of the cantilever beam for three different meshes. Of the three meshes, the
first one contains a single element to model the entire beam, the second mesh
contains two elements, each 2500 mm long, and the third mesh contains three
elements, each 1667 mm long. The final displacements are 74\%, 93\%, and 96\%
of the exact displacement for the one, two, and three element meshes,
respectively.
Figure 5.10 Displacement history of cantilever beam.
5.6 15-Story Building Under Base Excitation

Figure 5.11 shows the finite element mesh of a 15-story building model previously studied by Kianoush and Scanlon (1986). The same model is adopted for this study with some modifications. The plane stress and connection elements are replaced by shell and beam elements, respectively. Since the explicit solver implemented in Snaps3D does not allow the material density to be specified as zero, the mass density for both the wall panels and the connection are assumed to be 2400 kg/m³. This distributed mass is in addition to the 372000 kg roof mass considered in the previous study. The remaining material properties are as follows:

Panels:
\[
\begin{align*}
t & = 200 \text{ mm} \\
E & = 25 \text{ GPa} \\
\nu & = 0.20
\end{align*}
\]

Connections:
\[
\begin{align*}
E & = 17.6 \text{ GPa} \\
\nu & = 0.20
\end{align*}
\]

Initially, the dead load of 3648 kN is distributed over corner nodes at the roof level, followed by a base acceleration. A time step size of 0.05 ms is used. Figure 5.12 shows the displacement history of one of the roof nodes.
Figure 5.11 Mesh of the 15-story building model (Kianoush and Scanlon 1986)
Figure 5.12 Displacement history of 15-story building model.
Because of the added mass, the response is slightly different than what was observed by Kianoush and Scanlon (1986), however, the maximum lateral displacement is approximately 15 mm in both cases.

5.7 U-Shaped Shear Wall Building

The final example provided in this study is of a U-shaped shear wall building with horizontal connections, shown in Figure 5.13. The plan and elevation views of the building are shown in Figure 5.14 and Figure 5.15, respectively. The panels, slabs, and beams are assigned an elastic type material with the following properties:

\[
\begin{align*}
    \rho &= 2400 \text{ kg/m}^3 \\
    E &= 33 \text{ GPa} \\
    \nu &= 0.15 \\
\end{align*}
\]

For the horizontal connections, both linear elastic and nonlinear cases were considered. In the linear elastic cases, the same material as the panels are assigned to the connections. In the nonlinear cases, a concrete type material with the following additional properties is used:

\[
\begin{align*}
    \varepsilon_{c}, f_{c} &= 0.002, -48 \text{ MPa} \\
    \varepsilon_{\text{control}}, f_{\text{control}} &= 0.0035, -31 \text{ MPa} \\
    \varepsilon_{f}, f_{f} &= 1, 33 \text{ GPa} \\
    \mu &= 0.5 \\
    G &= 14.34 \text{ GPa}
\end{align*}
\]
Figure 5.13 Isometric view of U-shaped shear wall building.
Figure 5.14 Plan view of U-shaped shear wall building.

Figure 5.15 Elevation of U-shaped shear wall building.
The response of the building is evaluated in two different loading conditions: A triangular pulse excitation applied at the base in x-direction, shown in Figure 5.16, and a more general seismic base excitation applied in both x and y directions, shown in Figure 5.17, concurrently. In both loading cases, the gravity load is applied to the structure within the first 100 ms under very heavy damping and then kept constant. The lateral excitations are started at 100 ms under a much smaller mass proportional damping coefficient. For the general seismic base excitation case, the first 900 ms of 1940 El Centro earthquake is used (N-S component in x-direction and E-W component in y-direction).

Figure 5.18 shows the displacement histories of the node located at coordinates (3800, 3800, 5700) when linear elastic and nonlinear cases. Since the west side of the building is much stiffer than the east side, the building tips in positive x direction slightly under gravity loads. This is also evident in Figure 5.19, where the elevation of the wall 1 is shown. Figure 5.20 shows the deformed configuration of wall 1 at the time of the peak roof displacement (t=120ms), and Figure 5.21 shows the final state of the wall. Figure 5.22 shows the axial and shear strain history of the integration point closest to the eastern face of wall 1. Since no other gravity loads other than the building self weight is considered, the axial strains at the base are very small under gravity loads. As a result, the eastern tip of walls 1 and 3 lift up during the application of the pulse as evidenced by the positive (tensile) axial strain in Figure 5.22. This also leads to some slip accumulation in the connection region since the current connection model offers no shear resistance when axial stress is positive (i.e. tension).
Figure 5.16 Loading case 1: Triangular pulse in x-direction.

Figure 5.17 Loading case 2: General seismic load in x and y-directions.
Figure 5.18 Displacement history of northeast tip of the roof under the pulse.

Figure 5.19 Deformed shape of Wall 1 at t=100 ms
Figure 5.20 Deformed shape of Wall 1 at t=120 ms

Figure 5.21 Deformed shape of Wall 1 at t=300 ms.
Figure 5.22 Axial vs. shear strain history of the east end of Wall 1 at base.
Figure 5.23 and Figure 5.24 show the displacement histories of the same node at coordinates (3800, 3800, 5700) under the general seismic loading in x and y directions, respectively. Similar to the first loading case, the primary difference between the linear elastic and nonlinear models are due to slip at the connection region in the nonlinear model. In x-direction, the slip accumulates primarily in one direction causing a shifted up response compared to linear elastic model. However, in the y-direction, the initial slip is later recovered and the response observed in linear elastic and nonlinear models are almost identical after 600 ms.

5.8 Summary

The examples provided in this chapter demonstrate the capabilities that are currently implemented in SNAPS3D. The first example is used to verify the validity of the computer implementation of the solid element and the solver classes. It also shows that the solver is able to handle concentrated load cases. The second example is used to verify the solver when the loading is specified in the form of base acceleration. The third example is used to demonstrate how the element and material models in Snaps3D can be used to model precast concrete components. While modeling of unbonded reinforcement requires an additional beam element with reinforcing bar fibers, it is possible to model bonded reinforcement by defining rebar fibers alongside the concrete fibers in the same section. The fifth example demonstrates the capability of Snaps3D to model precast wall panel structures under gravity and seismic loads. The final example
Figure 5.23 Displacement history of the roof under seismic load in x-direction.

Figure 5.24 Displacement history of the roof under seismic loading in y-direction.
shows the response of a precast building with unsymmetrical layout and horizontal connections under base accelerations.

The elements and materials used to model the shear wall can be used to model frame connections as well. Beam elements with gap-shear slip behavior can be used to model the grout pads while beam elements with elastic material properties model the precast beams and columns. Unbonded or bonded reinforcement can be modeled by using beam elements with rebar fibers or by defining rebar fibers in the same section as grout fibers, respectively. Because all the elements are continuum based, there is no added work of coming up with force-displacement or moment-curvature relations for lumped elements such as plastic hinges.
Chapter 6

Conclusions and Recommendations for Future Work

6.1 Conclusions

In this study, a set of finite element class libraries developed using the Microsoft .NET Framework and C# programming language for the purpose of nonlinear analysis of precast concrete structures has been presented. Current library of elements include eight-node solid, shell, and beam elements. The material library contains an elastic material, which can be used with any of the element type in the library. The “concrete” material, which combines a gap model for normal direction and shear-slip model for the transverse directions and the “rebar” material, which uses a bilinear isotropic or kinematic hardening model for normal direction and elastic model for shear are only applicable to beam type elements. It is shown that the modeling capabilities of SNAPS3D can be extended through interfaces or by inheriting from existing classes. In most cases the existing program will require very minor modifications and there is no risk of breaking the existing program.
6.1.1 Modeling Precast Components

Solid and shell elements with elastic materials can be used to model wall panels and slabs since these members primarily remain linear-elastic under seismic loads. Beam elements with elastic materials can be used to model beam and columns. Nonlinear behavior at the connections can be modeled using beam elements with fibers that have concrete, rebar, or both material properties. Unreinforced connections can be modeled using beam elements with concrete fibers and reinforced connections can be modeled using beam elements with concrete and rebar fibers. Unbonded reinforcement or post-tensioning can be defined by adding beam elements with only rebar fibers.

All the element and material models are suitable for three-dimensional analysis. Instead of isolating frames or shear walls from the rest of the structure, entire buildings can be tested under various loading conditions including nodal, body, and seismic loads. Base accelerations can be applied in three orthogonal directions.

6.1.2 Programming Approach

The program presented in this study is not a rewriting of an existing program using a different programming language. An entirely new finite element platform has been developed. Because of object-oriented approach utilized in the design of the program, the addition of a new element or material model does not require any modifications to the existing program. For example, as far as the solver object is concerned, an element is an object that implements the IElement interface: It doesn’t matter whether it is a solid or a shell, has 8 or 16 nodes.
Every object encapsulates all the data it requires to function: There are no connectivity or coordinate matrices. Instead, each element references the node objects that define its connectivity through its NodeArray property and each node can access the vector that defines its position in space through its Position property.

New element or material models can be added using one of two ways. The easiest way is by inheriting one of the existing classes. For example a solid element with 16-nodes can easily be derived from the existing solid element with eight nodes. The only methods that need to be overwritten are the ones that define shape functions and its derivatives. There is no need to rewrite strain-displacement relations, mass matrix or force calculation methods. Alternatively, the new element can be written from scratch provided that it implements the IElement interface. While the former requires the developer to have some knowledge on the inner workings of the existing class, since some methods are overwritten and some are left alone, the latter does not require any knowledge at all and will alert the developer of the properties and methods he/she should implement in order to create the class.

The top level class in SNAPS3D is the Domain class which stores all the model information. Instead of having specialized lists for storing nodes, elements, etc., the Domain class has a single list accessed through its MemberList property. The list contains all the objects in the domain, whether they are of node, element, or section type. This allows the possibility of adding new types that are very different than the existing ones without having to rewrite
the Domain class and thus keeping compatibility with older input files. The Solver object creates the specialized arrays automatically when the time-history analysis starts using a .NET feature called “reflection”. This means that a class such as superelement/substructure which does not fit the current SNAPS3D type categories can be added to a domain’s memberlist. However, a new solver class which knows what to do with this type of an object needs to be developed as well.

The use of .NET Framework and C# programming language not only allow type checking by using reflection during runtime, but also relieve developers from having to manage memory as is the case with C++. Furthermore, any new feature does not have to be coded in a specific programming language. Referring back to the previous example on extending the element class library, while the code for the eight-node solid element is written in C#, the code for the 16-node solid element may be written in a different .NET compliant language.

6.2 Recommendations for Future Work

The primary contribution of this study is the development of the basic program framework for analysis of precast concrete structures. The future work related to extending SNAPS3D can be divided into the following categories:

1. Additional solvers: Because of the explicit integration scheme adopted, the current solver requires very small time steps to be used. This particularly becomes a problem at the precast frame/wall connections where the element dimensions are relatively small. The development of implicit solvers can help alleviate this problem.
2. Improved user interface: The current user interface is provided to facilitate basic input/output. Three-dimensional models have complex geometries and a preprocessor which allows the model geometry to be viewed would be very beneficial in reducing the errors in input.

3. Additional material models: Additional material models are needed to represent various vertical connection behavior (shear keys, shear-friction, etc.), post-tensioning tendons (nonlinear-elastic), and the connections between slab panels.

4. Subdomains: The domain can be divided into subdomains which are connected to each other through common nodes. The meshing of the subdomain can be done internally. Furthermore, using .NET Remoting, the internal force calculations of each subdomain can be performed at a different processor speeding up the execution time.

The future work related to analytical modeling of precast concrete structures can be divided into the following categories:

1. Conduct parametric studies to evaluate the effects of diaphragm stiffness, shear-wall configurations, and frame systems on response of precast concrete buildings.

2. Study the effects of irregularities in layout of lateral load resisting system on seismic response.
REFERENCES


ACI-ASCE Committee 550, and American Concrete Institute. (2002). *Emulating Cast-in-Place Detailing in Precast Concrete Structures (ACI 550.1R-01)*, American Concrete Institute ;

American Society of Civil Engineers, Detroit, Mich.

New York, N.Y.


Powell, G. H. (1993). "DRAIN-2DX Element Description and user Guide for Element Type01, Type02, Type04, Type06, Type09, and Type15." *UCB/SEMM-93/18*, University of California, Berkeley, CA.

Precast/Prestressed Concrete Institute. (1999). *PCI design handbook : precast and prestressed concrete*, Precast/Prestressed Concrete Institute, Chicago.


Appendix A

BCL Class Members
### Table A.1 System.Object class members

<table>
<thead>
<tr>
<th>Public Constructors</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object Constructor</strong></td>
<td>Initializes a new instance of the Object class.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Public Methods</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Equals</strong></td>
<td>Overloaded. Determines whether two Object instances are equal.</td>
</tr>
<tr>
<td><strong>GetHashCode</strong></td>
<td>Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.</td>
</tr>
<tr>
<td><strong>GetType</strong></td>
<td>Gets the Type of the current instance.</td>
</tr>
<tr>
<td><strong>ReferenceEquals</strong></td>
<td>Determines whether the specified Object instances are the same instance</td>
</tr>
<tr>
<td><strong>ToString</strong></td>
<td>Returns a String that represents the current Object.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Protected Methods</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Finalize</strong></td>
<td>Overridden. Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.</td>
</tr>
</tbody>
</table>
### Table A.2 System.MarshalByRefObject class members

<table>
<thead>
<tr>
<th>Public Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CreateObjRef</strong></td>
<td>Creates an object that contains all the relevant information required to generate a proxy used to communicate with a remote object.</td>
</tr>
<tr>
<td><strong>GetLifetimeService</strong></td>
<td>Retrieves the current lifetime service object that controls the lifetime policy for this instance.</td>
</tr>
<tr>
<td><strong>InitializeLifetimeService</strong></td>
<td>Obtains a lifetime service object to control the lifetime policy for this instance.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Protected Constructors</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MarshalByRefObject Constructor</strong></td>
<td>Initializes a new instance of the MarshalByRefObject class.</td>
</tr>
</tbody>
</table>
Appendix B

Input Example

Figure B.1 Input screen for L-Shaped beam analyzed in Chapter 5.
Vita

Serdar Astarlioglu; born in 1970, in Bursa, Turkey

Education:
- Ph.D./ Civil Engineering, The Pennsylvania State University
- B.S./Civil Engineering, Istanbul Technical University, 1992.

Experience:
- Developed structural engineering software:
  - Development of an object-oriented nonlinear dynamic finite element program in C#.
  - Development and validation of computer code for fully nonlinear analysis of structural systems using advanced single-degree-of-freedom approach under various explosive environments (both above-ground and underground structures) in Fortran 90.
  - Development and validation of Timoshenko beam code for more advanced fully nonlinear simulations in Fortran 90.
- Designed building structures:
  - City Center, Residential - North Tower, White Plains, New York. Designed lateral and gravity load resisting systems and foundation mat for a 37 story high-rise building.
  - City Center, Retail, White Plains, New York. Designed steel plate girders and reinforced concrete transfer girders, lateral load resisting system.
  - Signature Place, Stamford, Connecticut. Designed lateral and gravity load resisting system and caisson foundations for a 38-story residential high-rise complex.
  - Promenade at Port Imperial North, Building A, West New York, New Jersey. Designed lateral and gravity load resisting system, post-tensioned concrete slabs for a 10-story mixed-use facility.
- Taught courses:
  - CE 340 “Structural Analysis” with ~ 80 students.
  - CE 341 “Design of Reinforced Concrete Structures” with ~ 60 students.
  - CE 240 “Structural Analysis” with ~ 80 students.

Background Information:
- Member, American Society of Civil Engineers.
- Working knowledge of English and Turkish.
- Experienced in Quick Basic, Fortran 90, and C# programming languages.