InterPSS

Development Tutorial

In progress...

Version: <V0.42>
Date: <7/23/2014>

Document Version History

Version No.	Release Date	Description	Prepared by
V0.1	12/17/2013	start working on the first version, the outline of tutorial	T.Huang
V0.2	1/25/2014	completed ch2: intro to the three basic power system models:bus, branch and network	T.Huang
V0.3	03/02/2014	add the Appendix-B, intro to ODM	T.Huang
V0.4	03/16/2014	add ch4 short circuit, update ch5 dstab	T.Huang
V0.41	04/12/2014	add ch5 dstab monitoring and output	T.Huang
V0.42	07/23/2014	add Appendix-A sparse matrix and solver	T.Huang

Table of Contents

Introduction

This development tutorial is intended to help users of InterPSS to learn InterPSS and how to use it better and faster. Background knowledge of basic Java programming and some knowledge of power system modeling and simulation is required. For those whose are not that familiar with Java programming and Eclipse IDE, It is suggested that you spend one or two weeks on those stuff first and then come back to this tutorial. There are many on-line learning material for you when you google “Java tutorial” or “Eclipse tutorial”. For example:

- Java learning: http://www.learnjavaonline.org/
- Eclipse:http://eclipsetutorial.sourceforge.net/totalbeginner.html

If you have some background in Java and interested to learn InterPSS, then it is right for you. This tutorial will first guide you through setting up the development environment in Chapter 1. In Chapter 2, introduction to the basic power system models in InterPSS is presented. In the following Chapter 3 to 5, the modeling and simulation algorithm in InterPSS for typical applications are discussed , including power flow, short circuit and transient stability. In Chapter 6, integration of InterPSS with the GAMS (a high-level modeling system for mathematical programming and optimization) will be presented, along with a simple SCED example. For each chapter, simple application examples is created to help users to have better understanding of the material in the chapter. You are encouraged to create your own study case by following those examples.

- Tutorial example and code

This tutorial is accompanied by simple examples for each chapter. The code for all examples is available on GitHub: https://github.com/InterPSS-Project/ipss-common/tree/master/ipss.tutorial .

Chapter 1. Setting up the InterPSS Development Environment

InterPSS project has been trying to create a new open platform, to support the next generation power system simulation and analysis. The foundation of this platform is InterPSS core engine, which includes but not limited to power system network object model and power system analysis functions, e.g., power flow, short circuit and transient stability.

InterPSS now supports two versions, i.e., cloud edition and development edition. Both editions run on the same core simulation engine. InterPSS cloud edition runs on the Google cloud platform and provides power system analysis as a service, 24/7, anywhere in the world with the Internet access. For more information, please visit www.interpss.org . The cloud edition provides basic analysis functions and it is simple to use. However, flexibility and capability of InterPSS core engine is still not fully exploited. For those who want to have a better control of InterPSS, or customize it for their applications, InterPSS development edition is recommended. The development edition is based on the Eclipse IDE and InterPSS core libraries.

Those familiar with the Java programming knows that basic and core java functions are provided in Java system libraries. Each java projects should have a reference to these JRE System library. Similarly, for power system simulation and analysis with InterPSS, InterPSS core libraries and their dependent third party libraries are required.

Prerequisite of InterPSS development environment is as follows:

- **Java Environment Setup**

Make sure you have Java SDK installed on your computer. You can check your Java installation by launching a CMD window from Start/run/cmd. Type the following command

```
java -version
```

to make sure that your have Java version is 1.7.xx or above.

- **Eclipse IDE Setup**

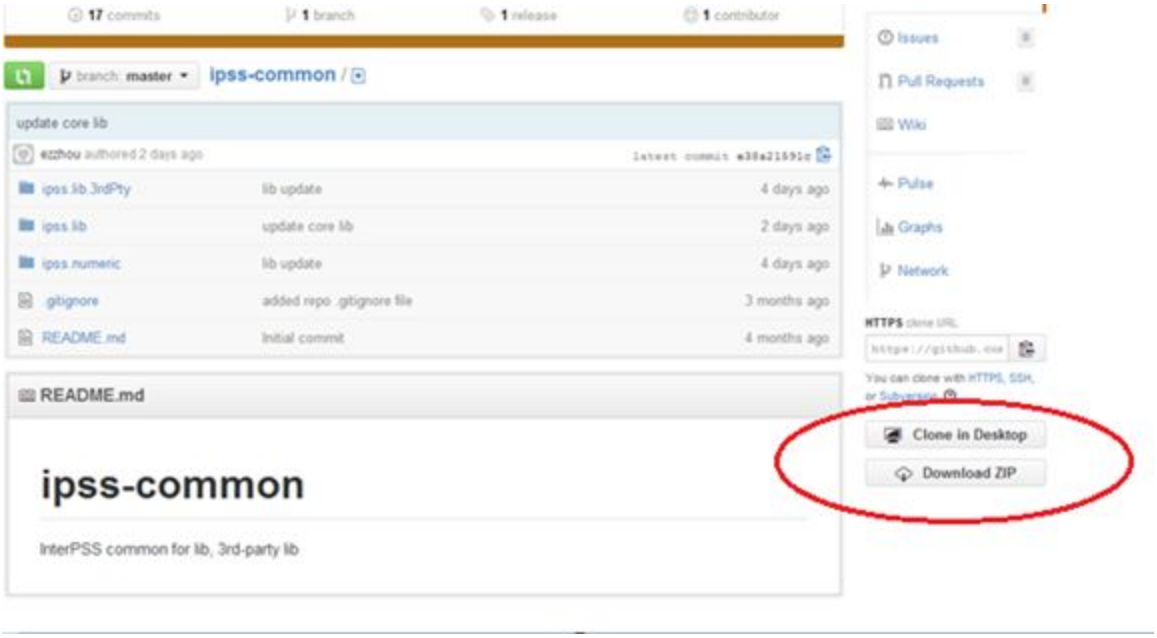
InterPSS is written in Java. InterPSS development team uses Eclipse as the main Java IDE. Currently, InterPSS uses Eclipse Kepler (4.3.1), which can be downloaded from www.eclipse.org . You can use the **"Eclipse IDE for Java Developers"** version.

1.1 Import InterPSS library projects

InterPSS core libraries and all dependent libraries (*.Jar) are stored in the following GitHub repository https://github.com/InterPSS-Project/ipss-common

The library projects can be downloaded fully as a ZIP file (about 60 MB), or clone to the desktop through

Git clone as indicated by the two operation buttons on the right bottom.

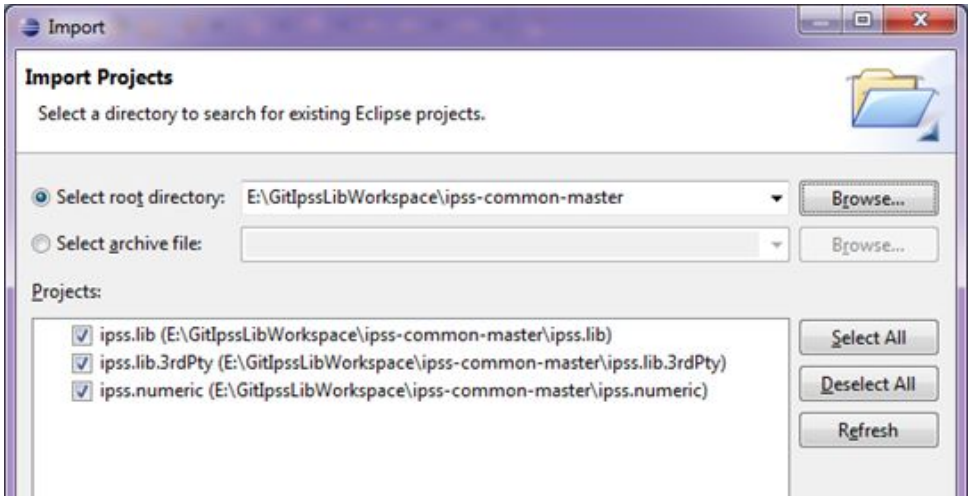


Some test cases are also available within the ipss.sample project stored in the ipss-plugin repository https://github.com/InterPSS-Project/ipss-plugin

In the following two subsections, two approaches of importing the InterPSS core lib projects and ipss.sample project into your working workspace of Eclipse will be introduced, and they are 1) downloading the whole repository Zip file and 2) clone the repositories on Github via Eclipse EGit plugin.

1.1.1 Library repository as a zip file

1) Click on the **Download Zip** button to download a zip file containing all the three lib projects
2) Extract (unzip) the projects from the downloaded zip file
3) Import->General-> Existing projects into workspace



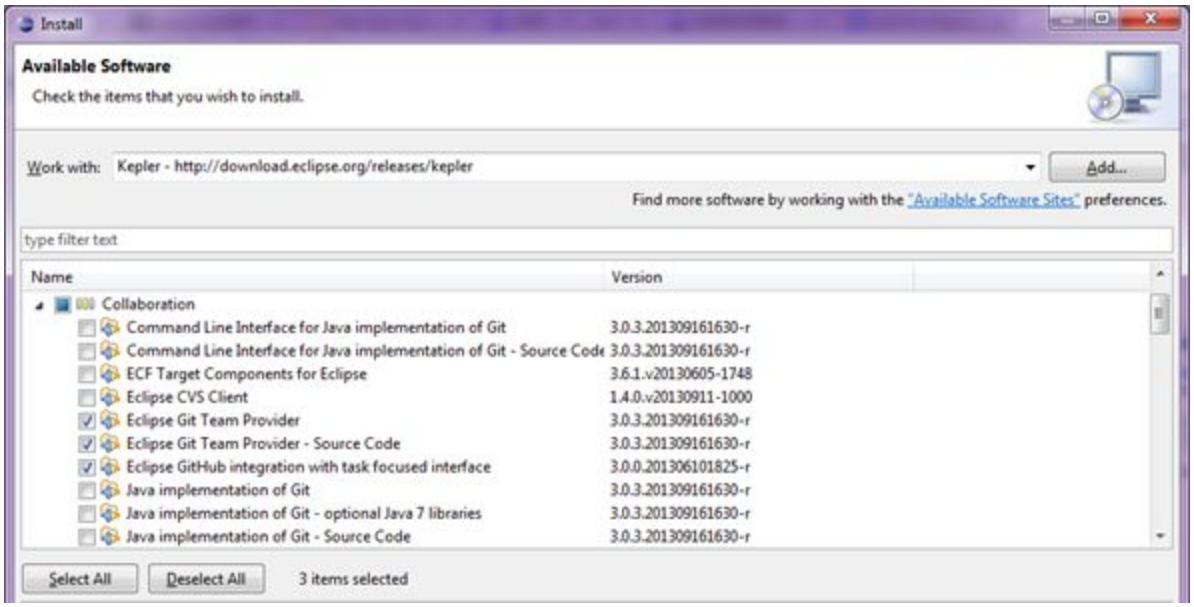
4) Following the same process as before to download the **ipss.sample** project from https://github.com/InterPSS-Project/ipss-plugin and run the sample cases.

1.1.2 Clone Library project repository via Egit

(1) GitHub Eclipse plugin

The EGit plugin is available from http://eclipse.org/egit/download/ or Eclipse->Help->Install new software

1) Select or add: Work with Kepler - http://download.eclipse.org/releases/kepler
2) Choose Eclipse Git Team related installations as shown in figure below

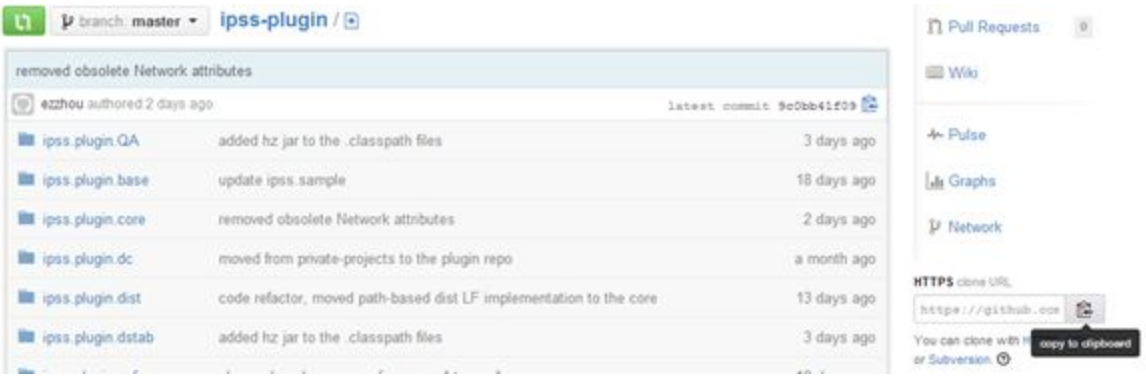


3) Install and restart

For the use of EGit, the following user guide is recommended
http://wiki.eclipse.org/EGit/User_Guide

(2) Clone the repository to local PC and import the projects into workspace

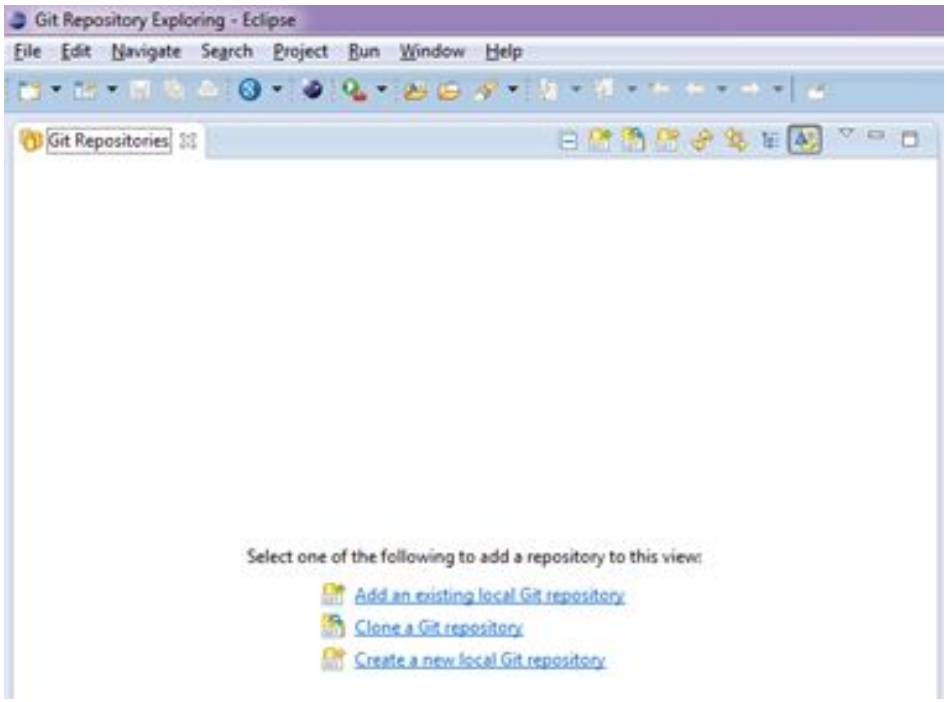
1) Get the project HTTPS clone URL by copying the URL to clipboard



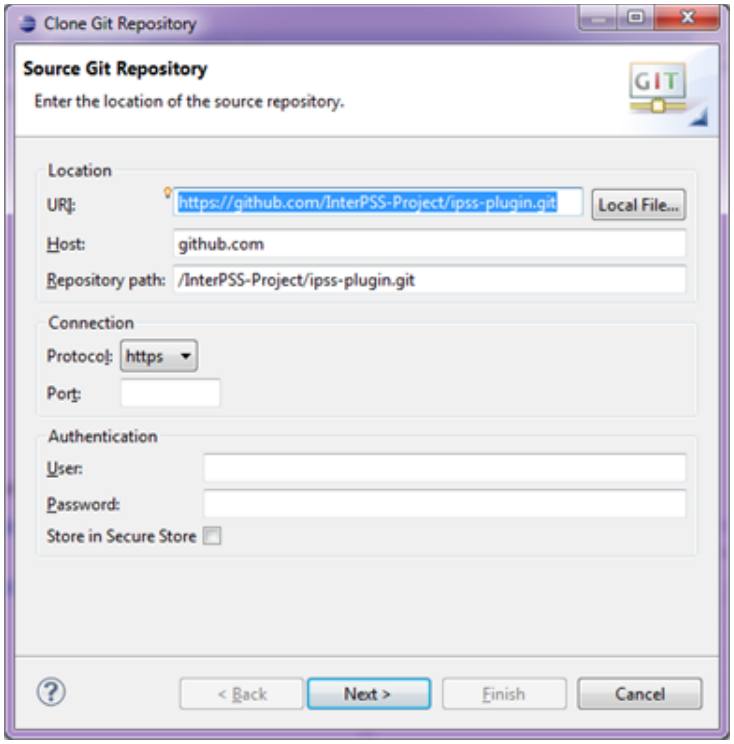
· InterPSS lib projects:
https://github.com/InterPSS-Project/ipss-common.git
· InterPSS plugin including the ipss.sample project:
https://github.com/InterPSS-Project/ipss-plugin.git

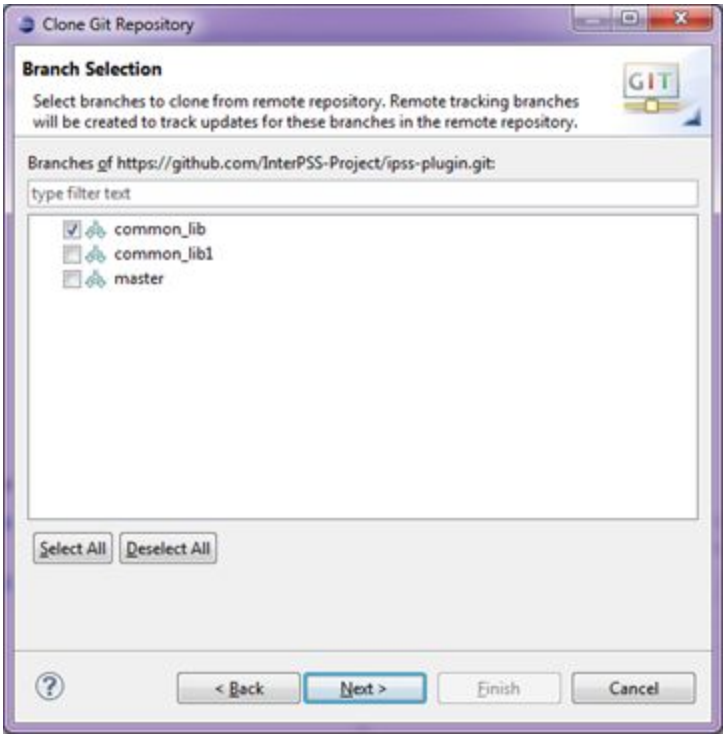
2) Open the Git repositories perspective through Eclipse->Window-> Open perspective-> Others-> Git Repository Exploring

3) Select “Clone a Git repository” from the Git repository perspective

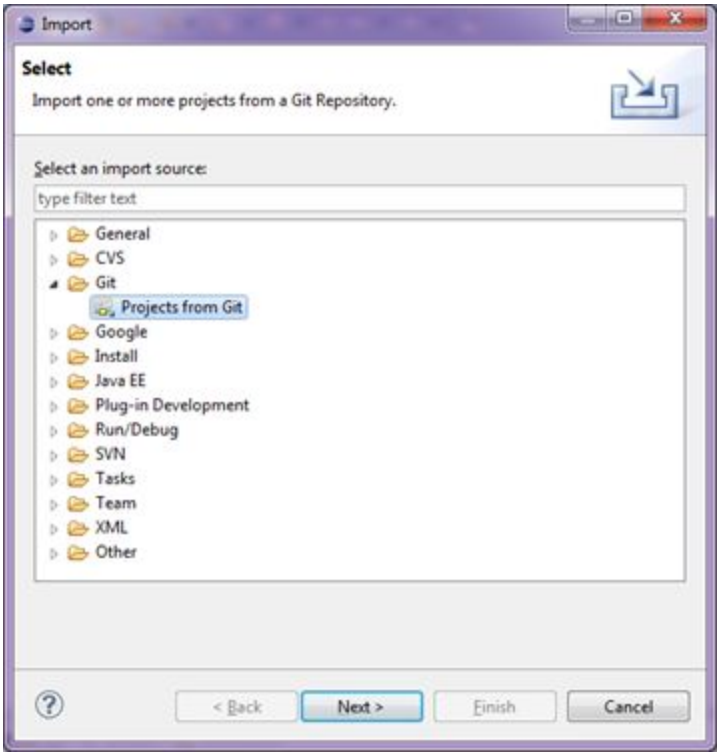


Clone the ipss-common and the ipss-plugin repositories to your local storage. Steps for cloning the ipss-plugin repository will be shown below, and the process is the same for ipss-common repository.

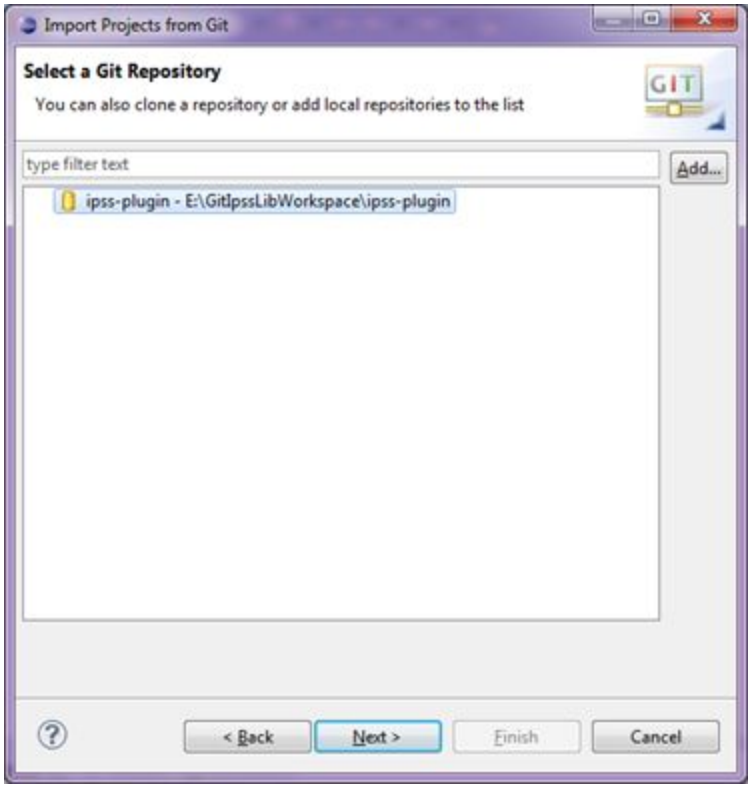


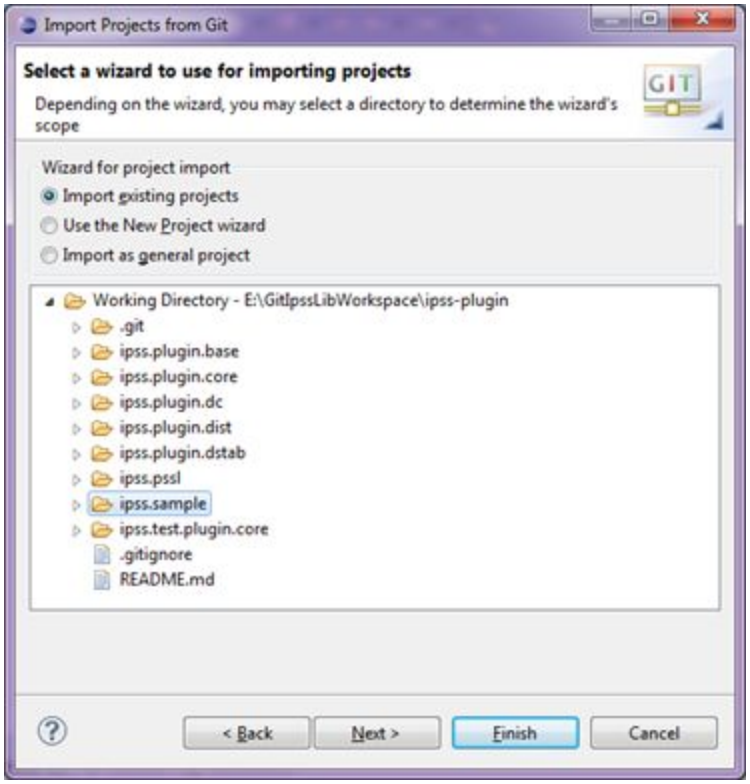


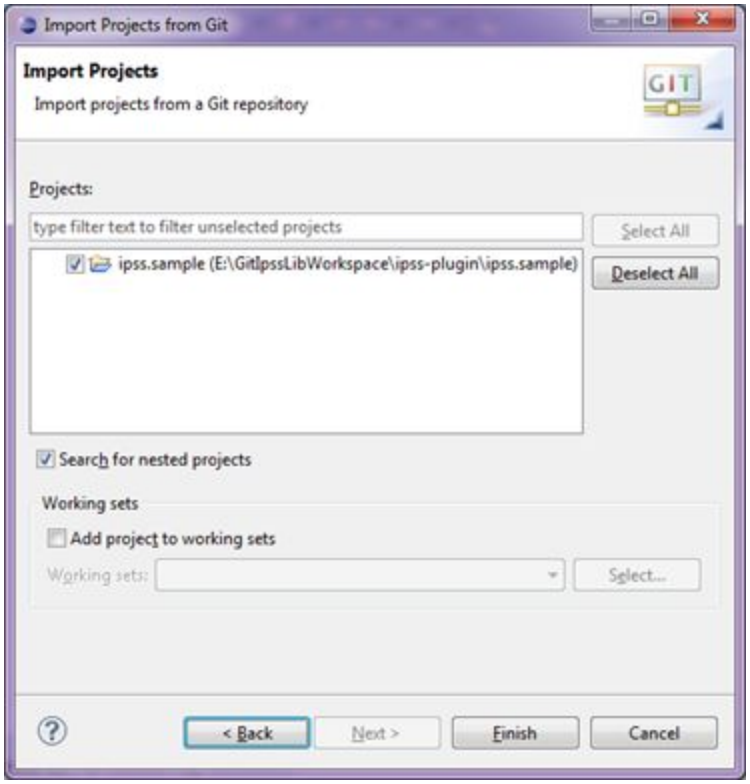
Note: Always choose the "**common_lib**" which means the project build path are linking to the common library Jars.

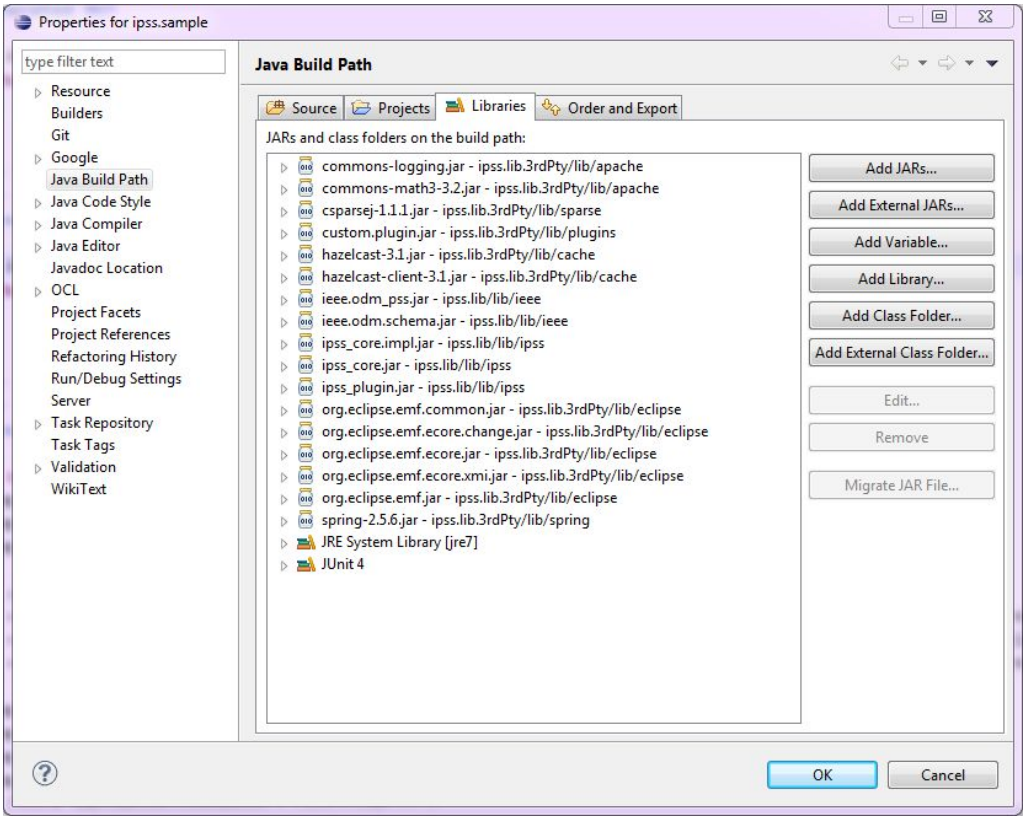


Choose Local and then select Ipss-plugin in the next step




Import Projects from Git
Select a wizard to use for importing projects
Depending on the wizard, you may select a directory to determine the wizard's scope
GIT
Wizard for project import
Import existing projects
Use the New Project wizard
Import as general project
Working Directory - E:\GitIpssLibWorkspace\ipss-plugin
.git
ipss.plugin.base
ipss.plugin.core
ipss.plugin.dc
ipss.plugin.dist
ipss.plugin.dstab
ipss.pssl
ipss.sample
ipss.test.plugin.core
.gitignore
README.md
< Back
Next >
Finish
Cancel

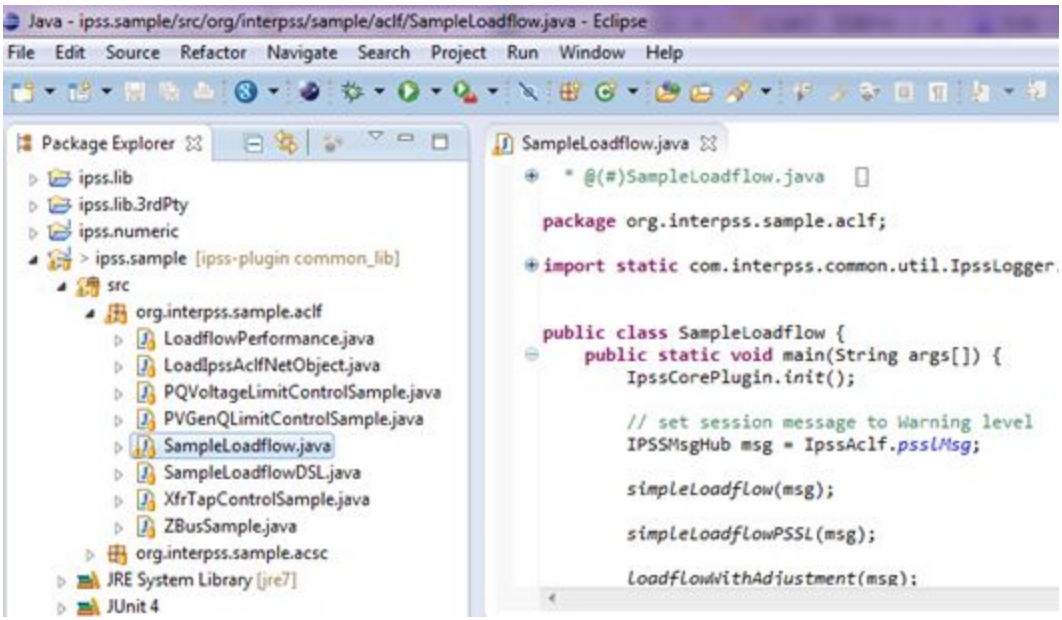

Import Projects from Git
Import Projects
Import projects from a Git repository
GIT
Projects:
type filter text to filter unselected projects
Select All
Deselect All
ipss.sample (E:\GitIpssLibWorkspace\ipss-plugin\ipss.sample)
Search for nested projects
Working sets
Add project to working sets
Working sets:
Select...
< Back
Next >
Finish
Cancel



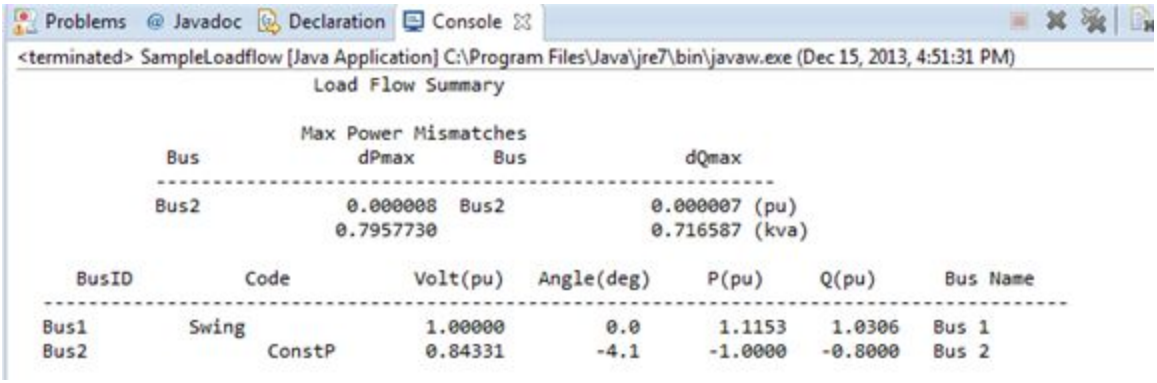
Check the build path of **ipss.sample** to make sure all the required libs are loaded and there is no error in the [Problems] view window.

1. 2 Run sample load flow test case

Right-click the **SampleLoadflow.java** and Run As "Java Application"



Two bus load flow result is as follows:



Chapter 2. An introduction to the power system basic models in InterPSS

As Java language is the programming language adopted by InterPSS, the object oriented programming (OOP) technique is widely used in the design and development of InterPSS. As is known to all, one of the most basic concept in OOP is the object. To understand InterPSS, and subsequently extend InterPSS for your own purpose, it is important to be familiar with the three basic types of power system models within the core of InterPSS, and they are network, bus and branch. For more information of OOP and InterPSS, please read this IEEE Transaction paper "Object-oriented Programming, C++ and Power System Simulation", IEEE Transactions on Power Systems, Vol.11, No.1, pp. 206-215, 1996

2.1. Overview of power system modeling in InterPSS

A network is to store all the information required for power system simulation in InterPSS. At the center of the network object is two linked lists, which are used to store the bus objects and branch objects, respectively, as shown in Fig.2.1.

InterPSS follows the bus-oriented convention to represent the network. Accordingly, Bus is defined to store all information (defined as attributes of the bus class) related to a bus, for example, bus Id and bus voltage, as well as all components connected to a bus, e.g., a generator or a capacitor. The other basic model is branch. It is used to represent overhead line, transformer (conventional two-winding transformer, phase shift transformer). Special type branches, currently including three-winding transformer and HVDC transmission lines, and series FACTs devices in the future, are stored in another list named **specialBranchList**. Within InterPSS, a three-winding transformer is automatically converted to three two-winding transformers.

In Fig.2.1, The bus and branch mutual reference help to define the topology of the network as a graph: the fromBusId and toBusId of a branch are the two attributes to relate two buses to a two-terminal branch, while the branchList of a bus store the branch objects connected to the bus at the “from-end” and “to-end” of the branch, respectively.

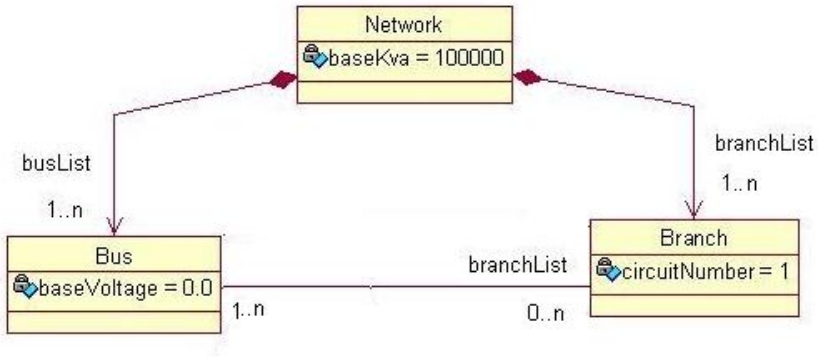


Fig. 2.1 InterPSS network model

Let's first look at the two-bus system example defined in the ***sampleLoadflow.java*** within the **ipss.sample** project under ipss.plugin, as this example give you some idea of these three types of objects and they together forms a power system network object for load flow analysis.

```
public static void simpleLoadflow(IPSSMsgHub msg) {
        // Create an AclfNetwork object
        AclfNetwork net = CoreObjectFactory.createAclfNetwork();

        double baseKva = 100000.0;

        // set system basekva for loadflow calculation
        net.setBaseKva(baseKva);

        // create a AclfBus object
        AclfBus bus1 = CoreObjectFactory.createAclfBus("Bus1", net);

                  // set bus name and description attributes
        bus1.setAttributes("Bus 1", "");
        // set bus base voltage
        bus1.setBaseVoltage(4000.0);

        // set bus to be a swing bus
        bus1.setGenCode(AclfGenCode.SWING);

        // adapt the bus object to a swing bus object
        AclfSwingBus swingBus = bus1.toSwingBus();

        // set swing bus attributes
        swingBus.setVoltMag(1.0, UnitType.PU);
        swingBus.setVoltAng(0.0, UnitType.Deg);


        AclfBus bus2 = CoreObjectFactory.createAclfBus("Bus2", net);
        bus2.setAttributes("Bus 2", "");
        bus2.setBaseVoltage(4000.0);

        // set the bus to a non-generator bus
        bus2.setGenCode(AclfGenCode.NON_GEN);

        // set the bus to a constant power load bus
        bus2.setLoadCode(AclfLoadCode.CONST_P);

        // adapt the bus object to a Load bus object
        AclfLoadBus loadBus = bus2.toLoadBus();
        // set load of the bus: Load = P + jQ = 1.0 + j*0.8 pu
        loadBus.setLoad(new Complex(1.0, 0.8), UnitType.PU);


        // create an AclfBranch object
        AclfBranch branch = CoreObjectFactory.createAclfBranch();
```

```
            net.addBranch(branch, "Bus1", "Bus2");

            // set branch name, description and circuit number
            branch.setAttributes("Branch 1", "", "1");

            // set branch to a Line branch
            branch.setBranchCode(AclfBranchCode.LINE);
            // adapte the branch object to a line branch object
            AclfLine lineBranch = branch.toLine();

            // set branch parameters
            lineBranch.setZ(new Complex(0.05, 0.1), UnitType.PU, 4000.0);

            // create the default loadflow algorithm
            LoadflowAlgorithm algo = CoreObjectFactory.createLoadflowAlgorithm(net);

            // use the loadflow algorithm to perform loadflow calculation
            algo.loadflow();

            // output loadflow calculation results
            System.out.println(AclfOutFunc.loadFlowSummary(net));
    }
```

There are several key points for the example above.

- Network object can be regarded as a container, which stores all buses and branches of a power system in InterPSS. Thus, it should be created at the very beginning.
- Bus are created latter and then branch. Such sequence is important as a branch needs to refer to the bus objects connected at the terminals in order to properly set up the bus-branch connection relations, hence the bus objects must be created before a branch tries to refer or connect to it.
- Both Bus and Branch objects are accessed and referred to by their IDs, for example,

  ```
  AclfBus bus1 = CoreObjectFactory.createAclfBus("Bus1", net);
  net.addBranch(branch, "Bus1", "Bus2").
  ```

  Where “Bus1” and “Bus2” are the IDs of bus1 and bus2, for identification purpose.

- Object attributes and/or information is set to the object through the **setter** methods (network/bus/branch.setX(value))

NOTE: *This example may give some of you a wrong impression that it needs to write thousands of lines of code to create a network for large power systems. Actually There is another open project called* ***Open Model for Exchanging Power System Simulation Data*** *that InterPSS has been devoting to. InterPSS has developed ODM-based adapters and mapper to import available network data defined in commonly used data format, such IEEE CDF, PSS/E, PowerWorld, PSD-BPA format to create a network object in InterPSS, For more information, please refer to Appendix B--Introduction to ODM.*

2.2. Inheritance and class hierarchy structure

- **Why inheritance?**

Inheritance is an important concept for objects in OOP. A sub-class, or “child” class, can be derived from the base, or “parent”, class through inheritance, such that the child class will inherit all the attributes and method defined the base class. With inheritance, we don’t have to rewrite all the common or similar parts of code again and again. That is commonly known as “code reuse” in computer science.

- **How is the class inheritance designed in InterPSS**

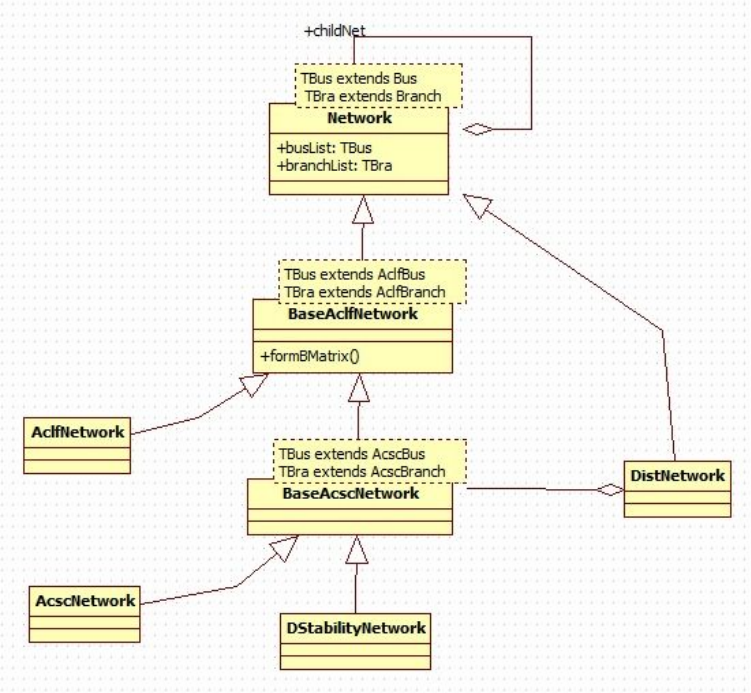
Based on the nature of power system analysis, we know that the most basic information for representing a power system is the network topology, as well as basic bus and branch identification attributes. The information is required for all power system analysis. Therefore, it is reasonable to abstract these basic, common information and behavior into base network, bus and branch classes.

Secondly, let’s look at the characteristics of the three basic power system analysis functions, i.e., load flow, short circuit calculation or fault analysis, transient stability simulation. Load flow is the most basic function, and both short circuit analysis and transient stability simulation rely on load flow data and results. Thus, load flow related objects should be on the second level of the hierarchy structure, from which the short circuit and transient stability related objects are derived.

Short circuit (SC) analysis usually requires load flow results to determine the operating point. Further, positive sequence network data for short circuit analysis is generally the same as that used in load flow study. Short circuit related objects can and should reuse load flow data and methods by inheriting the load flow objects. Moreover, the negative and zero sequence data, if available, will be included in the bus and branch objects for short circuit analysis.

Dynamic stability (DStab) simulation relies on the load flow data for constructing the network admittance as well as the equation YV=I and load flow results for determining the pre-fault system condition. During the fault period, also it needs to form the sequence network and calculate the positive sequence equivalent fault impedance at the fault point for unsymmetrical faults. Considering that such function is available in short circuit analysis. Thus both load flow and short circuit analysis should be reused for dynamic stability. For the bus objects for stability analysis (DStabBus), detailed generator (machine) model, with the controllers (exciter, governor, stabilizer) for the generator being a part of the generator model, will be included as a component in the bus object. Branch class for stability analysis (DstabBranch) extends from the branch class for short circuit and adds attributes and methods for relay and protection setting.

The network class structure in InterPSS is shown as follows:



Note for the naming in the figure above:

Aclf -- AC load flow
Acsc -- AC short circuit
Dstab -- Dynamic stability or Transient stability
Dist -- Distribution

The bus and branch class have a similar hierarchy structure, as follows:

- Bus<--AclfBus<--AcscBus<--DStabBus
- Branch<--AclfBranch<--AcscBranch<--DstabBranch

where the "<--" indicates that the latter extends the former.

InterPSS employs the "separating interface from implementation class" concept internally when implementing the the core network objects mentioned before. For example, a **bus** is first defined as an interface, which defines all the public operations or APIs accessible from outside, then it is implemented in a separate class, i.e., **BusImpl**.

2.3 Three basic types of power system models

Network, bus and branch are the three basic types of power system objects in InterPSS, other commonly-used models for simulations are generator, load, HVDC, area, zone, machines. In the following, only the three basic types will be discussed.

2.3.1 Network object

Network is the container for all the information required for power system analysis in InterPSS. As each type of power system analysis requires different information from network, bus and branch, it is reasonable to define a different network object for each type. In addition, it should be noted that the network topology information is the most basic for all analyses, thus a basic network object is employed in InterPSS. Further, there is some information exchange among these analysis. In the past, such data exchange is done by file input and output. Now the same process is perfectly solved through object inheritance, as discussed in Section 2.2. Considering that there are mainly three typical types of applications, namely, load flow, short circuit and transient stability, AclfNetwork, AcscNetwork and DStabNetwork are defined accordingly.

- **Basic Network**

The **Network** class and the corresponding implementation **NetworkImpl** class is to form the network topology with busList, the connection relationship among buses through branches.

- **Load flow Network**

The major difference between **Network** and **AclfNetwork** is that all the buses and branches objects contained in **AclfNetwork** object must be of type **AclfBus/AclfBranch** class or its sub-class, which means they contain all the information required for load flow analysis.

Also, the admittance matrix, or Ymatrix, and Jacobian matrix,as the basic underlying data structure for load flow analysis, are formed in the **AclfNetwork**.

- **Short Circuit Network**

BaseAcscNetwork and **AcscNetwork** class is for short circuit analysis. **BaseAcscNetwork** is extended from the **BaseAclfNework**, the major difference is that sequence network admittance matrix can be formed and accessed in **AcscNetwork,** besides all the **AclfNetwork** operations. Buses and branches in the AcscNetwork are of **AcscBus** and **AcscBranch** type. The sequence network data, including sequence impedance of generators, equivalent impedance or admittance of load and sequence impedance of branches, should be provided.

- **Transient stability Network**

DStabilityNetwork class is for transient stability analysis. all the buses and branches objects contained in this object must be of type **DStabBus/DStabBranch**. The main functions of this class include network initialization and solving network dynamic equation during simulation.

2.3.2 Bus

Recall the class structure**,** Bus<--AclfBus<--AcscBus<--DStabBus. **Bus** interface (and **BusImpl** class) is the basic class for a bus, and busId, baseVoltage.

The bus model for load flow, short circuit and transient stability is shown in Fig.2.3. A bus object contains a **genList** and **loadList** to store the generation/load information, which means *multiple generators and loads are allowed to connect to the same bus, with different ID for each generator/load*. In addition, there is at most one fixed shunt and one switched shunt. A bus is connected to other buses through line or transformer. The **BranchList** of a bus stores all the branches connected to it.

For short circuit analysis, sequence network data for generator and load connected to the bus is required. For transient stability, detailed dynamic model data of the generators, if any, is required, dynamic load model data will also be necessary if dynamics of the loads are to be included in the simulation.

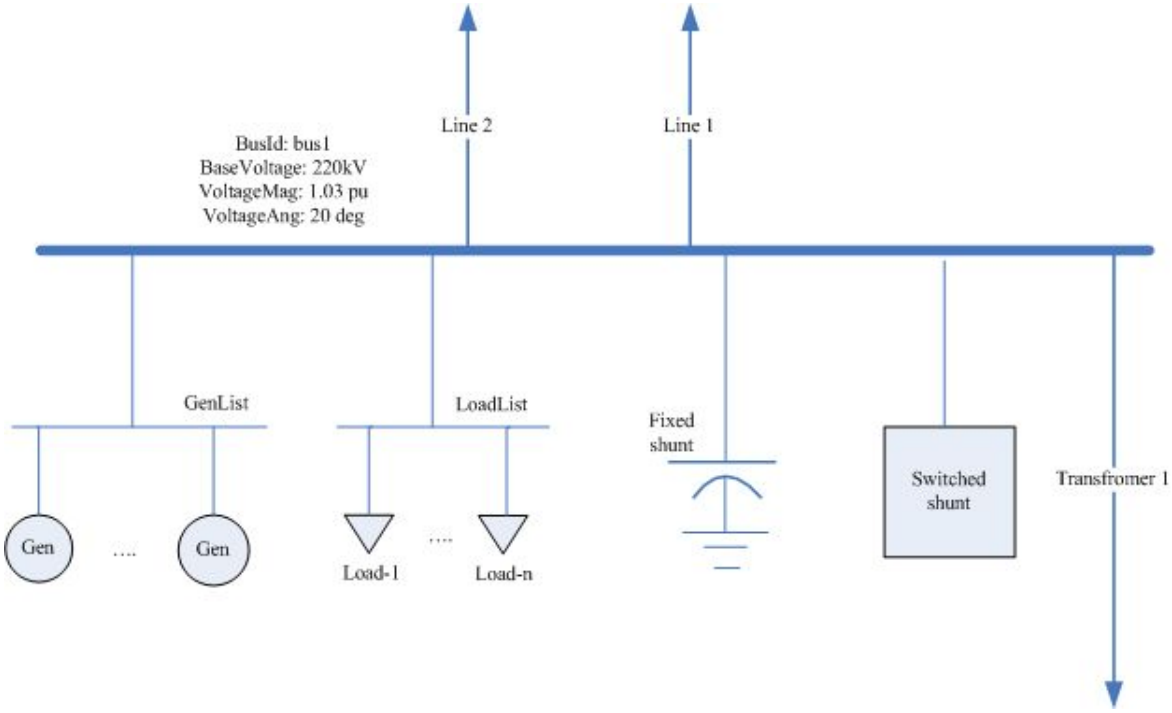


Fig.2.3 Bus model in InterPSS

2.3.3 Branch

Overhead line and underground cable, transformers are treated as an abstract **Branch** in a network, which connects two or more buses. So **Branch** is the basic class for all these connection type components in the power systems. The basic attributes for a branch are the fromBus, toBus (tertBus if it is a three-winding transformer).

Overhead lines and underground cables are modeled by the **Line class** in InterPSS. Transformers in power systems include two-winding transformer and three-winding transformer, and they are modeled separately. Internally one three-winding transformer is modeled by three two-winding transformer through Y-Delta transformation.

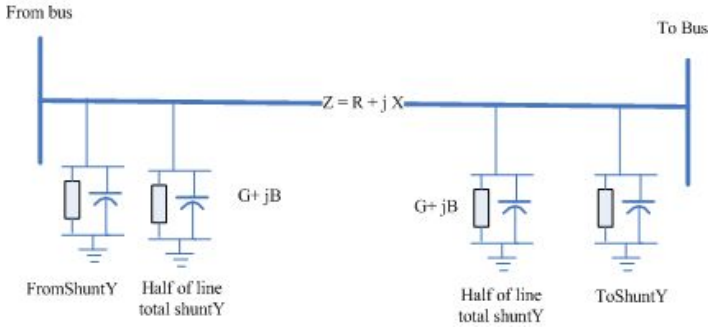


Fig 2.4 Line model in InterPSS

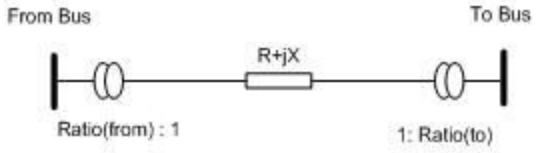


Fig 2.5 Two winding transformer model in InterPSS

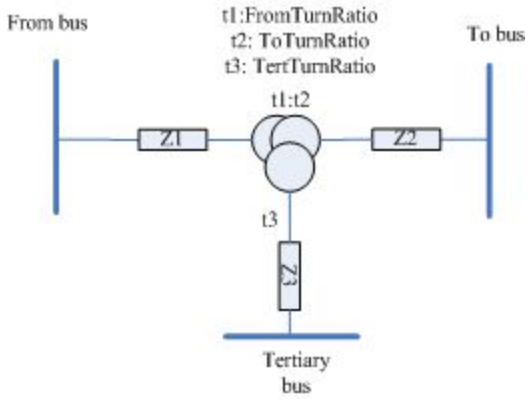


Fig 2.6 Three-winding transformer model in InterPSS

- **Line**

```
AclfBranch<--AcscBranch<-- DStabBranch
   |              |              |
   |              |              |
AclfLine  <-- - AcscLine  <--- DStabLine
```

The objects on the second row are object adapters of the corresponding object on the first row. The first row is the general. For more info about such implementation, please refer to the adapter pattern.

In the example below, there is such a line related to this adapter pattern:

```
// adapt the 'abstract' branch object to a 'specific' line branch object
      AclfLine lineBranch = branch.toLine();
```

- **Transformer**
  - two winding

```
AclfBranch<--AcscBranch<-- DStabBranch
   |              |              |
   |              |              |
AclfXformer  <-- - AcscXformer   <--- DStabXformer
```

  - three winding

```
Aclf3WBranch<--Acsc3WBranch<--DStab3WBranch
```

2.4. Getting and setting the data of an object

2.4.1 Network

- basic info: baseKva

net.setBaseKva(baseKva);	set network base KVA, unit in kVA
net.getBaseKva();	get network base KVA, unit in kVA

- Bus related

getNoBus()	get the total number of buses
getNoActiveBus()	get the total number of active buses
getBus(String busId)	get a specific bus by its id
addBus(AclfBus bus)	add a new bus to the network
getBusList()	Return a list of bus objects

- Branch related

getNoBranch()	get the total number of branches
getNoActiveBranch()	get the total number of active branches, i.e., those out-of-service branches are not included.
getBranch(String branchId)	get a specific branch by its id
addBranch(AclfBranch branch)	add a new branch to the network
getBranchList()	Return a list of branch objects which have two terminals

NOTE: *accessing a bus/branch object from a network object is used quite often, and the method via the bus/branch Id is recommended to use.*

2.4.2 Bus

- Get and set bus voltages

getVoltage() getVoltage(UnitType) setVoltage(Complex volt) setVoltage(Complex volt, UnitType type)	get and set voltage as a complex, voltage unitType can also be used

getVoltageMag() getVoltageMag(UnitType) setVoltageMag(double volt) setVoltageMag(double volt,UnitType type)	get and set voltage magnitude as a double, voltage unitType can also be used
getVoltageAng() getVoltageAng(UnitType) setVoltageAng(double volt) setVoltageAng(double volt,UnitType type)	get and set voltage angle as a double, voltage unitType (e.g., V, kV, pu) can also be used

- Get and set bus generation data

getGenCode() setGenCode(AclfGenCode genCode)	fours types of GenCode: Swing, PV, PQ and NoneGen
isGen() isSwing() isGenPV() isGenPQ()	check the bus type, based on the generation code.
getGenList() getGenerator(String genId)	Multiple generator connectin to the same bus is supported. The first method gets all the generator connected to the bus and return as a list; The second one get a specific generator by its id
get/setGen() get/setGenP() get/setGenQ()	1)For Swing bus, genP used to store VoltMag, genQ used to store VoltAng 2) For PV bus,genP used to store genP, genQ used to store VoltMag 3) For PQ bus, genP used to store genP, genQ used to store genQ
getGenResult()	get the effective power generation at the bus.

- Get and set bus load data

getLoadCode() setLoadCode(AclfLoadCode LoadCode)	tours types of LoadCode: Swing, PV, PQ and NoneLoad
isLoad()	check if there is any load at the bus
getLoadList() getLoad(String loadId)	get all the load connected to the bus and return as a list; get a specific load by its id
get/setLoad() get/setLoadP()	get the total in-service load of the bus in complex form, or real and reactive power of the load.

get/setLoadQ()	
getLoadResult()	get the actual load at the bus considering the load type and bus voltage at the moment.

- Get connected branches of a bus

getBranchList() getFromBranchList() getToBranchList()	return all the connected branches as a list, or the branches which are connected to this bus at the from end or the to end (put it another way, the branches whose trom-end /to-end terminal is this processing bus). **NOTE: Three winding transformer is also considered in this operation.**
nNonGroundBranchConnected(boolean inclActiveOnly)	The number of the total connected branches which are **NOT** ground branch, or connected to the ground, for example, shunt capacitor branch.
nBranchConnected	the number of the total connected branches
noConnectedBranch(AclfBranchCode type)	The number of the total connected branches of a certain type.

- Sort Number

Certain type of sorting algorithm (including Tinney 1, 2 and 3, and Tinney-2 is the default) is usually applied to arrange the internal sequence of the buses for load flow and short circuit analysis, and then a **sortNumber** is assigned to each bus after applying the sorting algorithm, both active and inactive (out-of-service). This **sortNumber** helps get access to the internal storage of Admittance matrix, Jacobian matrix and the network equation solution results.

Syntax:

```
bus.setSortNumber(int num)
bus.getSortNumber()
```

- Admittance

get/setShuntY()	get and set the bus shunt admittance Y= G+j*B
get/setYii()	get and set the bus self admittance

2.3.3 Branch

- From bus and to bus

set/getFromBus	set/get the from or to bus object or its bus Id.

set/getToBus	
set/getFromBusId set/getToBusId	set/get the from or tobus Id.

- Branch impedance and admittance

set/getZ()	branch impedance Z = r+ j*x
getY()	admittance Y =1/Z
get/setFromShuntY() get/setToShuntY()	shunt admittance at the from side and to side of the branch
get/setHShuntY()	half of the total branch shunt admittance

- Tap ratio and phase shift angle for transformer

get/setFromTurnRatio() get/setToTurnRatio() get/setTertTurnRatio() **[for three winding transformer only]**	InterPSS support to define the tap ratio at both sides
get/setFromPSXfrAngle() get/setToPSXfrAngle()	phase shift angle at the from side and to side

2.4 Example

The example **introNetworkObjectSample.java** under the *ch2_intro* folder of the tutorial package serves to help users and developers to learn how the models introduced before are used in network creation and bus data extraction. Within the sample, you will go through creating a simple 2-bus network and to output the bus data following specific txt format, with the InterPSS Core API.

The network creation part in this example is the same as that provided in this chapter before, so it is not replicated here. The following **busOrientedOutPut()** method is to output the bus voltage, generation and load and the power flow to connected buses.

```
private static String busOrientedOutPut(AclfNetwork net, AclfBus bus){

       StringBuffer str = new StringBuffer("");

str.append("-------------------------------------------------------------------------
-------------------------------------------------------------\n");
str.append(" Bus ID            Bus Voltage          Generation           Load
To             Branch P+jQ          Xfr Ratio    PS-Xfr Ang\n");
str.append("             baseKV  Mag/pu  Ang/deg    (mW)     (mVar)     (mW)     (mVar)
Bus ID      (mW)     (mVar)    (kA)    (From)   (To) (from)    (to)\n");
```

```
str.append("-------------------------------------------------------------------------------
-------------------------------------------------------------\n");


       double baseKVA = net.getBaseKva();

       AclfGenBusAdapter genBus = bus.toGenBus();

       //get the generation and load
       Complex busGen = genBus.getGenResults(UnitType.mVA);
       Complex busLoad = genBus.getLoadResults(UnitType.mVA);

       String id = bus.getId();

       //output the bus data with specific format

       str.append(Number2String.toStr(-12, id) + " ");
       str.append(String.format(" %s ", FormatKVStr.f(bus.getBaseVoltage()*0.001)));
       str.append(Number2String.toStr("0.0000", bus.getVoltageMag(UnitType.PU)) + "
");
       str.append(Number2String.toStr("##0.00", bus.getVoltageAng(UnitType.Deg)) + "
");
       str.append(Number2String.toStr("####0.00", busGen.getReal()) + " ");
       str.append(Number2String.toStr("####0.00", busGen.getImaginary()) + " ");
       str.append(Number2String.toStr("####0.00", busLoad.getReal()) + " ");
       str.append(Number2String.toStr("####0.00", busLoad.getImaginary()) + " ");

            //output the data of branches connected to the bus
           //
             int cnt = 0;
             //iterate over all the branches connected to the bus, both in-service and
off-line
             for (Branch br : bus.getBranchList()) {
                   if (br.isActive() && br instanceof AclfBranch) {
                         AclfBranch bra = (AclfBranch) br;

                         Complex pq = new Complex(0.0, 0.0);
                         double amp = 0.0, fromRatio = 1.0, toRatio = 1.0, fromAng =
0.0, toAng = 0.0;
                         AclfBus toBus = null;
                         if (bra.isActive()) {

                   // to determine whether the bus is at the from- or to-end of the
branch.
                          if (bus.getId().equals(bra.getFromAclfBus().getId())) {
                               toBus = bra.getToAclfBus();

                               // power transfer from from-end to to-end of the
branch
```

```
                                pq = bra.powerFrom2To(UnitType.mVA);

                                 //brnach current measured at Ampere.
                        amp = UnitHelper.iConversion(bra.current(UnitType.PU),
bra.getFromAclfBus().getBaseVoltage(),baseKVA, UnitType.PU, UnitType.Amp);

                        //if the branch is a transformer, then output the tap ratio and
                        //phase-shifting angle, if any
                               if (bra.isXfr() || bra.isPSXfr()) {
                                      fromRatio = bra.getFromTurnRatio();
                                      toRatio = bra.getToTurnRatio();
                                      if (bra.isPSXfr()) {
                                             AclfPSXformer psXfr = bra.toPSXfr();
                                             fromAng = psXfr.getFromAngle(UnitType.Deg);
                                             toAng = psXfr.getToAngle(UnitType.Deg);
                                          }
                               }
                        } else {
                        toBus = bra.getFromAclfBus();
                        pq = bra.powerTo2From(UnitType.mVA);
                        amp = UnitHelper.iConversion(bra.current(UnitType.PU),
bra.getToAclfBus().getBaseVoltage(),baseKVA, UnitType.PU, UnitType.Amp);
                               if (bra.isXfr() || bra.isPSXfr()) {
                                      toRatio = bra.getFromTurnRatio();
                                      fromRatio = bra.getToTurnRatio();

                                      if (bra.isPSXfr()) {
                                             AclfPSXformer psXfr = bra.toPSXfr();
                                             toAng = psXfr.getFromAngle(UnitType.Deg);
                                             fromAng = psXfr.getToAngle(UnitType.Deg);
                                      }
                               }
                        }
                }
        // if more than one branch connected to the bus, output the branch
        //information in a new line
                if (cnt++ > 0)
                        str.append(Number2String.toStr(67, " ") + "     ");
                        id = toBus.getId();
                        str.append(" " + Number2String.toStr(-12, id) + " ");
                        str.append(Number2String.toStr("####0.00", pq.getReal()) + " ");
                        str.append(Number2String.toStr("####0.00", pq.getImaginary()) + "
");

                        str.append(Number2String.toStr("##0.0##", 0.001 * amp) + " ");
                        if (bra.isXfr() || bra.isPSXfr()) {
                                if (fromRatio != 1.0)
                               str.append(Number2String.toStr("0.0###", fromRatio) + " ");
                               else
                                      str.append("       ");
```

```
                        if (toRatio != 1.0)
                              str.append(Number2String.toStr("0.0###", toRatio));
                        else
                              str.append("      ");

                        if (bra.isPSXfr()) {
                              if (fromAng != 0.0)
                                str.append("   " + Number2String .toStr("##0.0",
fromAng));
                              else
                                    str.append("        ");

                                if (toAng != 0.0)
                                str.append(" " + Number2String.toStr("##0.0",
toAng));
                                      else
                                            str.append("      ");
                              }
                              str.append("\n");
                        } else {
                              str.append("\n");
                        }
                  }
            }

            return str.toString();

      }
```

As you have learnt the basic and core models of InterPSS, you can roll up sleeves and start to play with InterPSS!

Chapter 3. Power system load flow analysis

Introduction to power system load flow

The goal of a power-flow study is to obtain complete voltage angle and magnitude information for each bus in a power system for specified load and generator real power and voltage conditions. For the generator buses, the exact voltage angles are not provided and are to be determined. Generally speaking, the voltage magnitude and the active power generation output are assumed to be known and kept constant during the solution. Considering the nonlinear relationship between voltage and power, load flow problem is a nonlinear problem by nature, numerical methods are employed to obtain a solution that is within an acceptable tolerance. Once the bus voltage magnitudes and angles are known, real and reactive power flow on each branch as well as generator reactive power output can be determined.

- **Bus type**

Based on the characteristics of the buses in the power system, they are usually categorized into three types, namely, Swing or slack bus, PQ bus and PV bus. They will be explained with a 5-bus system shown in Fig.3.1.

Swing/slack bus: Within one interconnected power system, a reference bus angle is required to determine the angle of the rest of buses, analogous to a reference zero-voltage point in an electric circuit. Also, as the total generation = total load + total loss in the system. The loss can not be exactly determined before the load flow, thus generation output of at least one bus need to be determined by the load flow to cover the system loss. This bus is called slack bus. For the sake of convenience, both swing and slack bus are combined and modeled by one bus with (large) generator connected to it. The bus voltage magnitude and voltage must be pre-defined (usually the angle is set to be zero degree). In the 5-bus system, there are two buses that has generator, thus either one can be chosen to be the swing bus. Suppose the Bus5 is selected as the swing bus.

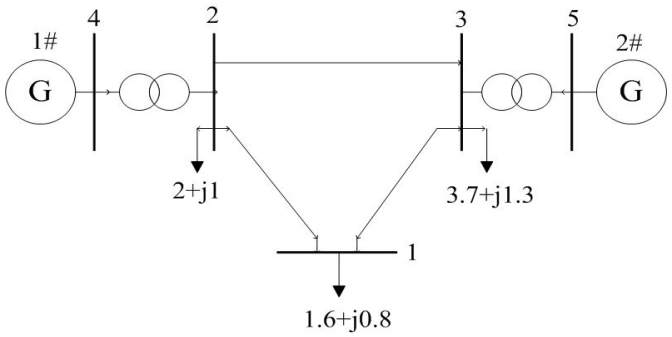


Fig.3.1 One-line diagram of a 5-bus power system

PQ bus: PQ means the bus net power injection into the network is constant, or as a static function of the voltage. This is usually used to model the substation serving local loads or a connection point in the system without any generation or load. Thus, Bus1 , Bus2 and Bus3 are PQ type bus.

PV bus: the real power P and voltage magnitude V are known for this kind of bus. This is usually used to model the bus with generator connected to it. Sometime a bus whose voltage is controlled by a local or remote reactive source can also modeled as a PV bus. Bus4 is a PV type bus.

3.1 Data required for load flow analysis

- Network data
- Bus data, including load data and generation data.
- Branch data, usually including both lines and transformers
- Other optional: HVDC, switched shunt

3.1.1 System/network data

- BaseMVA - All the per unit in the system for load flow study is based on the system baseMVA, which is usually set to be 100 MVA
- Area data, optional: For a large power system with multiple areas, area power interchange data should be provided.
- Zone data, optional

3.1.2 Bus data

- **Basic data:** bus Id, base voltage, bus type
- Load data (if the bus serves load (s)) : Load can be modeled as constant P, constant current , constant admittance or a combination of the three, which is called ZIP type load.
- Generation data (if the bus has generator(s))
    - PV bus: real power (P) and desired/scheduled voltage magnitude (V) must be provided, while the reactive power generation limit is usually required.
    - PQ bus: generation output (GenP + GenQ) must be defined and it is assumed to be constant.

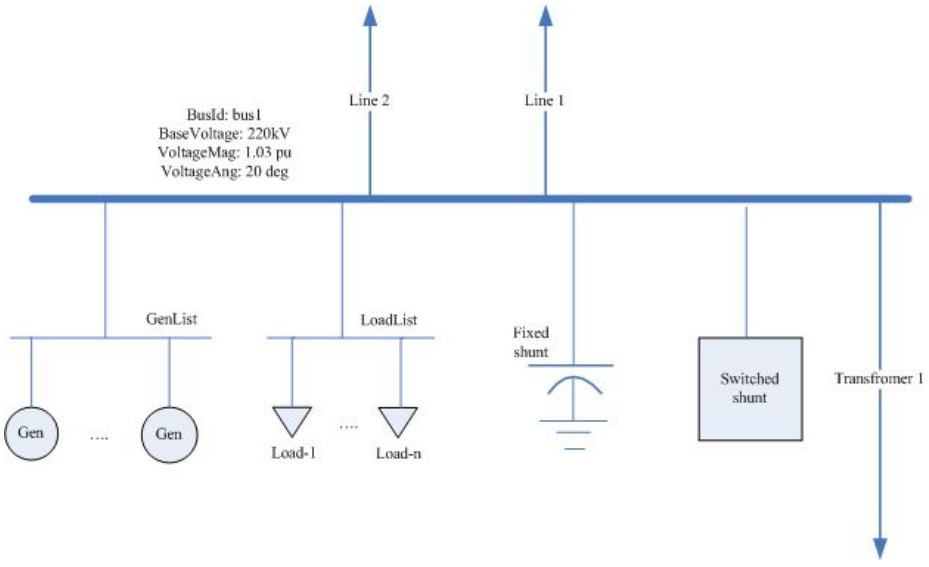


Fig.3.2 AclfBus model in InterPSS

- Fixed shunt data: It can be a fixed rector or shunt capacitor. The Var can be specified in PU (on system MVA rating) or MVAR = V^2*Bsh, Bsh is the bus fixed shunt.
- Switched shunt: the initial shunt, and the available var banks data is needed.

3.1.3 Branch data

- **Basic branch data:** fromBus, toBus, impedance Z, branch type
- Transmission line/cable: modeled as equivalent PI model. Shunt admittance is required besides the basic branch data
- Transformer
    - Two-winding transformer: tap ratio is required, for phase-shifting transformer, the phase-shifting angle is needed.
    - Three winding transformer: tap ratio on each winding is required, winding equivalent impedance Z12, Z23, Z13 should also be provided.

For further reading, please refer to the IEEE Common data format document, and find out how these data are documented and provided.

Please note that InterPSS itself does not have or limit to a specific data format for defining the data required for load flow analysis. Instead, it supports those widely used in the industry, e.g., IEEE CDF, PSS/E, PowerWorld, UCTE and BPA, through the adapters developed under the Open Data Model (ODM) project.

3.2. Supported power system models

Model	Details
Generation	one generator or multiple generators are allowed to connected to a bus. When there are more than one generators, they will be merged to an equivalent generator, whose genP is the total real power output of all generators.
Load	one load, or multiple loads are allowed to connected to a bus. constant power, constant current, constant impedance and ZIP type of loads are supported
Branch	line, two-winding transformer and phase shift transformer, 3 winding transformer.
Switched Shunt	supported switched shunt upto 8 Var banks, with three control modes: Fixed, Discrete and Continuous
Transformer correction table	compatible to the PSS/E model
HVDC	can be modeled in core engine, but not supported in the load flow algorithm yet
FACTs	can be modeled in core engine, but not supported in the load flow algorithm yet

3.3. Solution methods and internal sparse matrix data structure

AC load flow is the core algorithm for power system analysis, with no exception to InterPSS. The following three AC load flow solution methods are supported in InterPSS: Newton-Raphson, Fast Decoupled and Gauss. Moreover, the function of one-step load flow solution is also provided in InterPSS, such that users can run one-step solution and then check the result or apply adjustment or changes to the system, then run the next step solution. At the center of power network solution is the solution of sparse matrix equation

[A] x [x] = [B]

3.3.1 Newton-Raphson

Newton-Raphson (NR) is the default solution method in InterPSS. The NR implementation uses polar coordinates. Jacobian matrix can be formed by calling **aclfNetwork.formJMaxtrix()**. The matrix is represented by the ***SparseEqnMatrix2X2*** class in InterPSS. For more information of the sparse matrix used in InterPSS, please refer to the Appendix-A. The structure of the Jacobian-matrix is shown as

follows:

$$\begin{bmatrix} \frac{\partial P_1}{\partial \theta_1} & \frac{\partial P_1}{\partial V_1} & \cdots \\ \frac{\partial Q_1}{\partial \theta_1} & \frac{\partial Q_1}{\partial V_1} & \\ & & \ddots \\ & & \end{bmatrix} \begin{bmatrix} \Delta\theta_1 \\ \Delta V_1 \\ \\ \\ \end{bmatrix} = \begin{bmatrix} \Delta P_1 \\ \Delta Q_1 \\ \\ \\ \end{bmatrix}$$

- Power into the network is defined as the positive direction when calculating the power mismatch (dP, dQ)
- For each bus, its elements in the Jacobian matrix is always represented by a 2X2 block matrix

  [dP_dang dP_dV]
  [dQ_dang dQ_dV]

  This 2X2 block matrix is represented as **Matrix_xy** class in InterPSS, which has the following structure

  [xx xy]
  [yx yy]

  Thus, for a certain bus, if the corresponding Matrix_xy is obtained, then dP_dAng = Matrix_xy.xx , dP_dV = Matrix_xy.xy
- Bus arrangement is optimized before forming the Jacobian matrix
- The position of the vector [dPi, dQi] on the right hand side of the equation must be consistent with the 2X2 submatrix with the Jacobian-matrix. In addition, both are arranged according to the bus internal sortNumber.

For the purpose of customization, it is desirable to augment the original jacobian matrix in order to include extra equations for a new model or control strategy. For such requirement, extra dimensions can be added at the end of the J-matrix when calling the **formJMatrix(n, msg)** method, as shown in the following equation. When n =1, two rows and two columns (*to be consistent with the internal 2X2 block storage scheme*) will be added on the right and bottom of the original Jacobian matrix. The extra columns and rows could be used to implement new model, for example, SVC.

$$\begin{bmatrix} J & \vdots & \\ \cdots & \frac{\partial F_1}{\partial X} & \frac{\partial F_1}{\partial Y} \\ \cdots & \frac{\partial F_2}{\partial X} & \frac{\partial F_2}{\partial Y} \end{bmatrix} \begin{bmatrix} \Delta\theta_1 \\ \Delta V_1 \\ \vdots \\ \Delta X \\ \Delta Y \end{bmatrix} = \begin{bmatrix} \Delta P_1 \\ \Delta Q_1 \\ \vdots \\ \Delta F_1 \\ \Delta F_2 \end{bmatrix}$$

To implement a custom NR method, first step is to create a custom NR solver class, extending the DefaultNrSolver class and overriding the three methods. The DefaultNrSolver has methods to build the [J]

part and associated items in the right-hand side of eqn(1) ($\Delta Pi,\ \Delta Qi$), which should be inherited.

The `DefaultNrSolver` class defines the framework for customization. There are three methods to override.

```
class CustomNrSolver extends DefaultNrSolver {
   public CustomNrSolver(AclfNetwork net) {
      //Reuse the constructor of the DefaultNrSolver class
      super(net);
   }

   /**
    * formJMatrix method is called at the beginning of each NR iteration
    */
   @Override
   public SparseEqnMatrix2x2 formJMatrix(IPSSMsgHub msg) {
      // create network J-matrix with n extra-dimension
      int n = 1;
      SparseEqnMatrix2x2 lfEqn = getAclfNet().formJMatrix(n, msg);

      // at this point the original power network J-matrix is already stored
      // in the lfEqn. You can add extra elements here
      return lfEqn;
   }

   /**
    * setPowerMismatch method is called at the beginning of each NR
    iteration to  calculate the power mismatch to update the vector on the
    right hand side of the equation
    */
   @Override
   public void setPowerMismatch(SparseEqnMatrix2x2 lfEqn) {
      // calculate bus power mismatch. The mismatch stored on
      // the right-hand side of the sparse eqn

      super.setPowerMismatch(lfEqn);

       // at this point, bus power mismatch already stored in B[1, n]. You
       // add extra data to the right-hand side of the eqn
   }



   /**
    * updateBusVoltage method is called at at the end of each NR iteration,
    * after the sparse eqn has been solved. The results of the sparse eqn
    * solution is stored in the sparse eqn.
    */
   @Override
   public void updateBusVoltage(SparseEqnMatrix2x2 lfEqn) {
      // update the bus voltage using the solution results store in the
      // sparse eqn
      super.updateBusVoltage(lfEqn);

     // the solution result of the extra variable defined is stored at  //B[n+1
...]
   }
}
```

As for the part of newly added columns and rows, which is corresponding the derivation of existing bus and the newly added variables, if any, elements are of type **Matrix_xy** and are associated with bus object. You need to use bus.sortNumber to decide the element position in the matrix, since the bus number has been optimized to minimize the non-zero fill-ins in the LU decomposition. The following are some sample code:

```
Matrix_xy m = new Matrix_xy();
m.xx = …  // define the element values
m.xy = …

String id = ...; // get bus id associated with the element
Bus bus = aclfNet().getBus(id);
int i = bus.getSortNumber();  // get bus sortNumber
n = aclfNet.getNoBus();       // J-matrix dimension
lfEqn.setAij(m, i, n+1);      // n+1 point to the last column.
```

Then the custom NR solver class can be used to override the default NR solver in the LoadflowAlgorithm object.

```
AclfNetwork aclfNet = ...  // assume we have an AclfNetwork object

// create a Loadflow algo object
LoadflowAlgorithm algo = CoreObjectFactory.createLoadflowAlgorithm(msg);

// set algo NR solver to the CustomNrSolver
algo.setNrSolver(new CustomNrSolver(aclfNet));

// apply the algo to the aclfNet object to run Loadflow analysis.
aclfNet.accept(algo);
```

3.3.2 Fast Decoupled

The mathematical expression of the Fast Decoupled model is as follows:

$$\Delta P/V = [B1]\Delta\Theta$$
$$\Delta Q/V = [B11]\Delta V$$

where B1 and B11 matrix can be form by aclfNetwork.formB1Matrix() and aclfNetwork.formB11Matrix(), respectively.

This solution algorithm can be used by setting the method of load flow algorithm:

```
LoadflowAlgorithm algo = CoreObjectFactory.createLoadflowAlgorithm(net);
algo.setLfMethod(AclfMethod.PQ);
```

3.3.3 DC load flow

$$(\Delta P = B\Delta\theta)$$

Note: The application of B matrix in DCLF to sensitivity analysis and contingency analysis is provided in later chapter.

3.4 Adjustment During load flow

[Todo]
local adjustment: LfAdjustAlgorithm
system adjustment: NetAdjustAlgorithm

3.5 Configuration of load flow algorithm

setMaxIterations()	maximum iteration number
setTolerance()	tolerance in pu.
setInitBusVoltage()	This concerns whether flat start (bus voltage magnitude 1.0 , angle 0 degree) is enabled. If the initBusVoltage set to be true, then flat start will be used.
setLfMethod()	NR, PQ (Fast decoupled) are recommended.
setNonDivergent(boolean)	set true to use the non-divergent solution method by solving the load flow with optimal acceleration factor F as follows: $x(k+1) = x(k) + F * \Delta x$

3.6 Example

3.6.1 Run load flow and output result

```
package org.interpss.tutorial.loadflow;

import org.interpss.CorePluginObjFactory;
import org.interpss.IpssCorePlugin;
import org.interpss.display.AclfOutFunc;
import org.interpss.fadapter.IpssFileAdapter;
import com.interpss.CoreObjectFactory;
import com.interpss.common.exp.InterpssException;
import com.interpss.core.aclf.AclfNetwork;
import com.interpss.core.algo.LoadflowAlgorithm;

public class IEEE9BusLoadFlow {
```

```
	public static void main(String[] args) throws InterpssException {
		//Initialize logger and Spring config
		IpssCorePlugin.init();

		// import IEEE CDF format data to create a network object
		AclfNetwork net = CorePluginObjFactory
				.getFileAdapter(IpssFileAdapter.FileFormat.IEEECDF)
				.load("testData/ieee/009ieee.dat")
				.getAclfNet();

		//create a load flow algorithm object
		LoadflowAlgorithm algo = CoreObjectFactory.createLoadflowAlgorithm(net);
		//run load flow using default setting, which uses the NR method
		algo.loadflow();

		//output load flow result
		System.out.println(AclfOutFunc.loadFlowSummary(net));
	}
}
```

3.6.2 Customize NR load flow

```
package org.interpss.tutorial.loadflow;

import org.apache.commons.math3.complex.Complex;
import org.interpss.IpssCorePlugin;
import org.interpss.display.AclfOutFunc;
import org.interpss.numeric.datatype.Matrix_xy;
import org.interpss.numeric.sparse.ISparseEqnMatrix2x2;

import com.interpss.CoreObjectFactory;
import com.interpss.core.aclf.AclfNetwork;
import com.interpss.core.algo.LoadflowAlgorithm;
import com.interpss.core.algo.impl.DefaultNrSolver;
import com.interpss.simu.util.sample.SampleCases;

public class CustomLoadFlowExample {


	 /**
	  * Define a custom NR solver
	  *
	  */
	 static class CustomNrSolver extends DefaultNrSolver {
			public CustomNrSolver(AclfNetwork net) {
				super(net);
			}
```

```
	/**
	 * formJMatrix method is called at the beginning of each NR iteration
	 */
	@Override
	public ISparseEqnMatrix2x2 formJMatrix() {
		// create network J-matrix with one extra-dimension
	    // such that upto two addtional equations can be considered
	    // and included in the augmented Jacobian equations.

		ISparseEqnMatrix2x2 lfEqn = aclfNet.formJMatrix(1);

		// create a 2x2 matrix element
		Matrix_xy m = new Matrix_xy();
		m.xx = 2.0;
		m.xy = 0.0;
		m.yx = 0.0;
		m.yy = 2.0;

		// set the matrix element to J-matrix
		int n = aclfNet.getNoBus();

		// index is 0-based, which means the index originally is
		//0,1...n-1, now the last element index is n
		lfEqn.setA(m, n, n);

		return lfEqn;
	}

	// this is dummy variable for setting the extra mismatch field
	private double mis = 1.0;

	/**
	 * setPowerMismatch method is called at the beginning of each NR
	 * iteration
	 */
	@Override
	public void setPowerMismatch(ISparseEqnMatrix2x2 lfEqn) {
		// calculate bus power mismatch. The mismatch stored on
		// the right-hand side of the sparse eqn
		super.setPowerMismatch(lfEqn);

		// define a 2x1 vector
		Complex b = new Complex(1.0, 1.0);

		// set the vector to the right-hand side of the sparse eqn
		int n = aclfNet.getNoBus();
		//Again, index is 0-based, which means the index originally is
0,1...n-1, and now the last element index is n

		lfEqn.setB(b, n);
```

```
                }

                /**
                 * updateBusVoltage method is called at at the end of each NR
                   iteration, after the sparse eqn has been solved. The results
                   of the sparse eqn solution is stored in the sparse eqn.
                 */
                @Override
                public void updateBusVoltage(ISparseEqnMatrix2x2 lfEqn) {
                        // update the bus voltage using the solution results
                        // store in the sparse eqn
                        super.updateBusVoltage(lfEqn);

                        // the solution result of the extra variable defined is
                        //stored at B(n)
                        int n = aclfNet.getNoBus();
                        System.out.println("mis: " + this.mis + "  ---> " +
lfEqn.getX(n));

                  // reduce the dummy variable so that the loadflow can converge
                        this.mis *= 0.1;
                }
        }



        public static void main(String args[]) {
                //Initialize logger and Spring configuration
               IpssCorePlugin.init();

                // create a sample 5-bus system for Loadflow
                AclfNetwork net = CoreObjectFactory.createAclfNetwork();
                SampleCases.load_LF_5BusSystem(net);
                //System.out.println(net.net2String());

                // create a Loadflow algo object
                LoadflowAlgorithm algo =
                         CoreObjectFactory.createLoadflowAlgorithm();

                // set algo NR solver to the CustomNrSolver
                algo.setNrSolver(new CustomNrSolver(net));

                // run Loadflow, the custom NR Load flow algorithm is regarded
                //as a visitor
                net.accept(algo);

                // output loadflow calculation results
                System.out.println(AclfOutFunc.loadFlowSummary(net));
        }
}
```

Chapter 4. Short circuit analysis

Introduction to short circuit analysis

- **Superposition method**

The basic assumption of short circuit is that, during the very short period of interest (right after the fault), the fault current contributing sources, mainly generators and induction motors, can be regarded as constant voltage sources behind their corresponding impedances (usually either transient or subtransient impedance will be used). The voltage source are known, either based on the load flow solution or regarded as flat (vmag =1.0 pu). Further, the loads are all converted to the constant impedances. With these assumption and simplification, the network is a basically a linear circuit. From the electric circuit course, we learnt that **the superposition theorem** is valid for linear circuit and it is a powerful "tool" to analysis this kind of circuit.

Based on the superposition method, the total effect of network under the fault could be separated into two parts: 1) the effect of normal operation and 2) the effect of the fault current, which is illustrated in Fig.4.1.

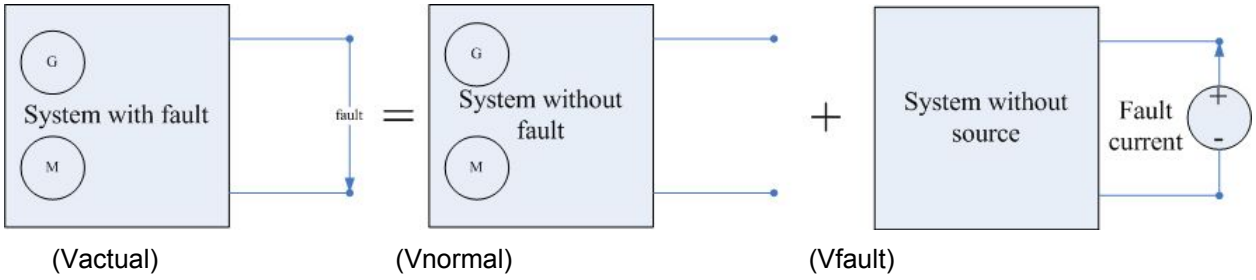


Fig.4.1 Superposition method for short circuit analysis

With this method, bus voltage is calculated by

Vactual = Vnormal + Vfault

where Vnormal is the pre-fault, normal operation voltage, Vfault is the bus voltage with the only source in the system is the fault current injection a the fault point.

Regarding the calculation of the fault current, it can be directly calculated through network solution for balanced faults. For unsymmectrical fault it is based on the symmetrical component and the positive sequence equivalent method, which will be discussed in the following.

- **Symmetrical component method**

In the paper[1], Fortescue found that a system of three unbalanced phasors can be transformed into two sets of balanced phasors and an addItional set of phasors, which are identical. These three sets of phasors are known as positive-, negative- and zero- sequence components. Under the condition of three phase symmetric impedance, the three components of the network are decoupled, which means they can be analyzed separately. Most of the actual power systems satisfy such condition, thus, the three-component methods is widely used. Based on the fault point boundary condition, the three

sequence networks are built and connected. Furthermore, the voltage sources only exist in positive sequence network, while there is no voltage source in negative- and zero- sequence network, thus both are passive circuit networks, and can be treated as equivalent impedance viewed at the fault point of the positive sequence network. This is the so-called positive sequence equivalent method.

For more information of the sequence impedance of each model and building of sequence network, please refer to the following **references for short circuit analysis :**

- English: P.M.Anderson, “Analysis of fault power systems”
- Chinese: 李光琦，电力系统暂态分析(第2/3版)

4.1 Power system sequence data

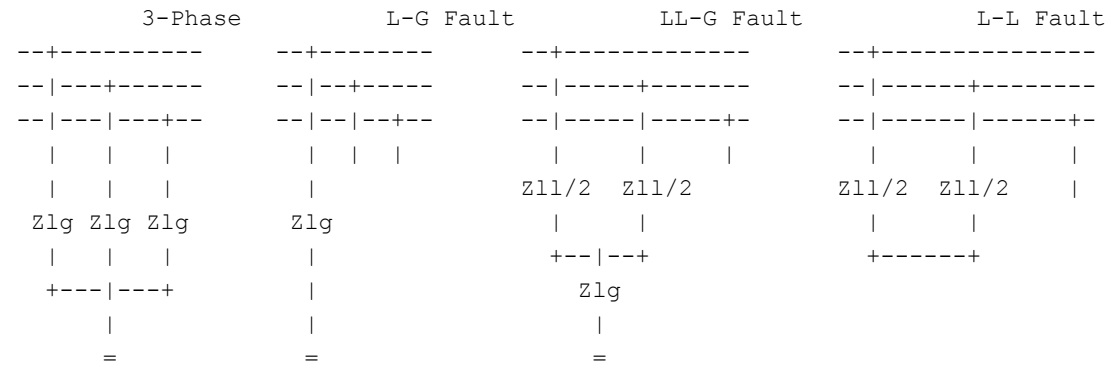
Model	Positive sequence	Negative sequence	Zero sequence
Generator	positive sequence impedance Generator sub-transient or transient impedance can be used here	Same as the positive sequence unless otherwise specified	Same as the positive sequence unless otherwise specified. However, usually it is less than positive sequence.
Load	All the loads are converted to constant impedance based on the bus voltage.	Same as the positive sequence unless otherwise specified	zero by default (due to the Delta-Wye connection of distribution transformer), unless otherwise specified
Non-transformer	same as the load flow data	Same as the positive sequence unless otherwise specified	zero impedance is **2.5-3.5** times of the positive impedance * zero sequence mutual impedance between two parallel lines is not supported yet
Transformer	same as defined in the load flow data	Impedance is Same as the positive sequence unless specified; Phase shift angle become opposite	Attention must be paid to the **transformer winding connection**. The connection and grounding info must be provided. if zero sequence path available, zero impedance is **same as** the positive impedance

Switch shunt	same as defined in the load flow data	Same as the positive sequence unless otherwise specified	zero by default, unless otherwise specified
HVDC	regarded as open or convert to equivalent load at terminals		zero by default, unless otherwise specified

Note: Currently, InterPSS supports the sequence data input with PSS/E V30 format, which is used in the second example in Section 4.5. A sample sequence data of IEEE 9 Bus is provided in the org.interpss.tutorial/testdata/psse/IEEE9Bus (download)

4.2 Bus based simple short circuit

Simple bus fault, symmetric and un-symmetric, is shown below:



Positive sequence equivalent

- 3-P fault

  zeq = zlg

- L-G fault

  zeq = zdd0 + zdd2, i1 = i2 = i0

  zdd0 = zdd0(net) + 3*Zlg

  zdd2 = zdd2(net)

- L-L fault

  zeq = Zll/2 + zdd2, i2 = -i1, i0=0

  zdd2 = zdd2(net) + Zll/2

- LL-G fault

  zeq = zll/2 + zdd0 || zdd2;

  zdd2 = zdd2(net) + Zll/2

  zdd0 = zdd0(net) + Zll/2 + 3*Zlg

Note:

Zlg : Phase to ground impedance

Zll : Total phase to phase impedance for Line to Line fault.

zdd0: The zero sequence equivalent impedance viewed at the fault point

zdd2: The negative sequence equivalent impedance viewed at the fault point

Negative- and zero- sequence current i2 and i0 are calculated based on positive sequence voltage v1 on

the fault point.

4.3 Branch based simple short circuit

[Todo]

4.4 Short circuit analysis in InterPSS

4.4.1 Create ACSC network

There is two ways to create an ACSC network object, one is through the input util provided by InterPSS, the other is by importing the industry standard data (currently only accept PSS/E v30 sequence data)

- Method 1: AcscInputUtilFunc and API

```
//create an acsc Bus
AcscInputUtilFunc.addScNonContributeBusTo(net, IdPrefix+"1", "Bus-1", 13800, 1,
1);

//create an acsc Branch
AcscBranch bra = CoreObjectFactory.createAcscBranch();
bra.setBranchCode(AclfBranchCode.LINE);
bra.setZ( new Complex( 0.0, 0.25 ));
bra.setZ0( new Complex(0.0,0.7));
net.addBranch(bra, IdPrefix+"1", IdPrefix+"2");
```

For a complete example, please refer to the load_SC_5BusSystem (AcscNetwork net) method in Acsc5BusExample.Java

Note: As all the data is set by coding, This method, if directly used, is suitable for defining a small system. However, you may use the APIs and develop your customized data importer, to load data from csv, excel or any other format data file to create an Acsc network.

- Method 2:PSSEAdapter

An example of using the PSSEAdapter to import and create an ACSC network object is provided below:

```
PSSEAdapter adapter = new PSSEAdapter(PsseVersion.PSSE_30);
        assertTrue(adapter.parseInputFile(NetType.AcscNet, new String[]{
                    "testData/psse/IEEE9Bus/ieee9.raw",
                    "testData/psse/IEEE9Bus/ieee9.seq"
        }));
   AcscModelParser acscParser =(AcscModelParser) adapter.getModel();

   AcscNetwork net = new
            ODMAcscParserMapper().map2Model(acscParser).getAcscNet();
```

4.4.2 Define a fault

Information needs to define a fault:

- fault point, bus or branch
- fault type
- fault impedance, including zlg and zll

Here is an example of defining a 3-phase bus fault:

```
AcscBusFault fault = CoreObjectFactory.createAcscBusFault("Bus4", acscAlgo );
fault.setFaultCode(SimpleFaultCode.GROUND_3P);  // fault type
fault.setZLGFault(new Complex(0.0, 0.0));     // fault impedance zlg
fault.setZLLFault(new Complex(0.0, 0.0));     // fault impedance  zll
```

4.4.3 Calculate short circuit

The setting of the pre fault bus voltage profile--is required. It can be based on the solved power flow (ScBusVoltageType.LOADFLOW_VOLT) or Flat voltage (ScBusVoltageType.UNIT_VOLT)

Example:

```
//pre fault profile : solved power flow
acscAlgo.setScBusVoltage(ScBusVoltageType.LOADFLOW_VOLT);
acscAlgo.calculateBusFault(fault);
```

4.4.4 Obtain results

- Fault current: either abc or 012 coordinate

```
fault.getFaultResult().getSCCurrent_012();
```

- Bus voltage: either abc or 012 coordinate

```
fault.getFaultResult().getBusVoltage_012(Bus);
```

4.5 Example

The corresponding code of the examples is provided in the ch4_shortcircuit package of the tutorial project

4.5.1 Build a system for short circuit analysis

See the example of **Acsc5BusExample.java**

4.5.2 Short circuit analysis with load flow and sequence data

See the example of IEEE9Bus_Acsc_test.java

Reference:

[1] Fortescue, Charles L. "Method of symmetrical co-ordinates applied to the solution of polyphase networks." *American Institute of Electrical Engineers, Transactions of the* 37.2 (1918): 1027-1140.

Chapter 5. Transient stability simulation

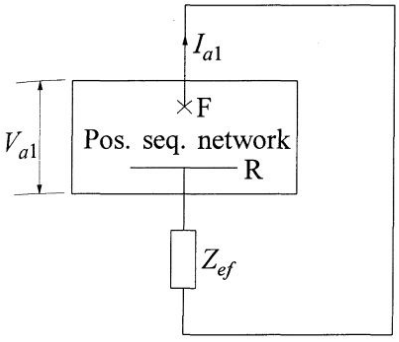
5.1 Introduction to transient stability simulation

- **Electromechanical transient stability**

Transient stability is the ability of the power system to maintain synchronism when subjected to a severe transient disturbance such as a fault on the transmission line, loss of large generation or loss of large load. The synchronism is mainly measured by the largest generator angle difference among all the generators, thus it is heavily related to the dynamic response of the generators to the fault, which is governed by both their mechanical input and electrical output. Thus the stability of concern is also known as electromechanical transient stability.

- **Stability studies with positive sequence network**

Power systems are usually operated under balanced conditions, and such three-phase systems can be represented by an equivalent single-phase network, with all voltage and currents represented in phasor form. When unsymmetrical faults are considered, the three-phase system can be modeled as three symmetrical systems ,i.e., positive-, negative- and zero- sequence systems, using the symmetrical component method. Considering that the motion of the generators is mainly affected by the positive sequence, and that the negative- and zero-sequence voltage and currents are not of interest in stability studies, stability studies can concentrate on positive sequence network, with the overall effects of the negative and zero sequence network on the positive sequence represented by effective equivalent impedance viewed at the fault point, which is shown as follows:



Fault type	Z_{ef}
Line-to-ground	Z_2+Z_0
Line-to-line	Z_2
Double line-to-ground	$\frac{Z_2 Z_0}{Z_2+Z_0}$
Three-phase	0

Fig.5.1 Positive sequence equivalent method [1]

Diagram below shows the key models of interest (generation, load and the network) and the interactions of them

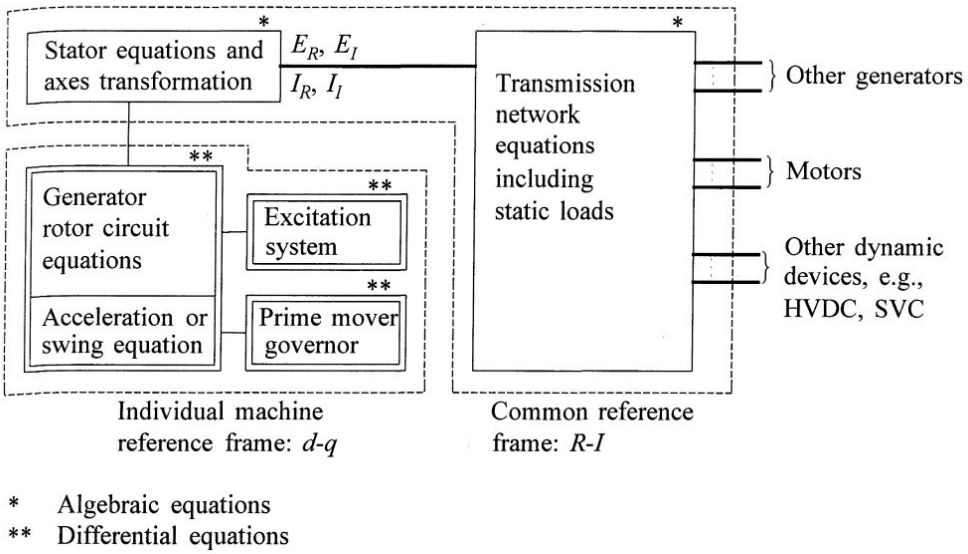


Fig.5.2 Structure of the complete power system model for transient stability simulation[1]

With the network modeled by nodal admittance, the network loading components (load, generator, SVC, HVDC, etc) are converted into Norton equivalents of injected currents in parallel with admittance. For constant impedance load, the current injection is zero. The equivalent admittances are added into the network to form a modified admittance matrix. The the network equation becomes:

[I] = [Y][V]

Solution process is summarized as follows:

(1) For each loading component, calculate the current injections by solving its differential and algebraic equations.
(2) Determine network voltages from the injection currents

As the bus voltages affect the loading components, an iterative process is required for the above two steps.

5.2 Dynamic models

5.2.1 Machine model

(1) Machine model in a DStabBus

Recall the bus model depicted in Fig.2.3, the generator for load flow study is modeled by a AclfGen, which is an entry in the GenList of a bus. For dynamic study, DStabGen model is used, and it is extended from the AclfGen with the following hierarchy structure:

AclfGen ← AcscGen ← DStabGen

Machine is not directly connecting to a bus, instead, it is contained within a DStabGen object. The relationships among the bus, dstabGen, machine, and other machine controllers (turbine-governor, exciter and pss) can be illustrated by the following Fig.5.3. Thus, a dstabGen must be created and added to the genList before the creation and modeling of a machine. This requirement is also applied to a machine controller as to a machine.

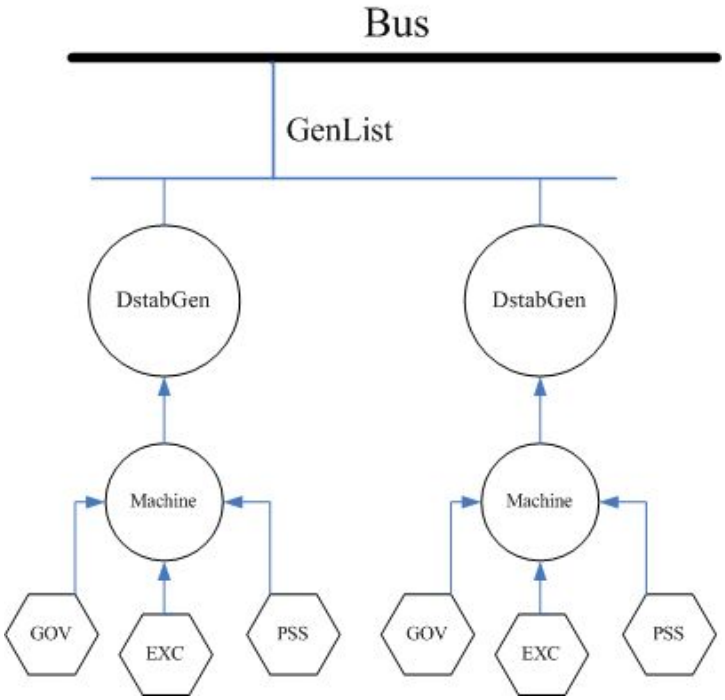


Fig.5.3 Relationships of controller, machine, dstabGen and bus

(2) Machine models of different levels of modeling details

InterPSS machine model implementation is based on IEEE Std 1110-2002: IEEE Guide for Synchronous Generator Modeling Practices and Applications in Power System Stability Analysis. Currently the following models are implemented.

Tab.5.1 Machine models

InterPSS Machine Model	IEEE Std 1110-2002 Model	Modeling consideration axis windings	Note
EConstant	N/A	N/A	• classical model or constant voltage behind reactance model • 2 order
InfinitBus	N/A	N/A	• To model the infinite bus, similar to classical modell with very large machine MVA, e.g., 999999 MVA

			• 2 order
Eq1 Model	IEEE 1.0	Only field winding f	• Salient pole model • 3 order
Eq1 Ed1 Model	IEEE 1.1	• d-axis: field winding f • q-axis: Q damper winding	• Two-axis model • 4 order
E11 Round Rotor	IEEE 2.2	• d-axis: field winding f, damper winding D • q-axis: damper winding Q and G	• detailed model for round rotor machines, • 6 order
E11 Salient Pole	IEEE 2.1	• d-axis: field winding f , damper winding D • q-axis: damper winding Q	• detailed model for salient pole machines, • 5 order

Tab.5.2 Machine modeling parameters and unit

Parameter	Unit
Machine rating—Rating	MVA
Machine rated voltage—Rated Volt	V
Shaft Mechanical Damping Factor—D	%MW/Hz
Armature Leakage Reactance—Xl	pu
Armature Resistance—Ra	pu
Synchronous-Direct Axis Reactance—Xd	pu
Synchronous-Quadrature Axis Reactance—Xq	pu
Transient-Direct Axis Reactance—Xd1	pu
Transient-Quadrature Axis Reactance—Xq1	pu
Open Circuit Transient-Direct Axis Time Constant—Td01	s
Open Circuit Transient-Quadrature Axis Time Constant—Tq01	s
Subtransient-Direct Axis Reactance—Xd11	pu
Subtransient-Quadrature Axis Reactance—Xq11	pu

Open Circuit Subtransient-Direct Axis Time Constant—Td011	s
Open Circuit Subtransient-Quadrature Axis Time Constant—Tq011	s
Saturation Factor at 100% Terminal Voltage—S_{E100}	%
Saturation Factor at 120% Terminal Voltage—S_{E120}	%
Poles	optional, for information only

Please Note :

- Per Unit system - All machine parameters, you entered into InterPSS are, assumed based on the machine rating and machine rated voltage. Internally, these machines are transferred to the system Kva base and the bus base voltage, to which the machine is connected to, when necessary.
- Machine output - All machine output, such as Pm, Pe are based on machine rating.
- **Round rotor model:**

$$\left.\begin{aligned}
\frac{dE'_q}{dt} &= \frac{1}{T'_{d0}}[E_{fq} - k_d E'_q + (k_d - 1)E''_q] \\
\frac{dE''_q}{dt} &= \frac{1}{T''_{d0}}[E'_q - E''_q - (X'_d - X''_d)I_d] \\
\frac{dE'_d}{dt} &= \frac{1}{T'_{q0}}[-k_q E'_d + (k_q - 1)E''_d] \\
\frac{dE''_d}{dt} &= \frac{1}{T''_{q0}}[E'_d - E''_d + (X'_q - X''_q)I_q]
\end{aligned}\right\}$$

$$k_d = \frac{X_d - X''_d}{X'_d - X''_d}, \qquad k_q = \frac{X_q - X''_q}{X'_q - X''_q}$$

- **Salient rotor model:**

It is almost the same as above, except there is only one damper winding in q axis, therefore this is no modeling of E'd , with x'q = x''q and T'q0 = 0.

(3) Modeling the effects of saturation

- Default method: Quadratic function:

$$S = \frac{B(E-A)^2}{E}$$

E is the input and A and B are determined by fitting the two points input as generator parameters, i.e. (1.0, S1.0) and (1.2, S1.2)

- Other available methods:

- Exponential function

$$S = S_{1.0} \times E^{X}$$

where:

$$\text{where } X = \frac{\ln\left(\frac{S_{1.2}}{S_{1.0}}\right)}{\ln(E_{1.2})} \text{ and E is the input.}$$

- Three section method

Saturation data input

Sliner - Voltage at the End Point of Liner Area on Generator OCC Curve. 0.8-0.9 pu is recommended

When $|\dot{E}_t| \le V_o$ $X_d(t) = X_{adu} + X_l$

When $1.3 \ge |\dot{E}_t| > V_o$ $X_d(t) = X_{adu} * K_{sd}$

When $|\dot{E}_t| > 1.3$ $X_d(t) = X_{adu} * K_{sd2}$

Where,

V_o , S_{E100} , S_{E120} are parameters defined by user.

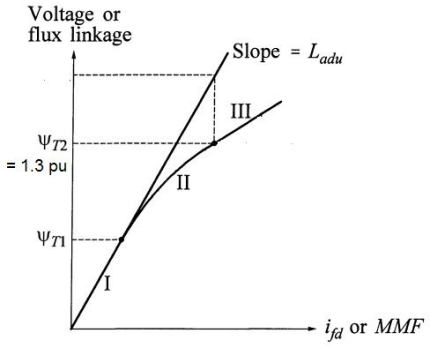
S_{E100} , S_{E120} are less than 100% and $S_{E100} < S_{E120}$.

$\dot{E}_t = \dot{V}_{bus} + \dot{I} * X_l$ air-gap voltage, $\dot{V}_{bus}$, $\dot{I}$ are machine terminal bus voltage and machine current.

$X_{adu} = X_d - X_l$

$K_{sd} = 1.0 + A e^{B*(Et - Vo)} / |E_t|$

$K_{sd2} = 1.0 + A e^{B*(1.3 - Vo)} / |E_t|$



5.2.2 Excitor

[Todo] Add more description

5.2.3 Turbine and governor

[Todo] Add more description

5.2.4 PSS

[Todo] Add more description

5.2.5 Load model

Now the loads can be represented as constant current, constant impedance and constant power types of load for transient stability simulation. Constant power load is converted to constant impedance load based on the load flow result, by default, within the DStabNetwork initialization process.

[Todo] Add more description

5.2.7 Bus Frequency Measurement

Bus frequency is measured by the change rate of bus voltage angle. The transfer function for the measurement is shown in the following diagram:

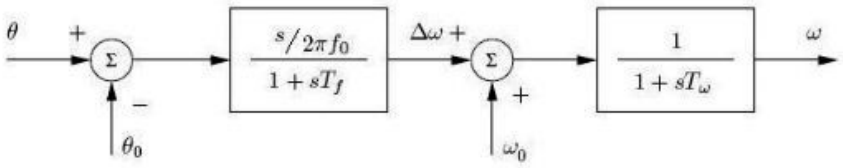


Fig.5.4 Bus frequency measurement block diagram

The following values are set:

Tf = 0.01 sec
Tw = 0.01 sec

You can modify the properties/coreLibContext.xml to change their values:

```
<bean id="busFreqMeasurementImpl"
      class="org.interpss.dstab.measure.BusFreqMeasurementImpl"
      scope="prototype">
          <constructor-arg index="0"><value>0.01</value></constructor-arg>
          <constructor-arg index="1"><value>0.01</value></constructor-arg>
</bean>
```

InterPSS bus frequency implementation Java source code can be found Here.

5.3 Numerical Solution

As shown by the diagram in section 5.1, the dynamic components in power system, for example, the generators and controllers, are modeled by ordinary differential equations, while the transmission network and the static loads are represented by algebraic equations. Thus, in general, the power system for transient stability simulation can be modeled as Differential Algebraic equations (DAEs):

$$\dot{X} = F(X, V) \qquad (5.1)$$

$$I(X, V) = YV \qquad (5.2)$$

There is generally two solution methods for the DAEs above, one is iterative method and there other is to solve both simultaneously using implicit solution, e.g. Trapezoidal method. InterPSS choose the former method. For each step, differential equations for the dynamic models (Eqn. 5.1) are solved first using the Modified Euler method, then the dynamic models interface with the network as Norton Equivalent (i.e., current sources in parallel with equivalent admittance).

As the interface error exists for the iterative method, iteration between network equation and dynamic model solution is often required to eliminate such errors. During the period when the system experiences considerable changes in the state variables, the iteration is conducted for 6 times, while maximum iteration is set to be 4 for the rest of the simulation.

5.4 Simulation procedure

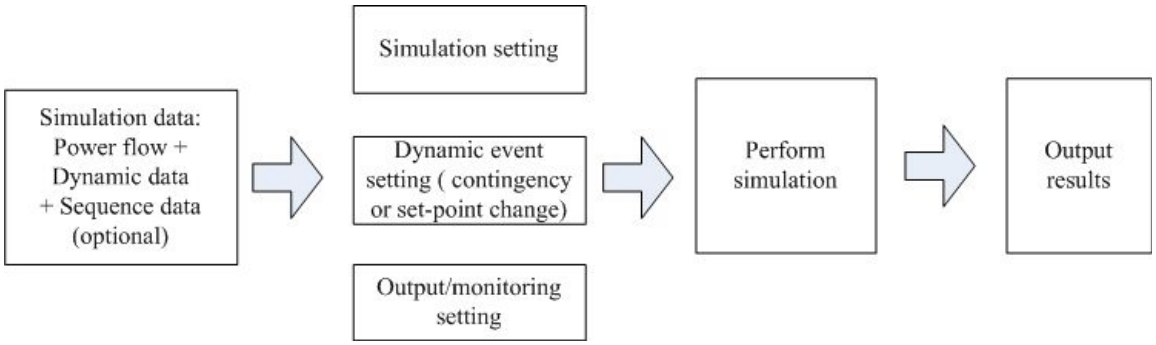


Fig.5.5 Flow diagram of InterPSS transient stability simulation procedure

5.4.1 Simulation data preparation

For transient stability simulation, load flow data and the dynamic model data is the minimum required data to conduct transient stability simulation for symmetrical fault. The sequence network data must also be included when any unsymmetrical fault is considered.

Now InterPSS supports load flow data defined in a variety of formats, for example, IEEE CDF, PSS/E(V29-V33), PowerWorld (v16), GE PSLF. However, for the dynamic model and sequence network data, the ODM adapter now supports the PSS/E and PSD-BPA format data as well as data in ODM/XML format. Dynamic models defined in other formats might be supported in the near future.

5.4.2 Simulation setting

Simulation time	DynamicSimuAlgorithm .setTotalSimuTimeSec()	total simulation time in seconds
Simulation time step	DynamicSimuAlgorithm .setSimuStepSec()	should be less than half of the smallest time constant in the system
Simulation method	DynamicSimuAlgorithm.setSimuMethod(DynamicSimuMethod.MODIFIED_EULER)	now ONLY the modified euler is supported.
Reference machine	DynamicSimuAlgorithm..setRefMachine(dsNet.getMachine(MachineId));	MachineId is formed as: BusId_GeneratorId

Network solution iteration **(optional)**	dsNet.setNetEqnIterationWithEvent(int); dsNet.setNetEqnIterationNoEvent(int);	The default values are 6 (with any change) and 4 (no event), respectively

5.4.3 Event setting

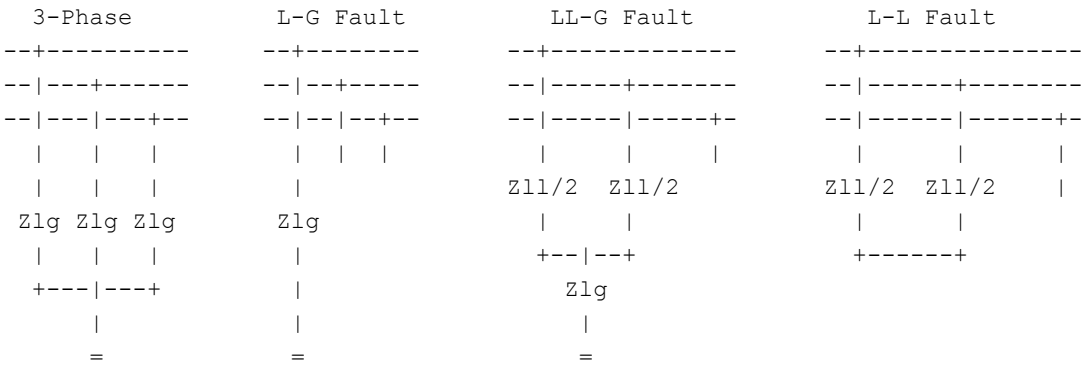
The event concept in InterPSS includes both the faults or device setPoint changes.

Event type	Note
Bus_Fault	all kinds of bus faults, LG, LL, LL-G, 3P
Branch_Fault	
Branch_Outage	
Branch_Reclose	
Load_Change	
SetPoint_Change	Control reference set point change, e.g., Vref

(1) Fault setting

(a) Bus fault

Bus fault needs to specify the 1) fault bus, 2) fault type and 3) fault impedance (ZLG and ZLL). Simple bus fault, symmetric and un-symmetric, is shown below:



Note: Any kind of network topology and/or parameter setting change, e.g., fault or reference point setting change, is modeled as an **Event** in InterPSS.

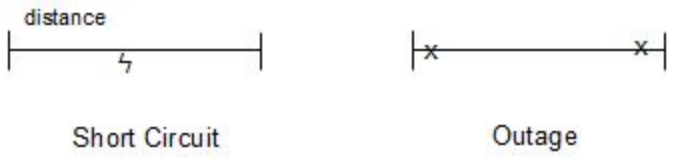
An example for creating a three phase solid ground fault is given below:

```
        // define an event, set the event id and event type.
        DynamicEvent event1 = DStabObjectFactory.createDEvent("BusFault3P@"+faultBusId,
"Bus Fault 3P@"+faultBusId, DynamicEventType.BUS_FAULT, net);
        event1.setStartTimeSec(startTime);
        event1.setDurationSec(durationTime);

       // define a bus fault
        DStabBus faultBus = net.getDStabBus(faultBusId);
        AcscBusFault fault = CoreObjectFactory.createAcscBusFault("Bus Fault
3P@"+faultBusId, net);
        fault.setAcscBus(faultBus);
        fault.setFaultCode(SimpleFaultCode.GROUND_3P);
        fault.setZLGFault(NumericConstant.SmallScZ);

       // add this fault to the event, must be consist with event type
       // definition before.
        event1.setBusFault(fault);
```

The event can be added to the DStabilityNetwork before running the dynamic simulation as follows:

```
        DStabilityNetwork.addDynamicEvent(event1)
```

(b) Branch Fault



There are two types of branch fault: short circuit and outage, as shown in the above diagram.

- **Branch Short Circuit** : Branch short circuit is simulated by creating an equivalent fault bus some distance from a terminal bus.

- **Branch outage** : Branch outage is modelled by inserting an equivalent Z between the two terminal buses.

[TODO] Add more sample for other types of events

5.4.4 Monitoring and output

(1) State Variable Recorder

The usage of state variable recorder can be found in the IEEE 9 Bus example DStab_IEEE9Bus_Test.java under the ch5_dstab of the tutorial.

```
StateVariableRecorder ssRecorder
   = new StateVariableRecorder(0.0001); // time  tolerance
ssRecorder.addCacheRecords("Bus2-mach1",// mach id
                MachineState,           // record type
```

```
        DStabOutSymbol.OUT_SYMBOL_MACH_ANG,   // state variable name
                          0.005,              // time steps for recording
                          1000);              // total points to record

              // set the output handler
               dstabAlgo.setSimuOutputHandler(ssRecorder);


            //Note: Here the dstab simulation part is skipped.

       // output recorded simulation results after the simulation.
         List<StateRecord> list = ssRecorder.getMachineRecords(
               "Bus2-mach1", MachineState,
          DStabOutSymbol.OUT_SYMBOL_MACH_ANG);

        System.out.println("\n\n Bus2 Machine Anagle");

       for (StateRecord rec : list) {
            System.out.println(Number2String.toStr(rec.t) + ", " +
                  Number2String.toStr(rec.variableValue));
        }
```

Per the setting above, the state variables recorder will keep a record of the machine angle of the Machine with Id “Bus2-mach1”, for every 0.005 seconds. We recommend users setting the time step to be one or multiple times the simulation time step. After the simulation

(2)State Monitor

Different from the stateVariableRecorder, which is time-stamped, and variables have to defined one by one. StaeMonitor is designed to help user easily monitor multiple important variables, e.g., machine angle, pe, pm, efd, bus voltage and angles, with easy setting and output. It is simulation step based.

The usage of the stateMonitor is demonstrated as follows:

```
            StateMonitor sm = new StateMonitor();

       //sm.addGeneratorStdMonitor(machId)
        sm.addGeneratorStdMonitor(new String[]{"Bus14931-mach1"});

       //sm.addBusStdMonitor(busId)
sm.addBusStdMonitor(new String[]{"Bus24151","Bus15021","Bus24085"});

        // set the output handler of DStabAlgorithm
        dstabAlgo.setSimuOutputHandler(sm);

        // set output frequency, measured by steps
        dstabAlgo.setOutPutPerSteps(1);
```

```
        // after the simulation, output the monitored variables or parameters:

            System.out.println(sm.toCSVString(sm.getMachAngleTable()));
            System.out.println(sm.toCSVString(sm.getBusAngleTable()));
            System.out.println(sm.toCSVString(sm.getBusVoltTable()));

       //or save it to a csv file
          FileUtil.writeText2File("E:/mach_angle.csv",
                sm.toCSVString(sm.getMachAngleTable()));
```

Output sample: Bus voltage

time,Bus30000, Bus14931, Bus24801, Bus24085, Bus47216, Bus15021, Bus24151
0.000, 1.030, 1.000, 1.036, 1.007, 1.020, 1.060, 1.049,
0.004, 1.030, 1.000, 1.036, 1.007, 1.020, 1.060, 1.049,
0.008, 1.030, 1.000, 1.036, 1.007, 1.020, 1.060, 1.049,
0.013, 1.030, 1.000, 1.036, 1.007, 1.020, 1.060, 1.049,

5.4.5 Load flow and system initialization

Load flow result is always required to determine the operating point of the system before any fault is considered.

```
DynamicSimuAlgorithm dstabAlgo  =....
LoadflowAlgorithm aclfAlgo = dstabAlgo.getAclfAlgorithm();
aclfAlgo.loadflow();
// make sure load flow is converged before dstab initialization
```

System initialization is then performed based on a converged load flow result, and it mainly includes four parts:

- Generator initialization:
  - Map generator sequence network data to the bus which it connects to, to form the equivalent admittance (Recall that the generators are converted and represented as current source in parallel with equivalent admittance, Yeq = 1/ Zsource)
  - Initialize variable states of generators and their controllers

- Load conversion

Constant power load is usually to constant admittance load: YLoad = (PL-j*QL)/V^2 and added to the bus shunt admittance.

- Device initialization

For other dynamic devices except generators, for example, induction motors, initialization process is also required to determine the states under pre-fault conditions.

- Form the positive sequence admittance matrix, which will be used in the network equation solution I= YV during simulation.

➔ Syntax: `DynamicSimuAlgorithm.initialization()`

5.4.6 Simulation

- Normal (successive) simulation

.Syntax: `DynamicSimuAlgorithm.performSimulation()`

- One-step simulation

For customization or extension purpose, one might what to stop after each or certain step, then make some changes or control to the system and continue to the next step simulation. In this regard, a so-call ”one step simulation” function is provided

(1) perform One-step Simulation
Syntax: `DynamicSimuAlgorithm.solveDEqnStep(boolean updateTime)`

(2) get current time in sec during simulation
Syntax: `DynamicSimuAlgorithm.getSimuTime()`

5.6 Data check and auto correction

Model data check is performed as part of the initialization process, while limited data correction have been considered so far.

5.7 Development of new dynamic device

[Todo]

5.8 Example

- IEEE 9 Bus system

This example within the tutorial under the ch5_dstab package includes all the steps of running dynamic simulation in InterPSS.

- IEEE 39 Bus system
- Development of SVC as a new dynamic device

Reference

[1] Kundur, Prabha. *Power system stability and control*. New York: McGraw-hill, 1994.

Chapter 6. Power system optimization through integrating InterPSS with GAMS

The purpose of this document is to provide instruction, including sample code, for InterPSS and GAMS integration.

6.1 GAMS V24

(1) Overview of GAMS new API

The object-oriented GAMS API allows the seamless integration of GAMS into an application by providing appropriate classes for the interaction with GAMS. The ***GAMSDatabase*** class for in-memory representation of data can be used for convenient exchange of input data and model results. Models written in GAMS can be run with the ***GAMSJob*** class and by using the ***GAMSModelInstance*** class a sequence of closely related model instances can be solved in the most efficient way. There are three versions of the object-oriented GAMS API: **Java**, **Python** and**.NET**. These APIs work with Java SE 5 and up, Python 2.7, and .NET framework 4 (Visual Studio 2010) and up. For details: see http://www.gams.com/dd/docs/api/

It is recommended to read through the GAMS Java API doc and the tutorial to learn more specific information regarding the Java APIs and usage.

http://www.gams.com/dd/docs/api/GAMS_java.pdf

http://www.gams.com/dd/docs/api/GAMS_java_Tutorial.pdf

(2) Installation

1. Download from GAMS download webside, **V24.0.2 or newer version** should be consistent with this document.

NOTE: Attention should be paid to choosing the 64-bit or 32-bit version GAMS. It was found that GAMS must be consistent with the JRE. That is, if the JRE installed in your machine is 64-bit, the 64-bit version GAMS should be your choice, otherwise the 32-bit version.

Error occurs when inconsistent version of GAMS and JRE are used:

Exception in thread "main" com.gams.api.GAMSException: expect 64-bit GAMS system in [C:\Program Files (x86)\GAMS24.0], but found 32-bit instead!

2. When installing the GAMS, make sure choose "*add the install dir to the system environment*".

6.2. Call GAMS from Java

Since v24.0, new GAMS Java APIs introduce the following concepts/classes, allowing better integration with Java-based projects.

- **GAMSWorkspace :** Workspace in Java environment which has most of the functions in the native GAMS workspace. To Integrate GAMS in any Java project, first we need to create an GAMSWork space, as follows:

  GAMSWorkspace ws = new GAMSWorkspace();

  Then we can use **ws** to create database

  GAMSDatabase db = ws.addDatabase();

  and create GAMSJob:

  GAMSJob ieee14ED = ws.addJobFromString(modelStr)

- ***GAMSDatabase:*** storing and processing the modeling data in in-momory database
- GAMSSet and GAMSParameter: Base data set for storing the indices and modeling data, respectively.

  Example:

```
GAMSSet loadBus = db.addSet("j", 1, "load buses");
GAMSParameter genPLow = db.addParameter("genPLow", 1,"lower bound of each generating unit")
```

- **GAMSVariableRecord:** Get the optimization results directly from the database

Example:

```
for (GAMSVariableRecord rec : ieee14ED.OutDB().getVariable("genp")){
    System.out.println("genP @ Bus-" + rec.getKeys()[0]+"  : Level=" + rec.getLevel() );
}
```

- ***GAMSJob:*** Now we can create GAMSJob from modeling String, no independent *"*.gms"* modeling file is needed.
- Example:

```
GAMSJob ieee14ED = ws.addJobFromString(modelStr);
```

The GAMS Java Library **GAMSJavaAPI.jar** can be found under the folder:

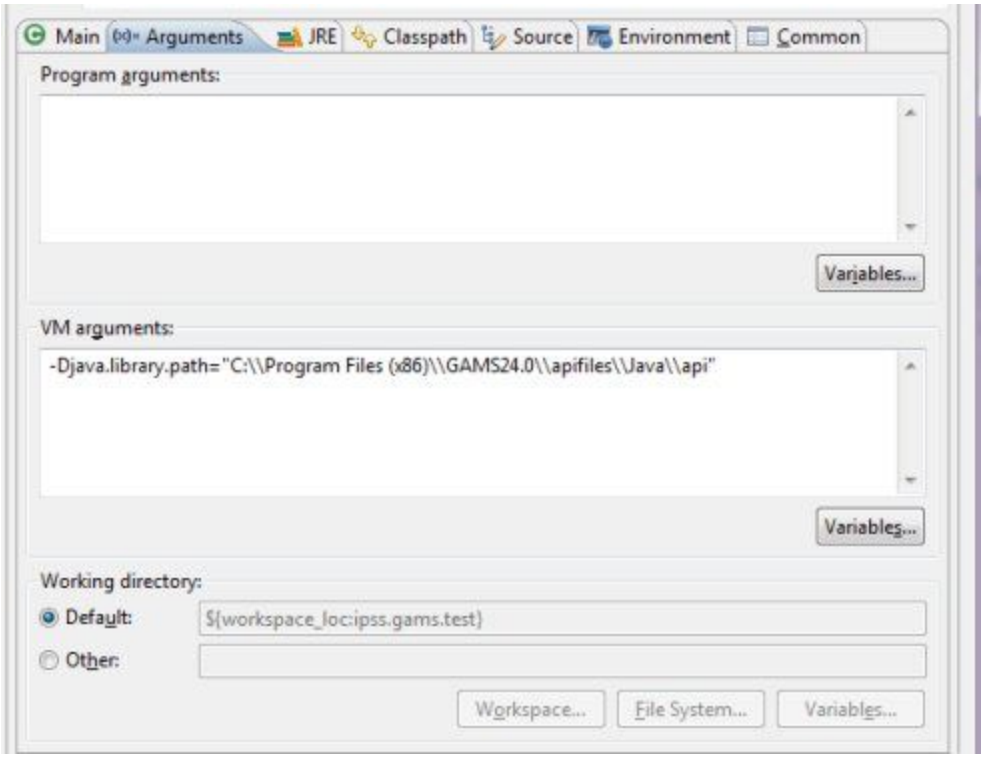
```
<Path to GAMS>\apifiles\Java\api
```

(Corresponding DLLs can also be found under this folder). This has been add to the *ipss.lib.3rdPty/gams*

Note: GAMS Related class *run configuration* setting within Eclipse

Run the class by run->run configuration…->setting the VM arguments as follows:

```
-Djava.library.path="<GAMS Install Dir>\\apifiles\\Java\\api"
```



Which guarantees that the required DLLs are accessible/visible from the class.

6.3 Economic dispatch Sample

Sampe testcase—EconomicDispatchGAMS.java

Result:

```
--------------------------------------------------------------------------------
---
    genP @ Bus-Bus1  : level=1.91 ,  marginal=0.0
    genP @ Bus-Bus2  : level=0.5 ,  marginal=-20.033200000000004
    genP @ Bus-Bus3  : level=0.05 ,  marginal=39.96479999999998
    genP @ Bus-Bus4  : level=0.0 ,  marginal=-30.038200000000003
    genP @ Bus-Bus5  : level=0.0 ,  marginal=-30.038200000000003
    genP @ Bus-Bus6  : level=0.06 ,  marginal=19.968999999999994
    genP @ Bus-Bus7  : level=0.0 ,  marginal=-30.038200000000003
    genP @ Bus-Bus8  : level=0.07 ,  marginal=9.974400000000003
    genP @ Bus-Bus9  : level=0.0 ,  marginal=-30.038200000000003
    genP @ Bus-Bus10  : level=0.0 ,  marginal=-30.038200000000003
    genP @ Bus-Bus11  : level=0.0 ,  marginal=-30.038200000000003
    genP @ Bus-Bus12  : level=0.0 ,  marginal=-30.038200000000003
    genP @ Bus-Bus13  : level=0.0 ,  marginal=-30.038200000000003
    genP @ Bus-Bus14  : level=0.0 ,  marginal=-30.038200000000003

 DC Loadflow Results

   Bud Id       VoltAng(deg)      Gen      Load     ShuntG
============================================================
    Bus1             0.00       191.00      0.00      0.00
```

Bus2	-4.32	50.00	21.70	0.00
Bus3	-11.80	5.00	94.20	0.00
Bus4	-9.52	0.00	47.80	0.00
Bus5	-8.13	0.00	7.60	0.00
Bus6	-13.41	6.00	11.20	0.00
Bus7	-12.32	0.00	0.00	0.00
Bus8	-11.61	7.00	0.00	0.00
Bus9	-14.23	0.00	29.50	0.00
Bus10	-14.51	0.00	9.00	0.00
Bus11	-14.17	0.00	3.50	0.00
Bus12	-14.53	0.00	6.10	0.00
Bus13	-14.70	0.00	13.50	0.00
Bus14	-15.73	0.00	14.90	0.00

FromId->ToId	Power Flow(Mw)	MWLimit	Loading%	Violation
Bus1->Bus2(1)	127.39	0.00		
Bus1->Bus5(1)	63.61	0.00		
Bus2->Bus3(1)	65.95	0.00		
Bus2->Bus4(1)	51.49	0.00		
Bus2->Bus5(1)	38.25	0.00		
Bus3->Bus4(1)	-23.25	0.00		
Bus4->Bus5(1)	-57.66	0.00		
Bus4->Bus7(1)	23.33	0.00		
Bus4->Bus9(1)	14.77	0.00		
Bus5->Bus6(1)	36.60	0.00		
Bus6->Bus11(1)	6.62	0.00		
Bus6->Bus12(1)	7.59	0.00		
Bus6->Bus13(1)	17.19	0.00		
Bus7->Bus8(1)	-7.00	0.00		
Bus7->Bus9(1)	30.33	0.00		
Bus9->Bus10(1)	5.88	0.00		
Bus9->Bus14(1)	9.71	0.00		
Bus10->Bus11(1)	-3.12	0.00		
Bus12->Bus13(1)	1.49	0.00		
Bus13->Bus14(1)	5.19	0.00		

Chapter 7. Sensitivity Analysis and DCLF-based contingency analysis

Chapter 8. New dynamic model development with Controller Modeling Language

Chapter 9. Graph based power system applications

9.1 Network Topology processing

Chapter 10. InterPSS Extension example

Chapter 6 presents a good example of functional extension by integrating InterPSS and GAMS. In this chapter, two more examples will be given in this chapter to help users better realize that InterPSS is designed for extension, and that it can be applied to a broad range of power system applications through model and/or algorithm extension.

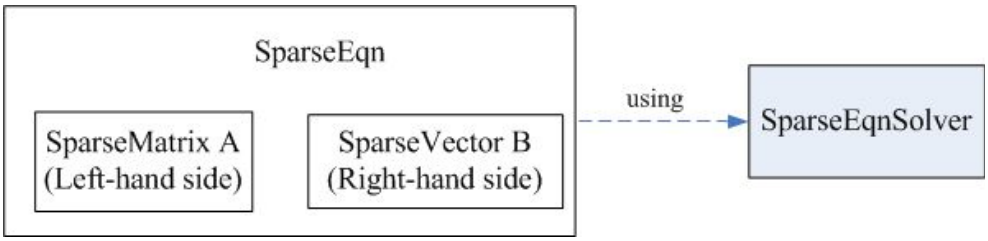
10.1 A DCOPF module by integrating QuadProgJ
10.2 Network topology visualization by integrating JGraphX

Appendix-A Sparse Matrix and Solver

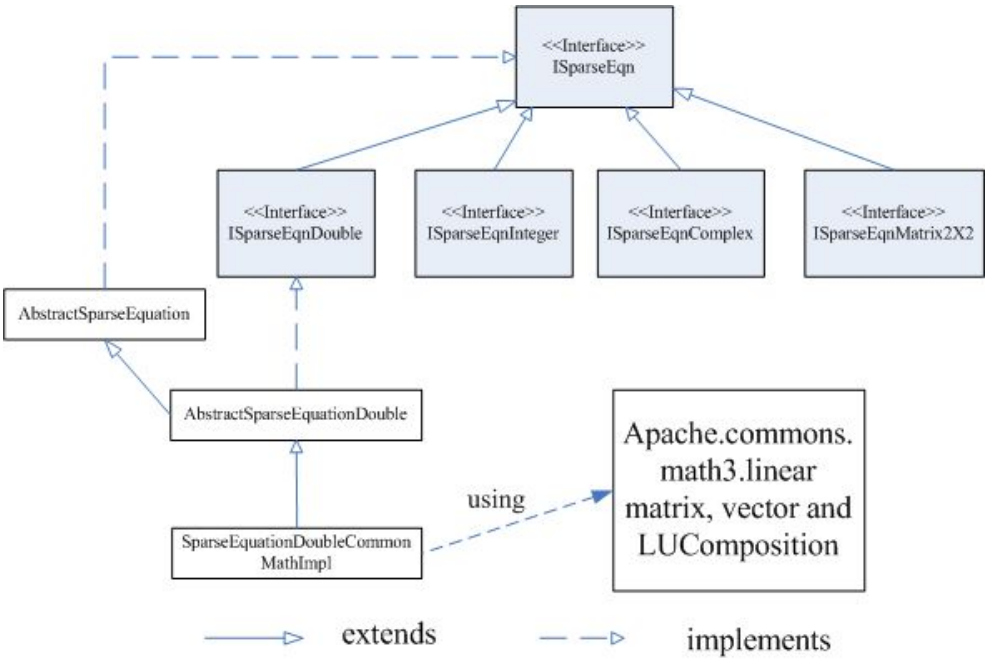
A-1. SparseEqn classes

(1) Overview

Sparse Matrix is the most basic and commonly-used data structure in power system simulations. For example, the Admittance matrix, Jacobian matrix for AC load flow, B' abd B'' matrix for DC load flow are sparse due to the inherent topology structure of power systems. In addition, we observe that power flow, short circuit and transient stability simulation all need solving the sparse Linear Equations involving these sparse matrix, like [Y] [V] = [I], [J][dX] = [dPQ]. Thesefore, we combine the concept of sparse matrix and linear equations together and create the SparseEqn class.



Two main attributes of a SparseEqn is the sparseMatrix A and sparseVector B, which is corresponding to the left hand side and right hand side of linear equation set AX= b.SparseEqn also refers to, or uses, a SparseEqnSolver object, which actually performs matrix manipulation, for example, LU decomposition, and caches the factorization table.



The class hierarchy structure is shown above. ISparseEqn is the basic interface, then for each numeric type, there is a corresponding interface. In particular, for the Jacobian matrix using 2 X 2 block matrix stroing structure, the interface ISparseEqnMatrix2X2 is defined. AbstractSparseEqn is an abstract class for sparseEqn, all implementation should extends it. In the figure above, only the sparseEqn for double type is shown as the concrete and complete implementation example, with the Apache common math 3 linear classes used.

SparseEqn for integer, complex and Matrix2X2 data type are also defined, as their actual implementations are very similar to the double type, they are not shown in the figure above. SparseEqnComplex is mainly used in forming admittance matrix (Y) and impedance matrix (Z), while SparseEqnMatrix2X2 is used to represent the Jacobian matrix (J), with the [dPdAng, dPdV ; dQdAng , dQdV] 2X2 block matrix represented by a Matrix2X2 class object inside.

For the code, please refer to the following link:
https://github.com/InterPSS-Project/ipss-common/tree/master/ipss.numeric/src/org/interpss/numeric/sparse

(2) Basic operation

Method	Description
setDimension	set the matrix dimension
increaseDimension	increase matrix dimension, the incremental part is augmented in the right and bottom side of the original network
addToAij(d, i, j)	A matrix aij = aij0 + d
get/setElem(i)	get and set aii of A matrix
get/setAij	get and set aij of A matrix
setB2Zero	set all entries in the B vector to zero
setB2Unity	set all entries in the B vector to 1
getXi	get the solution result of AX = b, index of i is based on the internal storage. The sortNumber of the bus is used by default.
getXVector	return the solution X in the form of an array
setBVector	set B in the form of an array
setBi	set bi of the B vector
addToBi	used to update the B by bi =bi0 + updateBi
setToZero	set all aij and bi to 0.0

luMatrix	perform lu factorization to the A matrix
solveEqn	solve AX = b

A-2. Sparse Linear equation solver

(1) Solver Interface

As is well known that LU decomposition based method is the basis for almost all existing sparse linear equation solver, therefore, the design of the interface focuses on two methods, i.e., LU decomposition and solving the [A] X = b problem.

```
public interface ISparseEqnSolver {
        /**
         * set the A matrix dirty status to true, due to change of its element
         */
        void setMatrixDirty();

        /**
         * Solve the [A]X = B problem
         *
         */
        void solveEqn() throws IpssNumericException;

        /**
         * LU decomposition of the matrix.
         *
         * @param tolerance the tolerance for matrix singular detection
         * @return if succeed return true.
         */
        boolean luMatrix( final double tolerance) throws IpssNumericException;
}
```

(2) How SparseEqn object uses a solver to solve the equation set : [A]X = b

Recall that SparseEqn object stores the matrix A and vector b. With that, the basic logic of solving [A]X = b is

i) create a new SparseEqn sover instance and then the SparseEqn object refers to it
ii) the solver LU factorize the A matrix
iii) With the factorized result, the solver solve the equation set: [A]X = b

- Example-- `SparseEqnDoubleImpl`

```
public SparseEqnDoubleImpl(int n) {
```

```
        // create a new SparseEqn sover instance and then SparseEqn object refer to it

        this.solver = SparseEqnSolverFactory.createSparseEqnDoubleSolver(this);

}

// the solver LU factorize the A matrix
public boolean luMatrix( final double tolerance) throws IpssNumericException {
            return this.solver.luMatrix(tolerance);
}

/*
* solve the equation set
*/
public void solveEqn() throws IpssNumericException {
            this.solver.solveEqn();
}
```

A-3 Customize the solver

With the rapid development of hardware and software, it is probably that you want to use the latest developed numerical solver to replace the existing default numerical solver of InterPSS. It is quite straight forward to customize the solver, as there are mainly two steps invovled:

1) Develop your own solver or adapting an existing solver by implementing the the solver interface shown in A-2 part (1). Specially, the SparseEqnDoubleCommonMathImpl.java is given out as a customized solver reference.

2) Configure the `com.interpss.core.sparse.SparseEqnSolverFactory` to assign your new solver to the solver creator.

Note: the Java 8 Function interface is used below

```
    // configure the sparse eqn solvers
      SparseEqnSolverFactory.setDoubleSolverCreator(
          (ISparseEqnDouble eqn) -> new CustomDoubleSparseEqnSolver(eqn));


      SparseEqnSolverFactory.setComplexSolverCreator(
        (ISparseEqnComplex eqn) -> new CustomComplexSparseEqnSolver(eqn));
```

Appendix-B Open Data Model for data import/output

Many years ago, IEEE recommended a Common Data Format (IEEE CDF) for exchanging Load flow study data[1] using flat file. Power system software companies uses their own internal data format, some have features similar to a mark-up language. However, it is our observation that these formats are proprietary and are often not well documented. We think that the power engineering community needs a completely open, free, flexible and well-documented model/format for power system analysis information exchange. XML is an obvious choice, since it is a very mature technology and has become the de facto standard for defining information exchange standards. Also, numerous open-source and free XML processing tools are currently available. For those who are unfamiliar with XML, some introductory information relevant to power system representation can be found in Ref[2]. (E.Z.Zhou, “XML and Data Exchange", IEEE Power Engineering Review, April 2000)

1. Prerequisite

1.1 Basic understanding of XML: schema, data binding and JAXB

Extensible Markup Language (**XML**) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.
For more detailed of XML, please see http://en.wikipedia.org/wiki/XML

- XML Schema

http://www.w3schools.com/schema/

- Xml data binding and JAXB
    - What is XML binding

XML data binding is the process of representing the information in an XML document as an object in computer memory (deserialization). With XML data binding, applications access XML data direct from the object instead of using the Document Object Model (DOM) to retrieve it from the XML file. Using XML binding, you can integrate XML data into a business rule application.

    - Introduction to JAXB

http://docs.oracle.com/javase/tutorial/jaxb/intro/

    - tutorial

http://www.javacodegeeks.com/2013/02/jaxb-tutorial-getting-started.html

1.2 Basic knowledge of data for power system simulation

At least some basic understanding of load flow data is required for understanding the modeling in ODM, a good reference for this is the paper "Common Data Format for the Exchange of Solved Load Flow Data", Working Group on a Common Format for the Exchange of Solved Load Flow Data, IEEE Transactions on Power Apparatus and Systems, Vol. PAS-92, No. 6,
November/December 1973, pp. 1916-1925

2. ODM in a nutshell

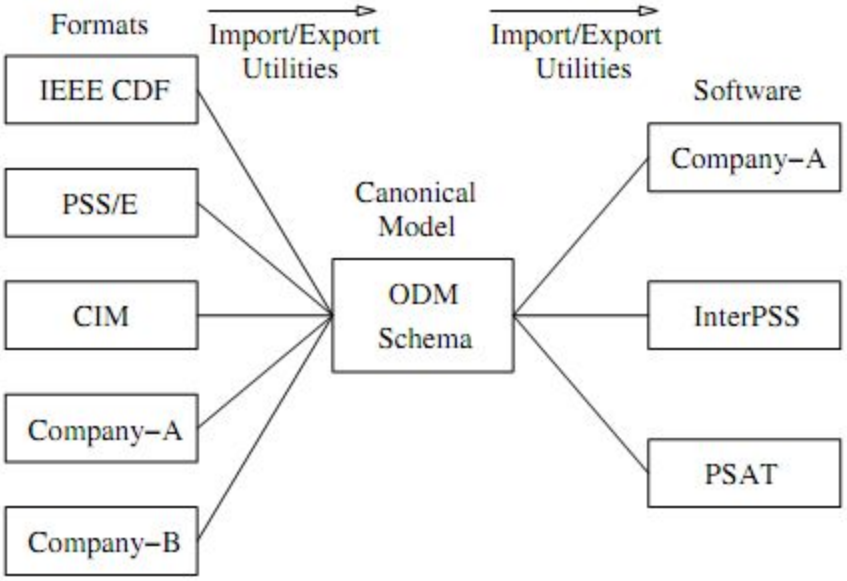
Some useful on-line resources:

- IEEE PES GM 2009 paper: Open Model For Exchanging Power System Data
- Introduction to ODM @ www.InterPSS.org
  https://sites.google.com/a/interpss.org/interpss/Home/ieee-pes-oss

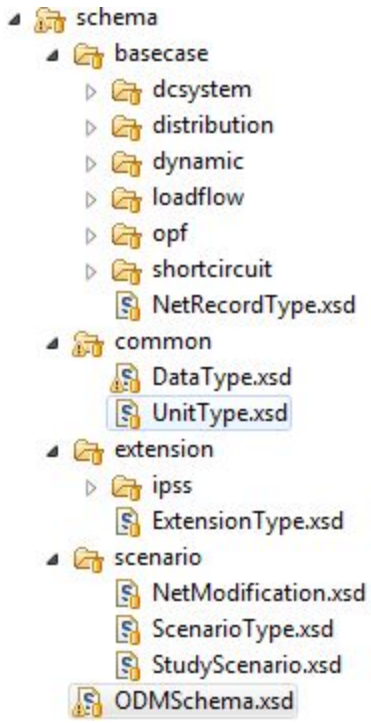
- ODM project on GitHub : https://github.com/InterPSS-Project/ipss-odm

Note: If you want to work on the adapter development based on ODM, please check out this project to your computer, by following the approach discussed in Chapter 1

2.1 ODM as a data-format free intermediary for data exchange

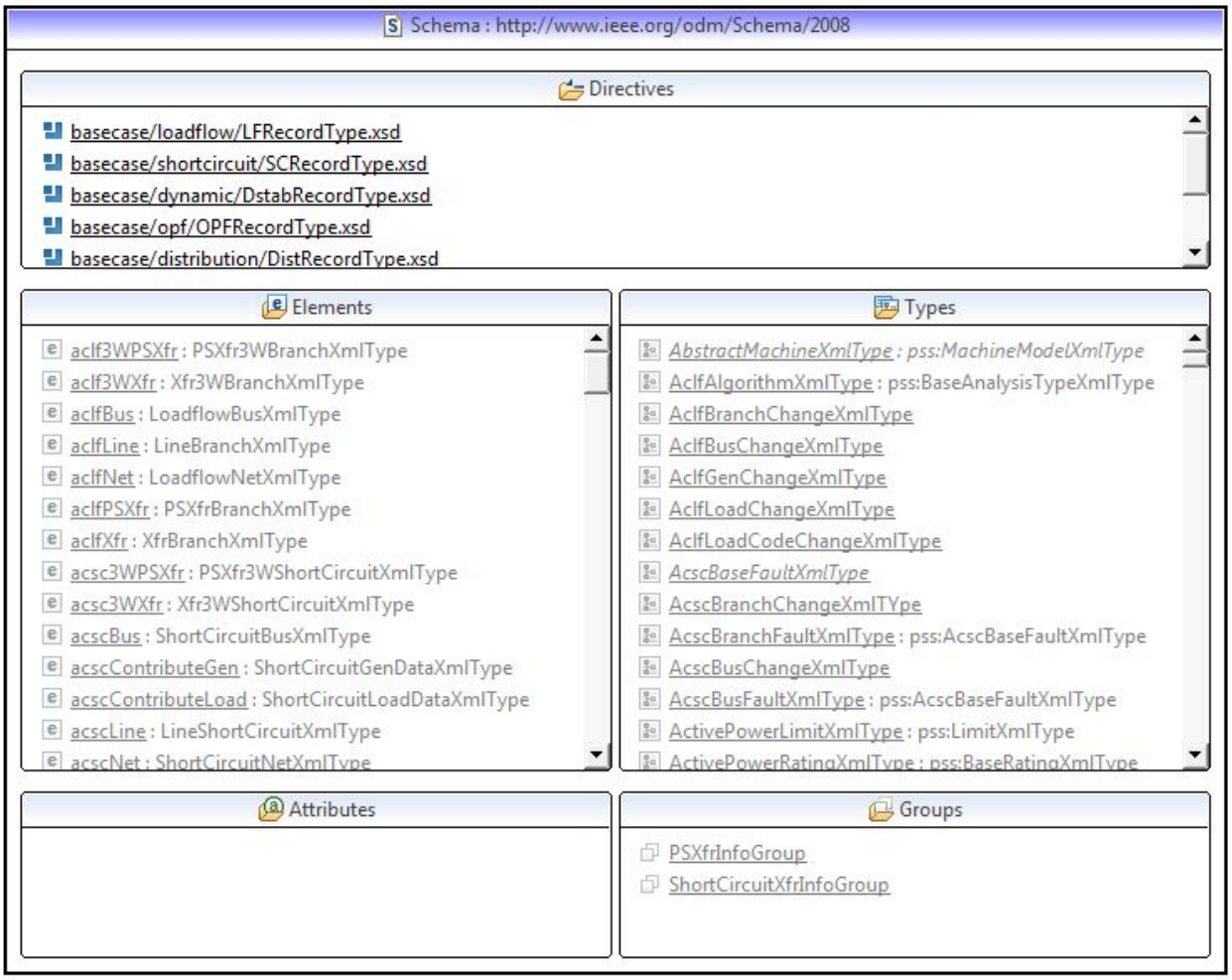


2.2 XML Schema for power system simulation data modeling



From the basecase folder, one can easily find the analysis categories of the ODM schema-- it covers a wide range of data modeling from the conventional loadflow, short circuit, dynamic (transient stability) to OPF and distribution network analysis. Data types and unit types are defined under the **common** folder.

ODMSchema.xsd is the main schema definition file. Eclipse XML Schema Editor view of the ODM schema (ODMSchema.xsd) is shown below:



NOTE: *We think it is better to info users/developers, before any further introduction, that you are not required to understand the schema completely in order to modify the ODM model or develop a custom adapter. In fact, some common and useful utilities have been developed to help user to develop customized ODM adapter to import/output data through ODM. You are encouraged to continue to the adapter implementation examples in Section 2.4 after you have some basic understanding of ODM schema. You will find it is easier to use it than expected!*

2.2.1 Basic Schema

Naming Convention

There is two main Xml data structures used in ODM, in addition to standard data types defined in Xml schema.

- Complex Type : <...>XmlType, is for defining complex data structure with different types of attributes and elements, for example, PSSNetworkXmlType. Complex type always ends "XmlType".
- Simple Type : <...>EnumType of String is for defining enumeration of simple tokens, for example, LFGenCodeEnumType has [Swing, PV, PQ]. Simple type of string always ends with "EnumType".

- Unit Type : <...>UnitType is special simple type for defining unit. for example, ApparentPowerUnitType has [Mva, Kva ...].

Name Space

The ODM Schema target namespace is "http://www.ieee.org/odm/Schema/2008". All types defined in the ODM Schema are "qualified" with a namespace prefix pss.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.ieee.org/odm/Schema/2008"
  xmlns:pss="http://www.ieee.org/odm/Schema/2008"
  elementFormDefault="qualified">
    ...
    <element name="PSSStudyCase" type="pss:StudyCaseXmlType"></element>
    ...
</schema>
```

Please see Namespaces in XML for more in depth discussion about namespace.

Version Number

The Schema root element type StudyCaseXmlType has a attribute schemaVersion for indicating schema version number.

PU System

Actual units, such as KV, MVA, and PU could be used in the ODM Model. All PU values in the model are based on the base case base Kva and bus base voltage, unless otherwise specified.

Extension

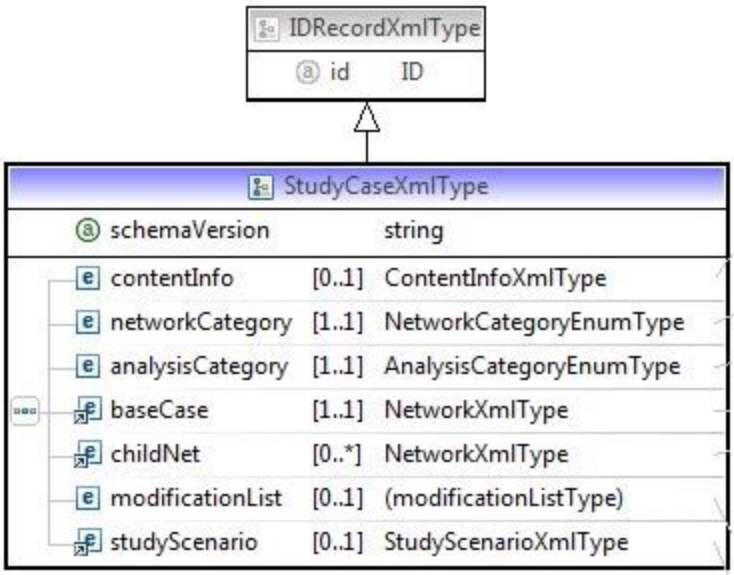
ODM is designed to be extended. The extension folder is for ODM extension by any interested party. Currently, it has a folder containing extension by InterPSS. The following namespace has been introduced in InterPSS extension.

xmlns:ipss="http://www.interpss.org/Schema/odm/2008"

Schema Root Element

The ODM schema root element **PSSStudyCase** is of type StudyCaseXmlType.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema" ...>
  <element name="PSSStudyCase" type="pss:StudyCaseXmlType"></element>
  ...
</shema>
```



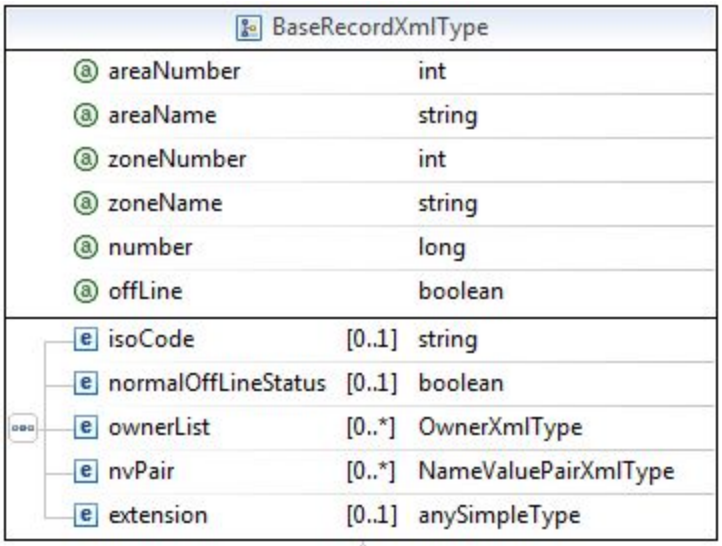
The **StudyCaseXmlType** has the following elements:

- schemaVersion - version number
- originalFormat - *[IEEE-CDF* | *PSS-E* | *UCTE-DEF* | *InterPSS*| *PSAT* | *PowerWorld* | *BPA*| *PSLF* | *Custom]* original data format.
- adapterProviderName - optional, data transformation adapter provider name
- adapterProviderVersion - optional, data transformation adapter provider version
- analysisCategory - *[Loadflow* | *ShortCircuit* | *TransientStability]* analysis type category. This could be expended to include more types in the future schema version
- networkCategory - [Transmission |Distribution] network type category.
- baseCase - element of type PSSNetworkXmlType for describing a power network as the base case.
- modificationList - optional, a list of modification to the base case
- scenarioList - optional, a list of scenarios, built on the base case.

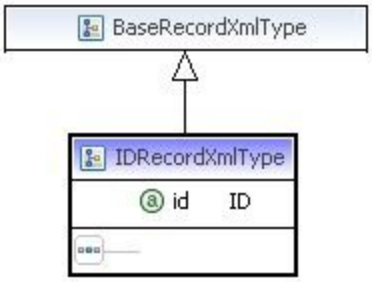
Base Record

The information are organized in the Schema using the record concept.

- name - Optional, element name
- desc - Optional, element description
- isoCode - Optional, element ISO code
- offLine - Optional, element actual off-line status
- normalOffLineStatus - Optional, normal element off-line status
- number - Optional, element number
- area - Area number. It is optional and should be a non-zero number.
- zone - Zone number. It is optional and should be a non-zero number.
- ownerList - Optional, owner list of the element
- nvPairList - Optional, name value pair for those elements not defined in the schema
- extension - Optional, extension point. One can add an element of Any type (any schema type) to extend the schema.



ID Record

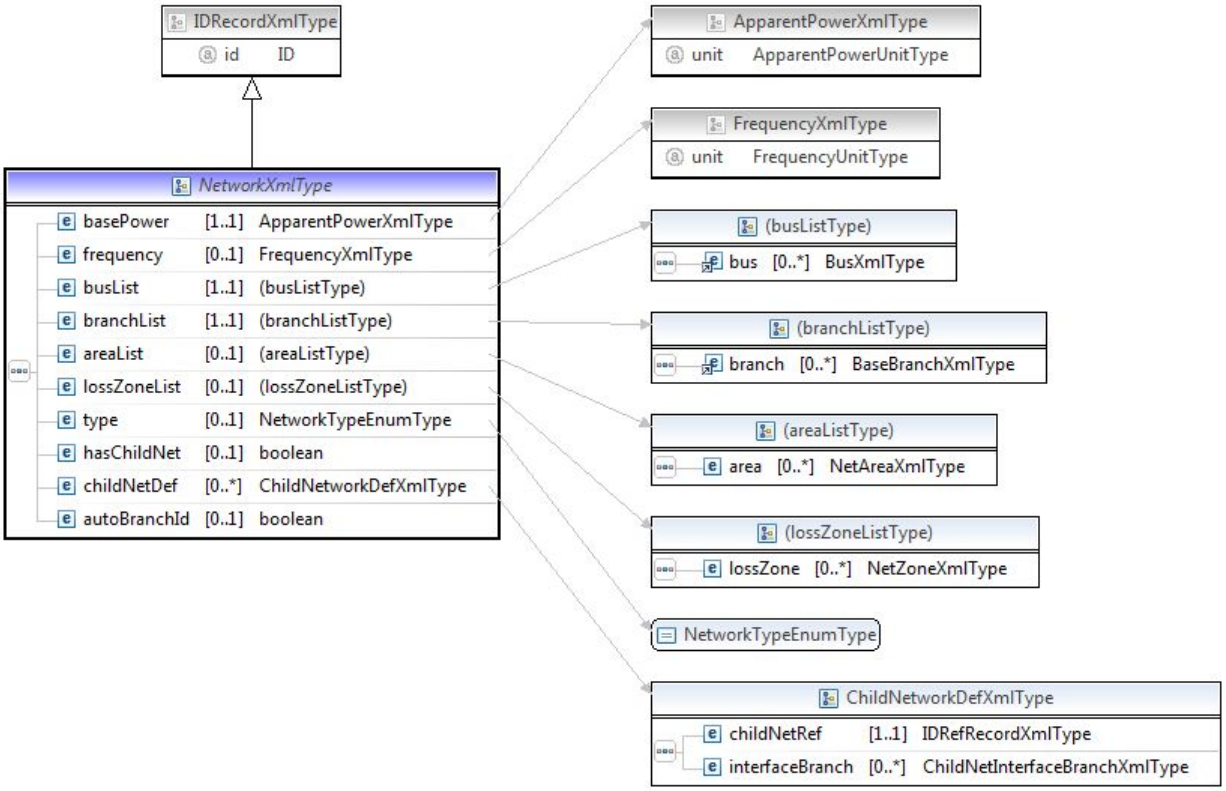


A ID record has an id of Type xsd:ID, which guarantees its uniqueness in an ODM xml file. It is the parent of all searchable records, such BusRecord, BranchRecord.

2.2.2 Base Case

The base case element is of type **PSSNetworkXmlType**. It is intend to describe a power network for simulation purpose. Currently, it our focus is on Loadflow study information. The structure has been designed in such a way that it could be extended to include more simulation information, such as short circuit, transient stability.

- baseKva - Base kva for the PU system.
- baseKvaUnit - [KVA | MVA] base kav unit
- busList - network bus record list
- branchList - network branch record list
- loseZoneList - lose zone list per IEEE CDF. It may be extended to cover other formats
- interchangeList - interchange list per IEEE CDF. It may be extended to cover other formats
- tieLineList - tie line list per IEEE CDF. It may be extended to cover other formats



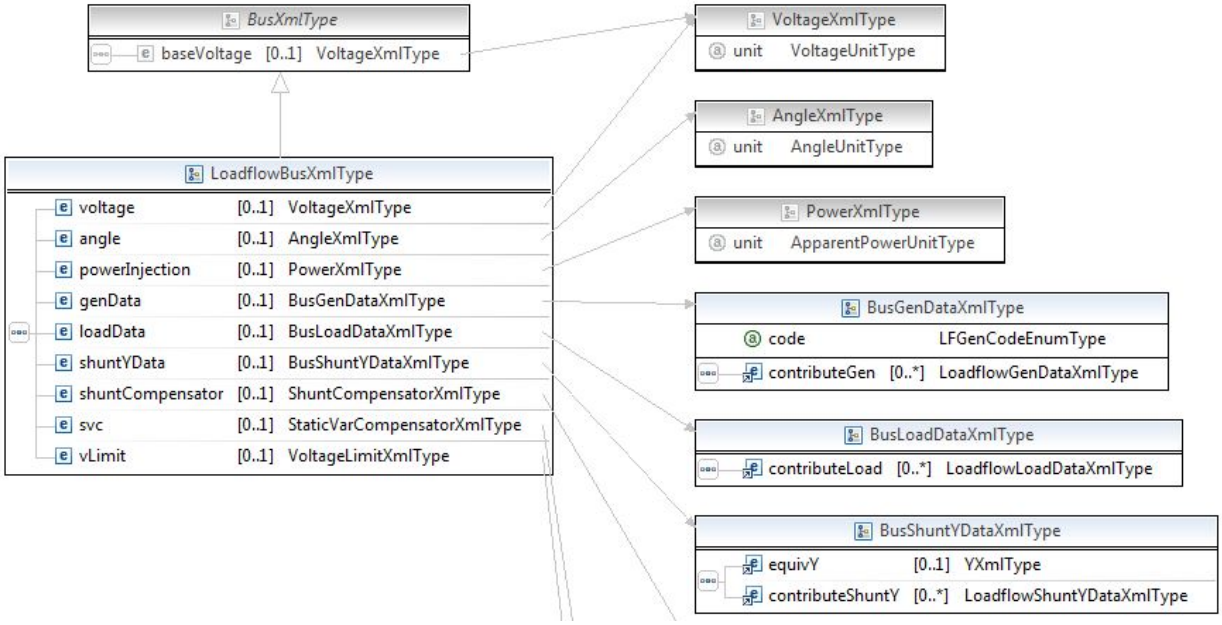
2.2.3 Bus Record

Bus record is basically defined by following a similar class hierarchy of the Bus in InterPSS core with the following structure:

ID Record<--CimRdf<- Bus<--Load flow Bus<-- Short circuit Bus <-- Dynamic Bus

The basic Bus record is is **BusXmlType**

2.2.3.1 Bus Record for AC Load flow



Bus Loadflow data is described by the **LoadflowBusDataXmlType**.

- baseVoltage - Bus base voltage
- baseVoltageUnit - [VOLT | KV] Bus base voltage unit
- voltage - Bus voltage
- angle - Bus angle
- genData - Bus generation data, see more description below
- loadData - Bus load data, see more description below
- shuntY - Bus shunt Y
- powerInjection - power injection into the network, for store Loadflow results

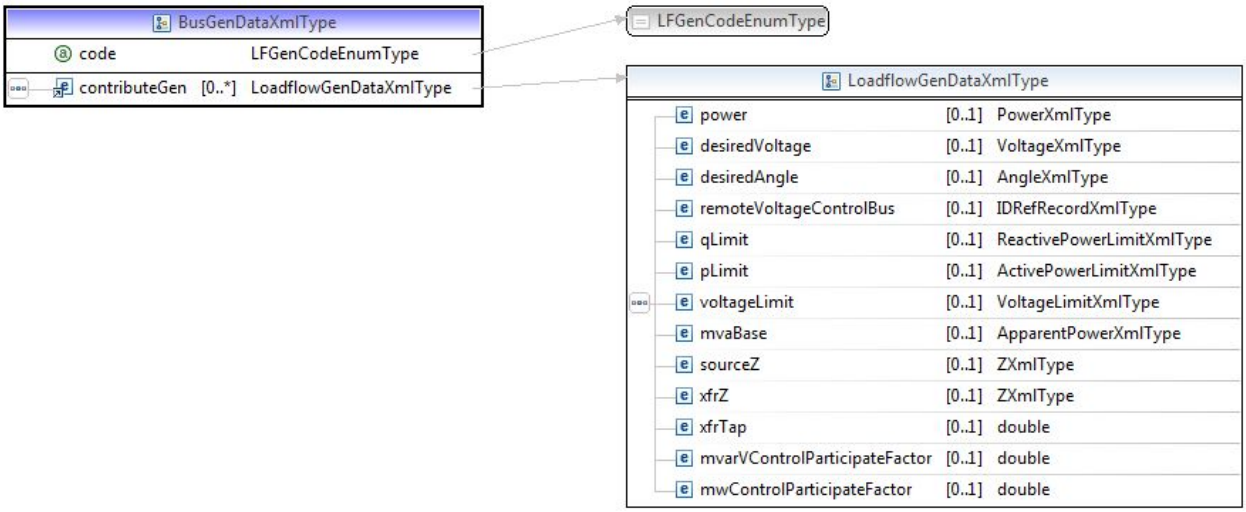
Bus Generation Data -- BusGenDataXmlType

- code - [PQ, PV, SWING]
- contributeGen list: a list for storing the records of generator(s) connecting to the bus. Thus, the scenario of multiple generators connected to the same bus is supported.

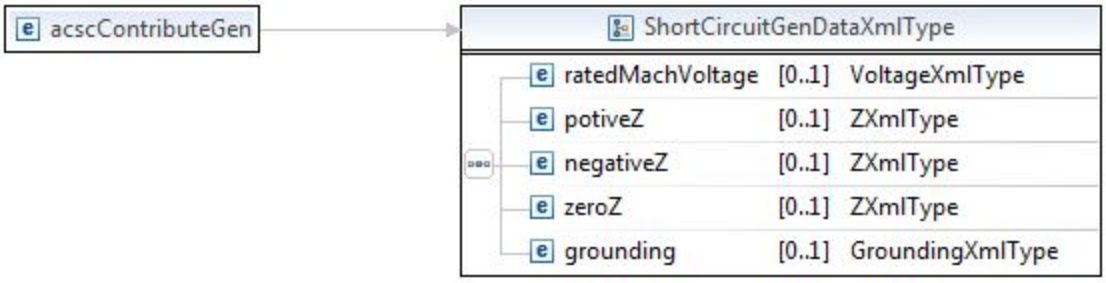
The record of a generator for load flow includes:

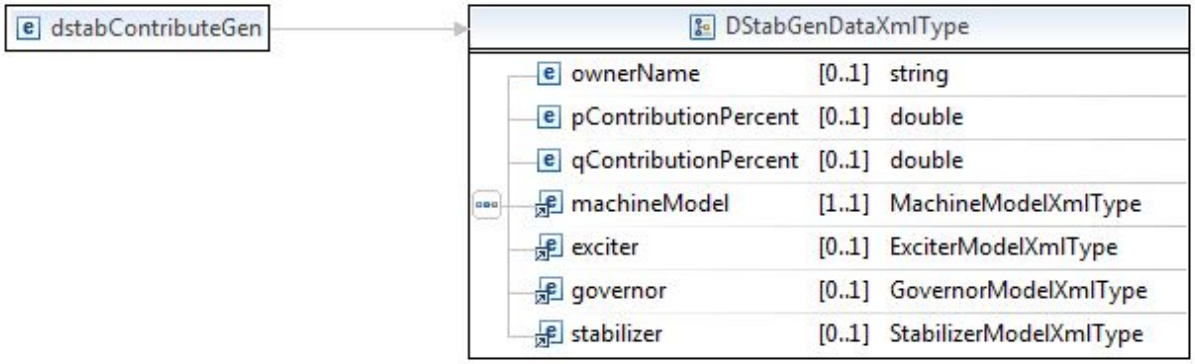
- power - generation, p, q plus a unit [PU | KVA | MVA] (required)
- desiredVoltage/Angle- scheduled voltage magnitude/ angle (required, if not provided at the bus level)
- pLimit - generator real power limit (optional)
- qLimit - generator reactive power limit, used for PV bus (required)
- vLimit - generator voltage limit (optional)
- mvaBase - generator MVA Base (required)
- remoteVoltageControlBus - describing desired remote bus when the generator Q is used to control the remote bus voltage. (optional)

- XfrZ - step up transformer impedance (optional)
- XfrTap - step up transformer tap ,defined at the high voltage side (optional)



The acscContributeGen and dstabContributeGenmodels for short circuit and the transient stability analysis, respectively:





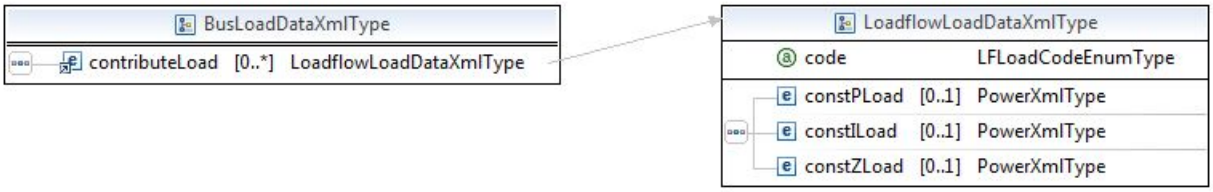
Machine, exciter, turbine and governor as well as stabilizer (PSS), if any, are linked to a dstabContibuteGen model.

Bus Load Data

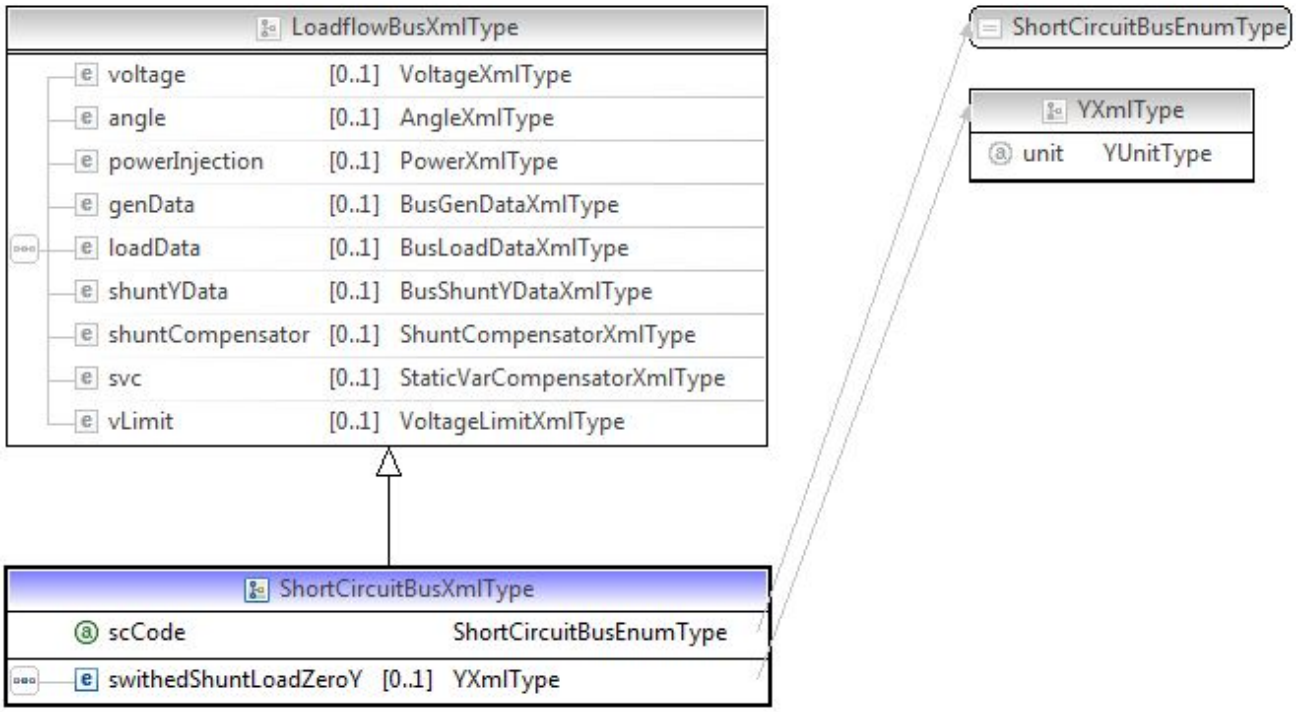
- code - [CONST_P, CONST_I, CONST_Z] constant power, constant current or constant z load.
- contributeload list - a list for storing all records of loads connected to the bus
- Modeled by **BusLoadDataXmlType**

A contributing load record

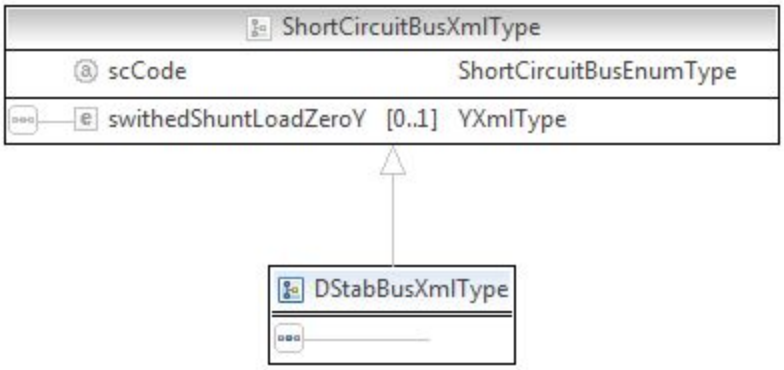
- Modeled by **LoadflowLoadDataXmlType**
- There are three components for each load record, i.e., constant power, constant current and constant impedance component. A contributing load can be a combination of these components.



2.2.3.2 Bus Record for AC short circuit



2.2.3.3 Bus Record for transient stability



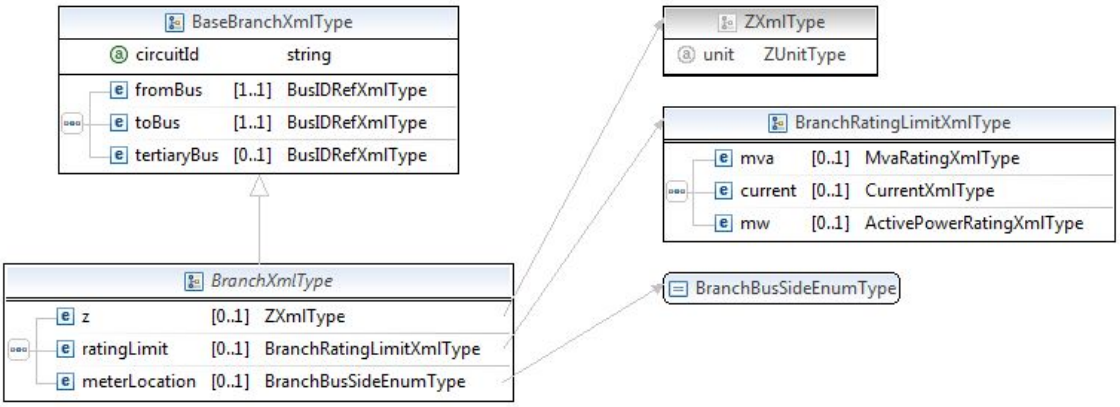
2.2.4 Branch Record

Branch record has an unique id, an id reference to a from bus record, an id reference to a to bus record, a circuit id. The tereiayBus is optional, which could be used to define a 3-winding transformer. It also holds branch related simulation data.

As a general practice, Line and 2-winding transformer are modeled separately, but they share some common information, e.g., terminals are defined by two buses. The class inheritance structure is as follows:

BaseBranch<--Branch<--LineBranch (conventional line/cable)
\
\ <-- XfrBranch (Xfr stands for 2-winding transformer)

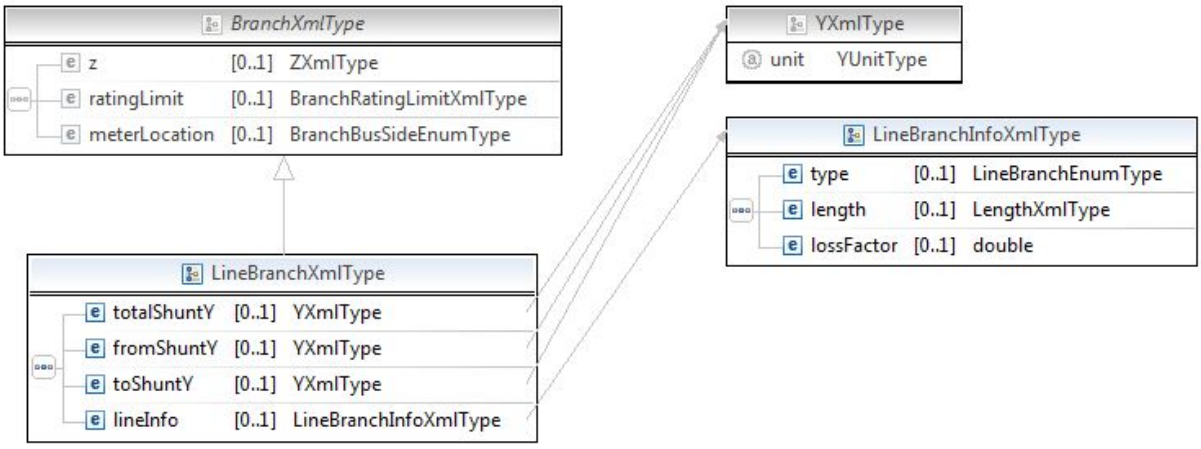
Base Branch Data



Loadflow Line Data

- z - line impedance, r, x, plus a unit [PU | OHM]
- totalShuntY - total line charging shunt Y, g, b, plus a unit [PU | MHO | MICROMHO]
- fromShuntY - extra shunt Y at the from bus end of the line

- toShuntY - extra shunt Y at the to bus end of the line
- ratingLimit - thermal or contingency rating limit
- LineInfo is optional

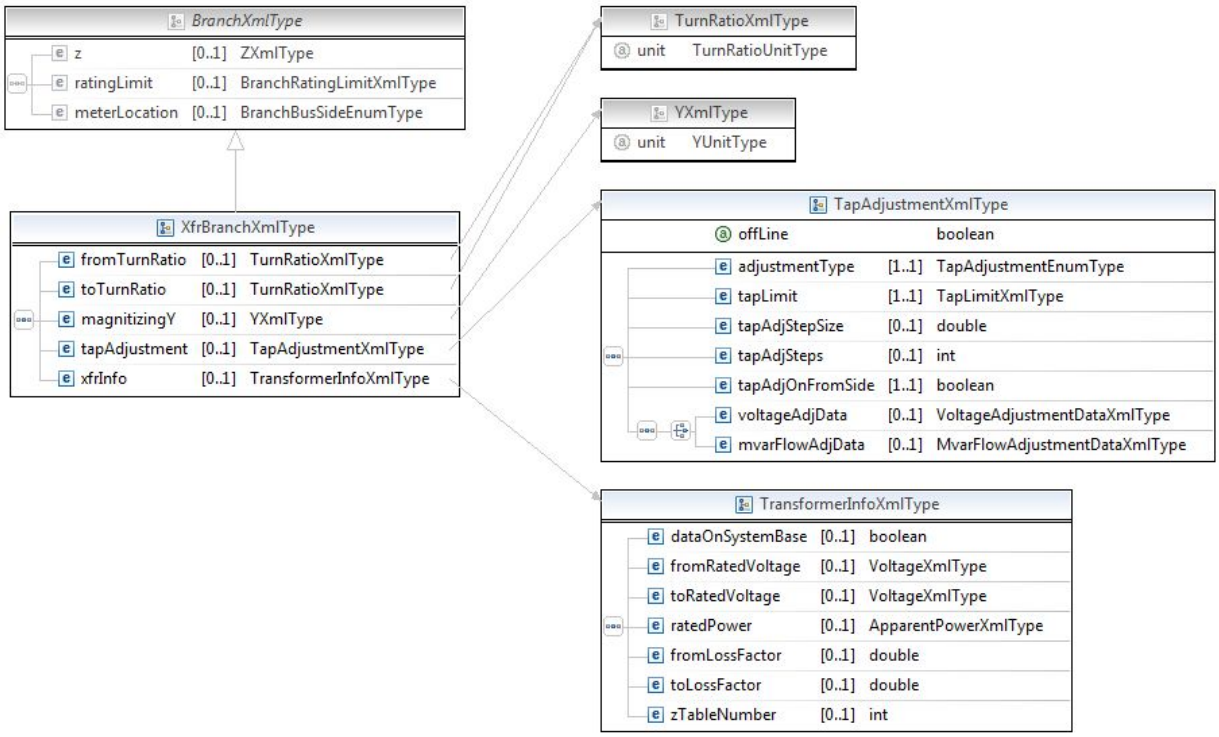


Loadflow Transformer Data

By analysis different data formats, we believe the above transformer model captured all possible permutations of a local transformer or a logical phase-shift transformer model for loadflow analysis purpose. Adjustment could be applied to transformer tap to adjust bus voltage or Mvar flow, or to phase-shifting transformer angle to adjust Mw flow.

Transformer Data

- z - transformer impedance, r, x, plus a unit [PU | Ohm].
- ratingData - transformer rating data. Voltage unit should be VOLT or KV and the same for both side.
- fromTurnRatio - PU based on the transformer rated voltage
- toTurnRatio - PU based on the transformer rated voltage
- tapAdjustment - transformer tap adjustment info, see more below.



In the transformer data schema definition, only transformer impedance z are required elements. The fromTurnRation and toTurnRatio should be set to 1.0 if not defined. The other optional elements should be set to zero if not specified.

Please Note: When transformer Z and Y are specified in actual value, in Ohm or Mho, the side where the value is measured need to be specified. ODM assumes that the value is measured at the *high voltage side*.

Transformer Tap Adjustment

- adjustmentType - [Voltage | MvarFlow] tap adjustment type, voltage adjustment or mvar flow adjustment
- tapLimit - tap limit
- tapLimitUnit - [PU | PERCENT] tap limit unit
- tapAdjStepSize - tap adjustment step size. If 0.0 or not defined, assume continuous adjustment
- tapAdjOnFromSide - Turn ratio at from side or to side could be adjusted.
- voltageAdjData - voltage adjustment data, see more below
- mvarFlowAdjData - mvar flow adjustment data, see more below.

Transformer Tap Adjustment for Bus Voltage

- mode - [ValueAdjustment | RangeAdjustment] adjustment could be based on a desired value or desired range.
- desiredValue - desiredValue for desired bus voltage for the ValueAdjustment
- desiredRange - desiredRange for desired bus voltage range for the RangeAdjustment
- desiredVoltageUnit - desired voltage unit [PU | Volt | KV]
- adjVoltageBus - adjustment voltage
- adjBusLocation - [TerminalBus | NearFromBus | NearToBus | FromBus | ToBus] adjustment bus location

Transformer Tap Adjustment for MVar Flow

- mode - [ValueAdjustment | RangeAdjustment] adjustment could be based on a desired value or desired range.
- desiredValue - desiredValue for desired mvar flow for the ValueAdjustment
- desiredRange - desiredRange for desired mvar flow range for the RangeAdjustment
- desiredMvarFlowUnit - desired mvar unit [PU | KVAR | MVAR]
- mvarMeasuredOnFromSide - describing mvar measuring location, from bus side or to bus side

Loadflow PhaseShift Transformer Data

PhaseShift transformer inherits from transformer. In addition it has the following fields

- fromAngle - from bus side angle
- toAngle - to bus side angle
- angleAdjustment - phase shifting angle adjustment to control MW flow.

Phase Angle Adjustment

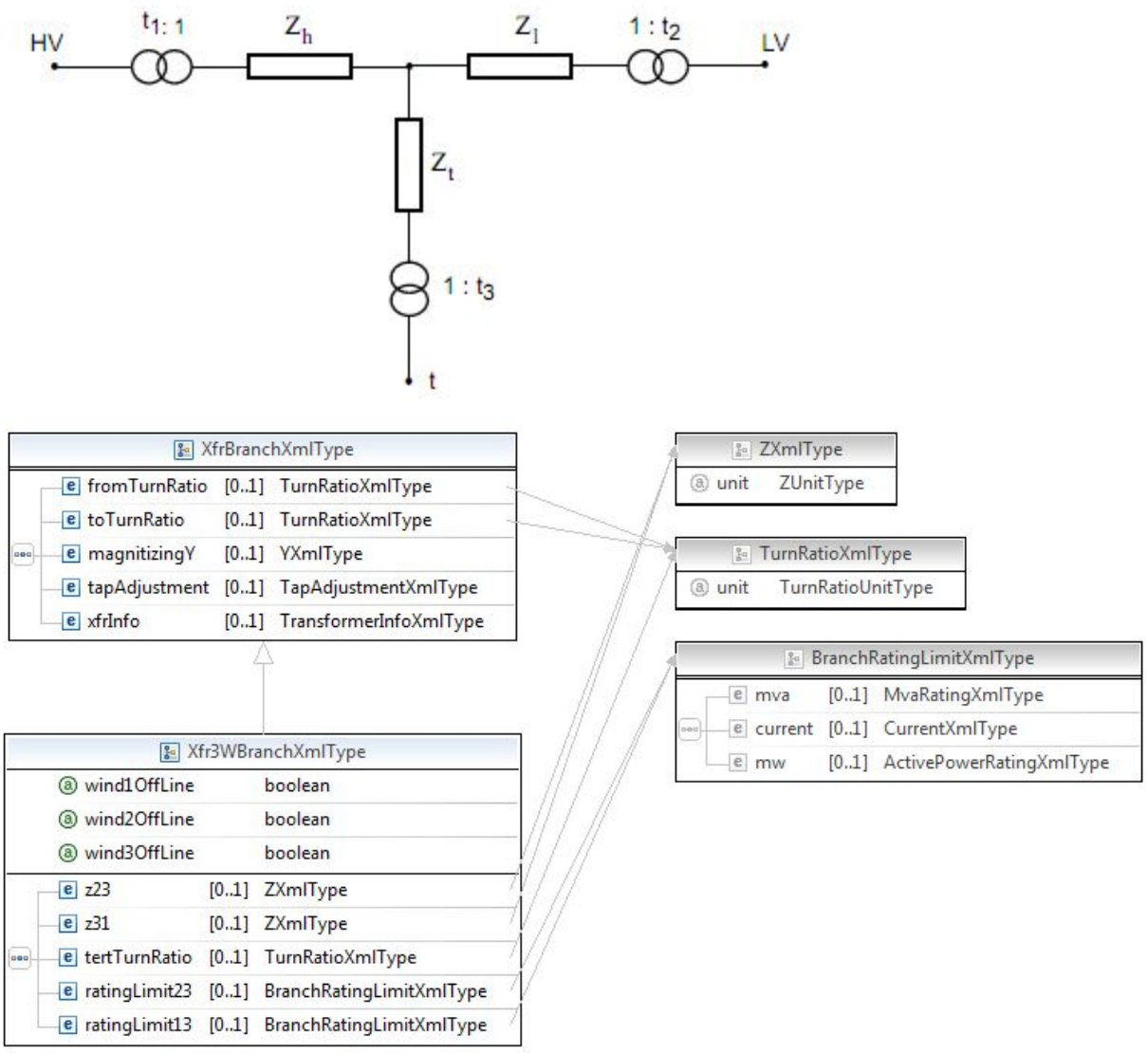
- mode - [ValueAdjustment | RangeAdjustment] adjustment could be based on a desired value or desired range.
- desiredValue - desiredValue for desired MW flow for the ValueAdjustment
- desiredRange - desiredRange for desired MW flow range for the RangeAdjustment
- angleDegLimit - phase shifting angle limit in degrees
- desiredPowerUnit - [PU | KW | MW] desired MW flow unit
- desiredMeasuredOnFromSide - desired MW flow measured on from bus or to bus side

Branch Rating Limit

Three mva rating limits and a current rating limit could be defined

- mvaRating1 - branch mva rating limit
- mvaRating2 - branch mva rating limit
- mvaRating3 - branch mva rating limit
- currentRating - branch current rating limit
- mvaRatingUnit - [PU | KVA | MVA] branch mva rating limit unit
- currentRatingUnit - [PU | Amp | KA] current rating limit unit

3-Winding transformer

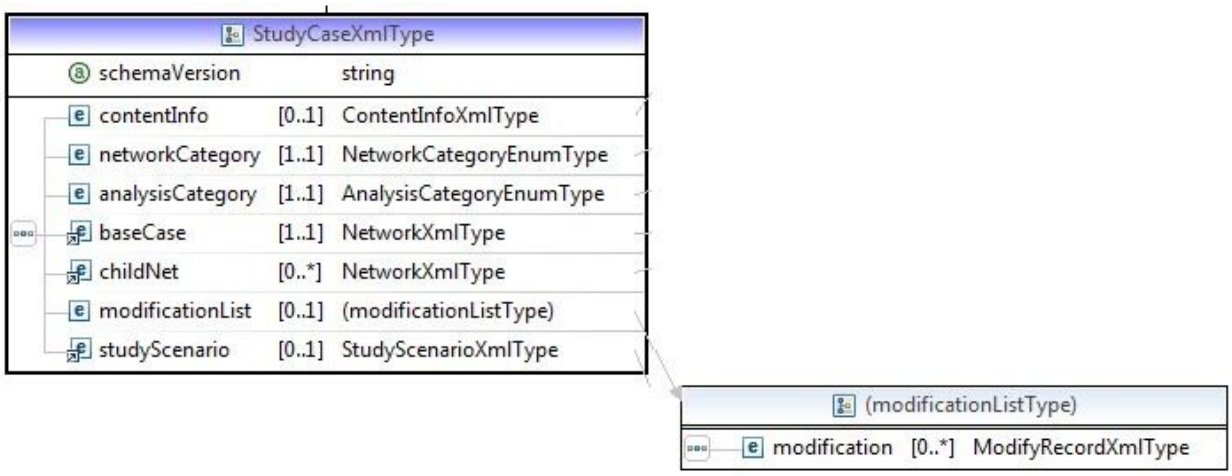


As a general modeling practice, the impedance of the three-widing transformer is determined by the following procedure: one winding is short circuited and one is left open circuited, while a voltage is applied to the remaining winding. This test yields the magnitudes of the three leakage impedances, ZLH, ZLT, and ZHT. The impedance, ZLH, is the sum of low- and high-voltage winding leakage impedances when the tertiary winding is open

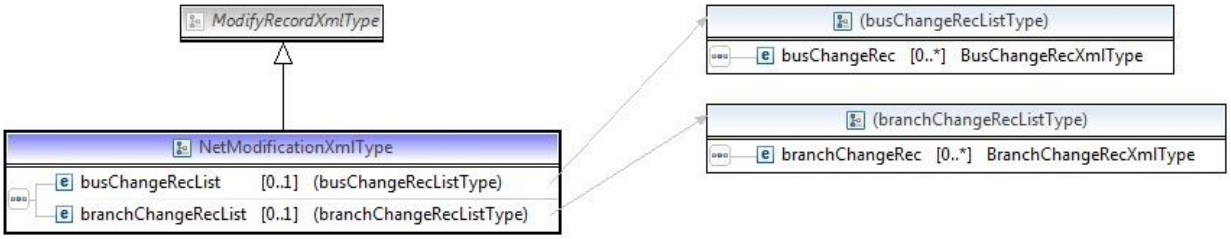
In the ODM, the default branch Z = Z12 = ZLH, Z23 =ZLT, Z31 = ZHT

Modification

A list of modifications could be defined in a StudyCase, as shown in the following figure:



The modification element is of type ModifyRecordXmlType, which is abstract. In actual application, it needs to be replaced with a concrete Xml type, for example, the NetModificationXmlType, in the following figure:



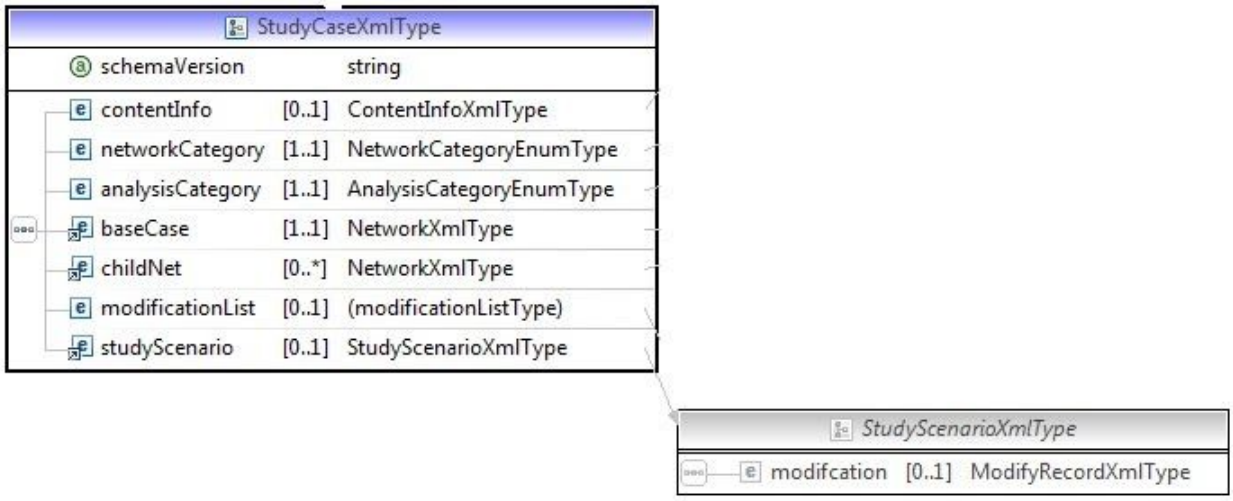
Contingency

In contingency analysis, there is a need to define a list of contingencies which might be applied to the base case. Here we present one possible ways to define contingencies.

- A contingency could be described as a set of branch offline status changes;
- A modification of type NetModificaitonXmlType could be defined to represent a contingency;
- A StudyCase XML file could contain a list of modifications for contingency analysis

Study Scenario

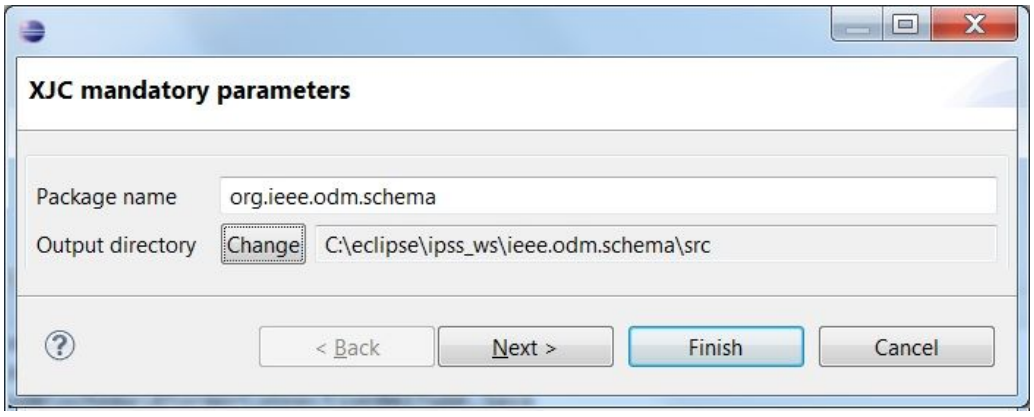
A studyScenario could be defined in a StudyCase, as shown in the following figure:



2.3 Data binding with JAXB

JAXB is used to compile ODM schema to a set of Java classes. Set up the JAXB plugin with the following procedure:
1) Download JAXB Eclipse plugin from Here and unzip;
2) Copy the plugin into <Eclipse install>/Plugins and restart Eclipse

When you check out the **ieee.odm.schema** project from the repo, it is empty. There is no source code inside. It is for hosting compiled ODM schema Java code. After the check-out, you need to compile the schema by right-click the **ODMSchema.xsd** file, select JABX 2.1-> XJC, and set parameters as follows: output directory should be pointed to your local src folder of the ieee.odm.schema project.



2.4 Data import to ODM/XML

Due to some legacy issue, power system simulation data is usually defined in a text file. Most of the existing data formats follow the following convention: the same type of data records are usually grouped into one data section, while each data record is usually defined in a single line, some models in a few

data format defined with multiple lines. The following table present the survey result of the most common data formats:

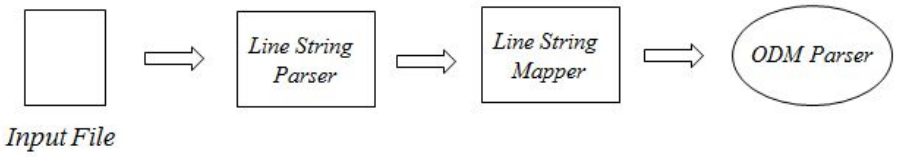
TABLE I
FEATURES OF MOST COMMON DATA FORMATS FOR POWER SYSTEM ANALYSIS

Format Name	Data Position	Data Order	Dynamic Data	Market Data	Short Circuit Data	Graphic Data	Custom Data	Number of Files	Default Values	Modification Command
CEPEL	Fixed	Fixed	No	No	No	No	No	Unique	No	No
CYME	Fixed	Fixed	Yes	No	Yes	No	No	Multiple, Fixed	Yes	No
DigSilent	Free	Free	Yes	No	Yes	Yes	No	Unique	Prototypes	No
EPRI/BPA	Fixed	Fixed	Yes	No	Yes	No	No	Unique	Yes	No
Eurostag	Fixed	Fixed	Yes	No	Yes	No	No	Multiple, Fixed	Yes	No
FlowDemo.net	Free	Fixed	No	No	No	Yes	No	Unique	No	No
GE-PSLF	Free	Fixed	No	No	No	No	No	Unique	Yes	No
IEEE CDF	Fixed	Fixed	No	No	No	No	No	Unique	No	No
INPTC1	Fixed	Fixed	No	No	No	No	No	Multiple, Fixed	No	Yes
MatPower	Free	Free	No	Yes	No	No	No	Any	No	No
Neplan	Free	Free	Yes	No	Yes	Yes	Yes	Multiple, Fixed	Yes	No
PowerWorld	Free	Free	No	Yes	Yes	Yes	No	Unique	Yes	No
PSAT	Free	Free	Yes	Yes	No	Yes	Yes	Any	No	Yes
PSS/E	Free	Fixed	Yes	No	Yes	No	Yes	Unique	Yes	No
PST	Free	Free	Yes	No	Yes	No	Yes	Any	No	Yes
Simpow	Free	Free	Yes	No	Yes	Yes	Yes	Any	Yes	Yes
UCTE	Fixed	Fixed	No	No	No	No	No	Unique	Yes	No

* source: Milano, F., M. Zhou, and GuanJi Hou. "Open model for exchanging power system data." *Power & Energy Society General Meeting, 2009. PES'09. IEEE*. IEEE, 2009.

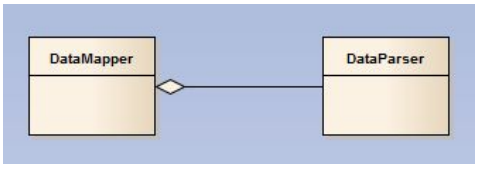
2.4.1 Model data parser and mapper

At high level, ODM data flow process of an ODM adapter is shown in the following diagram:



- Input text file is processed line-by-line. For each input data line string, a corresponding data line string parser is invoked to parse the data;
- After the input line string is parsed, a corresponding data line string mapper is invoked to map the data store in input line string parser to an ODM Parser object, which stores input file info in the ODM format.

For each data segment in the input file, a corresponding DataMapper and DataParser need to be implemented. And the mapper is called inside the parser to parse the input line string.



2.4.1.1 Input Line String Parser

Since the data record is usually defined in a single line or multiple lines, a general data parser

BaseInputLineStringParser has been defined, which defines the method to facilitate the text-based data processing. Further, the AbstractDataFieldParser class provides a higher level abstraction for data parser. Two major methods to be implemented/customized for each data adapter are : **getMetadata()** and **parseFields(String)**

In the case of position free data file, such as PowerWorld, input file has the following data structure.

```
A, B, C, D, A:1, E, F              <- Metadata
1.0, 2.0, 3.0, 4.0, 5.0, 6.0       <- Actual data

1.0, 2.0, 3.0                      <- Data could be in multi-line
4.0, 5.0, 6.0
```

The following are some sample code to parse the input data line string.

```
// assume we have a data parser class defined
parser = new Parser();

// first parse the metadata line and cache the info
parser.parseMetaData("A, B, C, D, E, F");

// parse a data line
parser.parseData("1.0, 2.0, 3.0, 4.0, 5.0");
double a = parser.getDouble("A");
if (parser.exist("A:1")
   a = parser.getDouble("A:1");
```

In case of fix position input data file, such as IEEE CDF, PSS/E, there is no explicit meta data defined, use the **getMetadata()** to define metadata for the input line string parsing.

```
@Override public String[] getMetadata() {
    return new String[] {
       //  0---------------1---------------2---------------3---------------4
         "BusNumber",  "BusName",       "Area",          "Zone",        "Type",
       //  5               6               7               8              9
         "VMag",        "VAng",         "LoadP",         "LoadQ",       "GenP",
       //  10              11              12              13             14
         "GenQ",       "BaseKV",        "DesiredV",     "MaxVarVolt",
"MinVarVolt",
       //  15              16              17
         "ShuntG",    "ShuntB",         "RemoteBusNumber"
    };
}
```

Please Note: because of using the line string parser, we can use field name,e.g., "BusNumber", to access the input data field, instead of using field position. This decouples the mapper from the underlying input data structure.

Parse input line String

Syntax: `dataParser.parseFields(str);`

This method needs to be implemented and overrided to reflect how the data is defined in the input line string

2.4.2 Implement a specific data parser and mapper

2.4.2.1 Implement input line string parser

The major work of implementing input line string parser is to realize and override the **getMetadata()** and **parseFields(final String str)** method. The following is some sample code of IEEE CDF bus data parser.

```
public class IeeeCDFBusDataParser extends AbstractDataFieldParser {
    @Override public String[] getMetadata() {
        return new String[] {
          //  0---------------1---------------2---------------3---------------4
            "BusNumber",  "BusName",       "Area",          "Zone",        "Type",
          //  5             6              7               8              9
            "VMag",       "VAng",        "LoadP",        "LoadQ",       "GenP",
          //  10            11             12              13             14
            "GenQ",       "BaseKV",       "DesiredV",     "MaxVarVolt",   "MinVarVolt",
          //  15            16             17
            "ShuntG",    "ShuntB",        "RemoteBusNumber"
        };
    }

    @Override public void parseFields(final String str) throws ODMException {
        if (str.indexOf(',') >= 0) {
            final StringTokenizer st = new StringTokenizer(str, ",");
            int cnt = 0;
            while (st.hasMoreTokens()) {
                this.setValue(cnt++, st.nextToken().trim());
            }
        } else {
            //Columns  1- 4   Bus number [I] *
            this.setValue(0, str.substring(0, 4).trim());

            //Columns  6-17   Name [A] (left justify) *
            this.setValue(1, str.substring(5, 17).trim());
          ……
         }
```

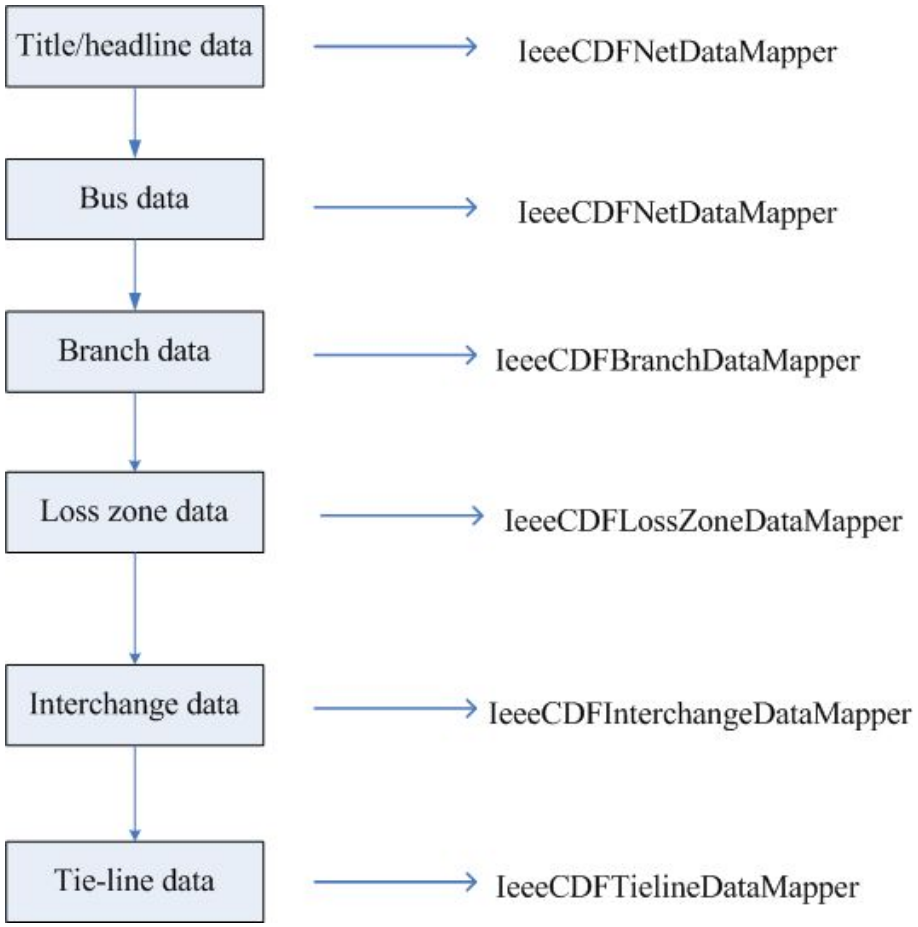
2.4.2.2 Implement input line string mapper

For a line string mapper, there is a corresponding line string parser, which parse the input data first, then the mapper can easily access the data and map it accordingly to the ODM java objects.
The following are some sample code to implement input line string mapper:

```
// define data parser
IeeeCDFBusDataParser dataParser = new IeeeCDFBusDataParser();

public void mapInputLine(final String str, AclfModelParser parser) {
    // parse the input data line
    dataParser.parseFields(str);

    // map data to ODM
    String busId = dataParser.getString("BusNumber");
    LoadflowBusXmlType aclfBus = parser.createAclfBus(busId);
    ...
}
```

2.4.3 Data Adapter

With parsers and mappers being implemented for all data sections, the last step is to logically organize them together to parse a complete data file or set of files. The IEEE CDF adapter will be taken as an example, the sequence of the data section is shown below, with the corresponding data mapper shown on the right.



A simple logic for processing such sequential data would be : 1) iterate over each line of the input data file 2) determine what type of data is under processing 3) apply the proper data mapper to parse and map to data into ODM. The outline of the IEEE CDF adapter implementation is given below:

```
// read the first line - head line
 // sample :
 //       08/19/93 UW ARCHIVE           100.0  1962 W IEEE 14 Bus Test Case
 String str = din.readLine();
 netDataMappe.mapInputLine(str, parser);

 int dataLineIndicator = DataNotDefine;
 do {
          str = din.readLine();
          if(str!=null){
             if (str.trim().equals("END OF DATA"))
                   break;
                   try {
                         /*
                          * process section head record - for example
                          *
                          *   BUS DATA FOLLOWS
                          */
                         if ((str.length() > 3)
                                       && str.substring(0, 3).equals("BUS")) {
                               dataLineIndicator = BusData;
                               ODMLogger.getLogger().fine("load bus data");
                         }
                   else if (
                   ......

                         /*
                         *  End of processing section head record
                          * parse data line
                          */
                         else if (dataLineIndicator == BusData) {
                               busDataMapper.mapInputLine(str, parser);
                         }
                         else if (dataLineIndicator == BranchData) {
                               branchDataMapper.mapInputLine(str, parser);
                         }
```

- **IEEE CDF Format Adapter:**

The complete implementation of IEEE CDF adapter is available from:
https://github.com/InterPSS-Project/ipss-odm/tree/master/ieee.odm_pss/src/org/ieee/odm/adapter/ieeecdf

2.5 ODM -> InterPSS

1) ODM Parser

Appendix-C Useful Plugin Tools

C.1 InterPSS Network Topology Visualizer
C.2 Network Equivalencing Tool