Copyright © 1987, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

# AN OBJECT-ORIENTED DATA REPRESENTATION AND TASK BASED REASONING SYSTEM FOR ENERGY MANAGEMENT SYSTEMS: A PROPOSAL

by

Andreas F. Neyer, Karl Imhof, and Felix F. Wu

Memorandum No. UCB/ERL M87/75

27 October 1987

COVER PAGE

# AN OBJECT-ORIENTED DATA REPRESENTATION AND TASK BASED REASONING SYSTEM FOR ENERGY MANAGEMENT SYSTEMS: A PROPOSAL

by

Andreas F. Neyer, Karl Imhof, and Felix F. Wu

Memorandum No. UCB/ERL M87/75

27 October 1987

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

# AN OBJECT-ORIENTED DATA REPRESENTATION AND TASK BASED REASONING SYSTEM FOR ENERGY MANAGEMENT SYSTEMS: A PROPOSAL

by

Andreas F. Neyer, Karl Imhof, and Felix F. Wu

Memorandum No. UCB/ERL M87/75

27 October 1987

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

# An Object-Oriented Data Representation and Task Based Reasoning System for Energy Management Systems: A Proposal [1]

*Andreas F. Neyer, Karl Imhof [2] and Felix F. Wu*

Department of Electrical Engineering
and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720

*ABSTRACT*

The research reported has been motivated by the idea of integrating expert systems into energy management systems (EMS). A study of current expert systems in the area of power engineering demonstrated the need for a fundamental analysis of the problem. Based on an examination of the data and reasoning knowledge we propose a data modeling and reasoning system that facilitates building more flexible software systems for EMSs. The system is based on an object oriented data model whose objects correspond directly to the entities of the power system. The processing is performed by calling local procedures associated with the objects. Additionally the system offers three fundamental generic tasks. A generic task is a basic programming building block that can be parameterized and applied to many different problems. Finally a rule based expert system tool is described that can call any procedure that is defined for the objects of the data model. The use of the system is demonstrated with two conventional applications, load flow calculation and state estimation, and with a rule based system for contingency selection.

---

[1] Research sponsored by the National Science Foundation Grant ECS-8715132.

[2] On leave from Brown Boveri Company (BBC), Baden, Switzerland.

## Table of Contents


**1. Introduction** ........ 1

**2. Power System Knowledge**

2.1 Data Knowledge in Energy Management Systems

2.1.1 Power System Model ........ 2

2.1.2 Characterization of the Power System Data Knowledge ........ 3

2.2 Problem Solving Knowledge in Power System Operation ........ 3

2.2.1 Power System Analysis ........ 4

2.2.2 Power System Synthesis ........ 4

2.2.3 Learning ........ 4

2.2.4 Characterization of the Problem Solving Knowledge ........ 5

**3. A Critical Review of State of the Art**

3.1 Data Knowledge ........ 7

3.2 Reasoning Knowledge ........ 8

**4. Four New Software Concepts**

4.1 Data Abstraction: A Data Representation Concept ........ 10

4.2 Generic Tasks: A Reasoning Concept ........ 10

4.3 Local Processing: A Function Implementation Concept ........ 11

4.4 Declarative Programming: A Modular Knowledge Representation Concept ........ 12

**5. A Data Representation and Reasoning System for EMSs: A Proposal**

5.1 Object-Oriented Programming: An Implementation Tool ........ 14

5.2 Definition of the System

5.2.1 The Object-Oriented Model ........ 15

5.2.2 The Implementation of the Generic Tasks ........ 17

5.2.3 Expert System Tool ........ 23

**6. Use of the System: Three Examples**

6.1 Load Flow Calculation ........ 26

6.2 State Estimation and Bad Data Detection ........ 29

6.3 Rule Based Contingency Selection

6.3.1 Introduction ........ 32

6.3.2 The Contigency Selection Rules ........ 32

**7. Conclusion** ........ 35

**8. References** ........ 36

**Appendix A: DC Load Flow in Declarative Form** ........ 39

**Appendix B: Knowledge Representation** ........ 42

## 1. Introduction

The research described in this report was motivated by the idea of integrating expert systems into energy management systems (EMS). Expert Systems (ES) evolved from artificial intelligence (AI) research and are now applied to all fields where expertise plays a crucial role. In EMSs expert systems have been proposed for areas where formal reasoning methods are missing and heuristic solution techniques based on expert knowledge have to be applied [Alv.2, Liu.1, Sak.1, Schu.2, Tal.1]. However, a study of those systems shows that the approach of applying a general purpose expert system tool to power system problems poses major problems and works only for relatively small prototypes. The reason for this is that the field of EMSs is fundamentally different from other areas where ES are applied:

- For power system applications there exist well defined models. Usually these models are quite extensive and require efficient data modeling tools.
- There is a host of analytical tools that should be integrated into an expert system.

General purpose expert system tools, however, are weak in exactly these two domains. The data representations in expert systems are often unstructured and very inefficient. The data model of expert systems tools is also a major problem for the application of analytical tools. Therefore a more fundamental analysis of the problem is necessary before ESs can be successfully applied to EMSs:

In this report we characterize the knowledge involved in power systems and describe the fundamental problems with current implementations of EMSs. Then we present four new software concepts that increase the efficiency of the programmer and the flexibility of the software. All of these software concepts facilitate more user friendly, conceptual programming. This is usually paid with a lower efficiency in execution and more memory requirement. However, we are convinced that such a decrease of run time efficiency can be tolerated because of the rapid developments of faster and more efficient computer hardware (parallel processors). The cost for software development is steadily increasing, whereas the cost for computer hardware is decreasing. Therefore it is necessary to provide the power system programmer with more efficient tools.

The system proposed addresses this problem by providing facilities for programming on a conceptual level. We have defined an object-oriented data model that has a direct correspondence to the power system. Each element of the power system is represented as an object in the data model. However, the object-oriented model is not purely a data base, but it is also responsible for all the data processing. The standard EMS functions such as load flow calculation, state estimation and observability analysis have been broken down into processing steps that are local and can be directly associated with the objects of the data model. These local processing steps are implemented as messages of the objects.

This structure guarantees a strictly modular organization of the software. The modularity increases the flexibility and reusability of the software. The objects are the modules and can communicate with each other only over the well defined function interfaces (messages). The implementation of an object is private to that object and not visible from outside. Modifications of the program are therefore confined to a local area, basically a single object.

In addition to the object oriented data model the system provides three generic tasks: graph traversal, conceptual object manipulation and element power flow. Generic tasks are basic reasoning building blocks that are parameterized with some data or problem solving knowledge in order to be applied to a specific problem. The three generic tasks discussed are fundamental in the area of power systems, i.e. they can be used as program building blocks in many EMS functions. The application of the generic tasks together with the object oriented data model is demonstrated in the functions load flow calculation, state estimation and bad data detection.

The well defined interfaces make it also easy to implement a rule based expert system on this data model. The rules can access the data and initiate some processing in the same way as all the other functions, i.e. by sending messages. The rules can call any procedure that is defined for the objects and therefore also all the standard EMS functions. Additionally the rules can be efficiently organized by associating them with the objects they affect. As an example a rule based system for contingency analysis is outlined.

## 2. Power System Knowledge

In this chapter the various types of knowledge encountered in a control center are listed and classified. Basically, we distinguish between two classes of knowledge, data knowledge and problem solving knowledge (reasoning knowledge). Data knowledge describes the power system and its environment, whereas the problem solving knowledge describes how the system should be operated. The latter is a rather broad class. It includes the power system functions implemented in EMS system as well as the knowledge contained in handbooks and operation manuals and the experience of the operator.

### 2.1. Data Knowledge in Energy Management Systems

The data knowledge is divided into the knowledge about the power system model and miscellaneous knowledge, for example, the knowledge about the influences of the environment.

The model data is structured in a natural way. The knowledge can be associated either with the physical elements of the power system such as lines, transformers, buses, or with the conceptual entities such as nodes and observable islands or network groups.

The miscellaneous knowledge e.g the information about the environment such as weather conditions, available computer power, social impacts (holidays, daily consumer behavior) can be usually associated with the power system data knowledge. Because this knowledge is not as important as the power system model knowledge it will not be discussed further.

#### 2.1.1. Power System Model

The power system is modeled and described as a set of some power system entities and the relations between them. The extent of the modeling of the entities and their relations depends on the application.

E.g. For a load flow study it is sufficient to model transmission lines and transformers as branches between the two nodes. Such branches are usually modeled as $\pi$-elements. For the monitoring of a power system, however, it is necessary to describe all the physical elements (buses, switches, etc.) which belong to a lower hierarchical level of the model description.

The data about the power system model has been divided into static and dynamic data. Whether some data is static or dynamic depends on the time period considered. In the extreme of a very long time period most model data is dynamic. For example, if parameter estimation is performed periodically then also the parameter values are dynamic data.

##### 2.1.1.1. Static Model Data

The static parameter data associated with the entities are time independent parameters such as impedances, voltage and line flow limits. The static structural data describes the relations between the entities. These relations are either the connections between entities such as those between line ends and connected switches or their hierarchical structure. The hierarchical structure defines which elements form a generic entity on a next higher level of the representation. For example, some buses, switches, transformers and the line ends connected to them form a substation.

##### 2.1.1.2. Dynamic Model Data

The dynamic model data describes the state of the system at a certain time. This description changes as the system evolves over time. Thus, to each dynamic data a time tag in a implicit or explicit form has to be added. The dynamic model data are either measured or derived quantities.

The sensors attached to the power system provide measured quantities in the form of analog quantities and events (discrete values). While analog quantities are usually reported cyclically, events are transmitted by exception.

E.g. Analog quantities are usually active and reactive power, voltage, current, frequency, etc. associated with specific entities such as buses, line ends, etc. Events may be switch changes, transformer and capacitor tap position adjustments and alarms.

The derived data includes all the data derived in some way from the measured data. Derived data can be the result of a procedure such as the state estimator for analog data or the alarm processor for discrete data. But it can also be the conclusion of an operator or the result of applying an expert system to measured or derived data. Derived data is either discrete or analog and is given in a qualitative or quantitative form.

### 2.1.2. Characterization of the Power System Data Knowledge

The information about the power system model can be associated with the different power system entities. Characteristically there are only a few different and well known classes of entities in a power system. Basically, such classes stand either for physical entities, e.g. buses, lines, and transformers or for conceptual entities, e.g. nodes, branches, and observable islands. While the physical entities stand for the static model, the conceptual entities represent the dynamic part of the power system model.

Most classes in a power system include a large number of particular entities. These are called instances of the class. For example, the class line consists of the particular elements LINE_1, LINE_2, LINE_3, etc.

Often there is a hierarchical relationship between different classes of entities in a power system. A subclass represents a specialization of the hierarchically higher class. For example, the class of physical elements consists of the subclasses: buses, lines, breakers, switches, etc.

In addition to the class-instance and the class-subclass relationship there are two additional relations between entities that are important for power systems. Both are relationships between instances. The first one represents the connectivity between elements; the second defines the relation between a group of entities and its members. These two relationship are explained below.

Connectivity: Since the power system is a network, its structure must be modeled. The elements of the power system are connected to each other. Connectivity is a bidirectional relationship. For example, if LINE_15 is connected to BREAKER_24, then also BREAKER_24 is connected to LINE_15. Connectivity relations between entities are important on various levels in the power system model. For example, for fault analysis the connections between all physical elements have to be considered, whereas for a load flow calculation only the connections between nodes and branches are important.

Groups: In power systems entities are often collected in groups. Such groups are conceptual entities such as nodes, substations, network groups, etc. Each group is a set of particular entities that have certain properties. For example, a node consists of all elements which are interconnected without impedance in between. These conceptual entities are essential in power system modeling because most functions work directly with these entities.

The data knowledge either represents a snapshot of the power system at a particular time or describes the changes between snapshots. The latter is temporal knowledge because it describes the behavior of the system with respect to time.

Data knowledge is given in a quantitative or qualitative form. Quantitative information is exact knowledge obtained by commonly known algorithms such as topology (network configurator) and state estimation. Qualitative information is condensed and non-exact knowledge describing the overall behavior of the system. While the first type of information deals with the numeric nature of the system state description, the second type of information is concerned with a more symbolic (logic) representation.

Considering a particular time step the dynamic model data has the same structure as the static model data. However, if the changes of the system over time are considered, then cyclically reported data as well as events have to be treated. In AI the problem of representing changing situations is known as the *frame problem* [Ric.1]. Whenever the data model is accessed the time has to be specified either implicitly (current time) or explicitly.

## 2.2. Problem Solving Knowledge in Power System Operation

The power system problem solving knowledge or reasoning knowledge includes not only the control and monitoring functions usually implemented in a modern EMS, but also the information contained in

handbooks and operation manuals and the knowledge the operator gained from experience. Therefore a very broad range of problem solving knowledge has to be considered. On one side there are well defined analytical tools such as load flow calculation and state estimation and on the other side there is the heuristic knowledge that a dispatcher uses in the operation of the power system.

In order to organize the problem solving knowledge a criteria for classification is needed. Such a classification should depend only on the structure of the problem solving knowledge and should be independent of the way the knowledge is represented. In the literature function-oriented classifications can be found [Hor.1]. However, in these classification schemes only computerized functions are taken into account. Since we want to include all the knowledge about the operation of power systems we generalize these classifications such that non-computerized problem solving knowledge is also included. Hence, we divide the operational tasks into the analysis task, the synthesis task, and the learning task. Note that the current classification schemes are more purpose-oriented whereas we try to emphasize the structure of the underlying problem solving knowledge. For example, security assessment and economic assessment belong to the same category since they have the analysis of a given power system in common.

### 2.2.1. Power System Analysis

This task is concerned with the analysis of the data available in a control center in order to provide a concise description of the state of the system. This definition includes monitoring functions as well as security and economy assessment functions. The inputs are the basic power system description and the information (raw data) provided by the monitoring devices. The outputs are either consistent non-redundant data sets describing the power system and its changing behavior or some representative key figures such as power system losses and security indices describing the behavior of certain power system properties. Examples for this are state estimation, contingency analysis, load flow as well as the operator's problem solving knowledge which helps him/her to get an overall picture of the power system and the current system trends.

Usually only the basic functions for providing a non-redundant power system description are implemented in EMSs. However, the knowledge of how to identify and analyze changes in the functionality of the power system and the knowledge of how to get an overall picture of the power system behavior exists neither in a formal nor in a computerized representation.

### 2.2.2. Power System Synthesis

Power system synthesis is the task of finding a single or a sequence of control actions to be applied to the system in order to achieve some security and economic objective. For a power system the control actions are limited to changes of switches, adjustments of tap positions (transformers and compensators) and generator set-points. Future control tools may be load management and spot pricing.

As long as the control actions are applied to the current power system we are talking about power system control. In case of applying the synthesis task to a power system model representing some future system states we are talking about power system planning.

Today, the synthesis is either performed closed loop without human interaction, e.g. automatic generation control, or open loop where the recommendations for the running power system or for some future system states are provided by the operator. In the latter case some problem solving knowledge is not given in computerized form. In both situations appropriate power system analysis tools are a prerequisite for the synthesis task.

### 2.2.3. Learning

Learning is defined as the change in the behavior to a given situation brought about by repeated experiences in that situation. In other words learning is the acquisition of problem solving knowledge for both the analysis and synthesis task. There are currently no implementable methods for the learning task. The problem solving knowledge for many analysis and synthesis task is acquired by the operator and not formalized. The best example for this type of reasoning knowledge is the knowledge of how to diagnose the behavior of a power system. The operator has to acquire this knowledge by observing the

system.

### 2.2.4. Characterization of the Problem Solving Knowledge

The problem solving knowledge specifies how new quantitative and qualitative information about the power system process can be derived from the input data. The inputs are the basic power system model as well as the measured and the derived data. The input and the output information is given either for a specific time (snapshot or non-temporal information) or for a set of transitions (behavioral or temporal information). The consideration of the temporal behavior of the power system process adds additional information to the model and to the reasoning knowledge.

#### 2.2.4.1. The Analysis Task

In order to facilitate the insight into the problem solving knowledge used in EMS, we restrict ourselves to the examination of the analysis task. In the following we introduce some present and future EMS-functions used to solve the analysis task. The goal is to provide an overview of the type of problem solving knowledge used throughout the analysis task and to determine the basic reasoning blocks used to implement such functions.

Particularly the last two functions in the list, the intelligent alarm processor and the state abstractor are functions that are usually not implemented in EMSs. However, they are of major concern for the operation of a power system [Schu.1]. Therefore they are discussed in more detail.

TOPOLOGY (network configurator) is concerned with building up and maintaining quantitative, high level structure information summarizing the current switch positions. Appropriate algorithms scan the power system elements and test which of them share the same properties (e.g. electrically connected) [Imh.1].

OBSERVABILITY is concerned with building up and maintaining quantitative, high level structure information summarizing the topology result, the measurement placement and the current availability of the measurements. Basically, appropriate algorithms search the largest possible spanning tree in the node-branch model. A node with an injection measurement or a branch with a flow measurement extends the spanning tree by that node or branch, if the appropriate measurement equation is structurally or numerically independent from the measurement equations already in the spanning tree. Algorithms to compute the structural and the numerical observability are given in [Cle.1, Mon.1].

STATE ESTIMATION provides quantitative and non-redundant state information (complex voltages) for the observable islands given a set of redundant measurements. Appropriate algorithms solve a non-linear optimization problem. The objective is to minimize the deviation of the measured values from a consistent set of values that satisfies the load flow equations. Appropriate non-temporal implementations are discussed in [Gar.1]. Temporal implementations simply start with the previous result as an initial guess.

BAD DATA DETECTION improves the matching between the given measurements and the computed voltage profile in case there are abnormally large differences between observed and calculated quantities. Hypotheses about which measurements are bad, are proposed based on the largest residuals or other heuristic rules. Hypotheses are tested by rerunning the state estimator. Non-temporal algorithms are presented in [Mon.1, Mon.2].

LIMIT CHECKING is a simple, straightforward task which is used to determine which quantities have violated various operational and physical limits. It is the only tool today providing some primitive qualitative information. It is used to check the current as well as simulated system states (contingency analysis, short circuit analysis, etc.).

CONTINGENCY ANALYSIS evaluates the effect of outages on the power system. Usually, contingency analysis is split up into two activities: the selection of the appropriate contingencies, and the simulation of the proposed contingencies. For the second activity, starting from a base case, the outages are modeled and the resulting system state is analyzed. There exits a static, steady state contingency analysis that examines the power system state some time after the contingency, as well as a dynamic contingency analysis that considers the transition from the base case to the after-contingency

steady state. Contingency analysis itself is a temporal task.

The INTELLIGENT ALARM PROCESSOR models the effect of all incoming messages from the power system such as switch indication changes, tap settings and analog-valued changes and proposes a temporal quantitative power system change scenario augmented with alarms to be expected from the power system. The incoming alarms are matched with the modeled and hypothetical alarms and in case of successful matching replaced by the power system change scenario obtained from the modeling. A typical case is given below:

E.g. Due to a short circuit at a transmission line, the protection system causes the opening of the line at one side (> relay alarms, switch indication changes). As the line is now deenergized (> alarm) and the automatic reclosure procedure fails (> alarms), the line will be opened at the other side too (> relay alarms, switch indication changes). The outage causes a local redistribution of the flow which in turn generates some overload in nearby lines (> alarms). Furthermore, some LTC-transformers fail to maintain their set-points (tap change messages, alarms).

Hence, to come up with hypothetical alarms non-temporal, how is the current state, as well as temporal model information, what has been changed, is required. The final result consists of suppressing component-oriented alarms and messages in favor of providing some higher level information describing the power system changes. For example, in our case the intelligent alarm processor would output a sequence of events such as:

E.g. Short circuit in Line XYZ, reclosure failed opening of Line XYZ at both sides, overloads in Line ABC (15%), Transformer HJL cannot longer maintain its set-point (2kV over limit).

It is important to note that for our definition of an the intelligent alarm processor the whole line of commonly known analysis functions (topology, observability, state estimation, etc.) is required.

STATE ABSTRACTION is concerned with providing some qualitative, high-level structure or analog valued information about the power system state and its changes. Inferring this type of knowledge is left to the operator because the appropriate problem solving knowledge is heuristic and formally not defined. Basically, the state abstraction has to condense the quantitative knowledge provided by functions such as state estimation and contingency analysis and provide a concise overview of the system state and its changes.

State abstraction is a job operators perform routinely. Nevertheless, the full functionality of state abstraction is not clear. Thus, here only some basic ideas can be provided. Basically, the operator observes some representative power system quantities and relates them to a qualitative description and qualification of the power system. This quantitative-qualitative relation he keeps in mind in the form of some power system state and structure prototypes or in the form of rules relating power system quantities to some stored information. In a very simplified form this maybe described, for example, as follows:

> Total active generated power is higher than 3000 MW, the losses exceed 4% and the tie line flows have an upward trend -> Line ABC, Line DEF... are critical contingencies, reactive power deficiency in Area XYZ.

The relation between a power system case description, a power system instance, and some stored information about typical power system instances or typical situations have to be learned by the operator. This is done by experience or training.

# 3. A Critical Review of the State of the Art

## 3.1. Data Knowledge

Two decades ago EMSs have emerged from some independent functions used to study the behavior of planned or current power systems. Each function has been developed based on a particular algorithm and therefore each has its own data model that is determined by this algorithm. The data model plays a minor role and is only chosen to serve the appropriate algorithmic needs. An example for this is the sparse matrix representation of the data for a load flow calculation. This purely algorithm-oriented view brings the following problems:

- Since each function has its own data structure, this data structure has to be built up each time the function is called which affects performance adversely. But not only the performance is affected but also the effort for maintenance. The same information is stored differently depending on the data model of the particular functions. Therefore all these different data structures have to be maintained which requires a considerable effort for systems that are as complex as EMSs.
- There is a very strong relationship between the data structure and the particular procedures. Therefore the procedures that access some data must have knowledge about the structure and the implementation of the data models. This means that if the data model is modified all the procedures that depend on it have to be modified too. However, for these procedures implementation issues of the data are of no interest. This is a problem of surface area [Cox.1]. The more information is needed to access some data the larger is the surface area between the data and the functions. Large surface area make programs inflexible and hard to maintain.
- In large software systems such as EMSs numerous subroutines are used to implement the major power system functions such as load flow calculation, state estimation, etc. These subroutines are very often the same for different functions. Yet, since the functions have different data models, the subroutines have to be reprogrammed for each function. This increases the effort for the development and the risk for programming errors.
- Because the different functions have different data models special interfaces are needed that transfer the information from one data model to another. These interfaces require considerable programming and maintenance efforts. Additionally they affect the performance of the system. This is particularly the case if expert systems (ES) are applied to EMS. Expert systems for EMS should make use of the host of available analytical tools. This means that the ES has to collect all the data needed for the function to be called from its own data model and transfer into a format that can be handled by the conventional function. However, ES system tools are not very efficient for this data transfer.

Another aspect to be considered is the organization and structure of the data model. Currently in most data models for EMS functions the information is stored vector-wise. This means that the values of attributes for elements of the same type are stored together. For example, the voltage values of all the nodes are collected in a single vector. There are two reasons for this: First this model corresponds directly to the usual mathematical formulation of the problem where vectors and matrices are used. Second, EMS are commonly programmed in languages that have only very restricted data modeling capabilities.

In the characterization of the power system data knowledge we have pointed out the particular structure inherent in the data model. This structure does not correspond to vectors. Rather power system entities are the basic elements of the data model. Therefore it is natural to reflect this structure in the computer model of the power system.

A first example of such an approach is described in [Imh.1]. A linked record data structure is proposed for the power system model data. Each each entity of the power system is represented by a record in the computer model. Between the linked record structure and the power system is a direct correspondence. The data model can be viewed as a graph of a power system and the graph techniques for solving a sparse linear equation presented in [Pis.1] can be employed. Entities are either physical elements such as buses, lines, transformers, etc. or conceptual elements such as nodes and observable islands.

The links represent the relations between the entities which are either their hierarchical structure or the connections between them.

It is shown that the EMS functions run easily and efficiently on such a model. For example, the CPU-time required for an ordinary load flow function was be reduced at least by 15% in comparison to the same load flow operating on sparse matrices. A reason for this is that indexing schemes for sparse matrix representations are unnecessary.

The record link data model has been implemented in Pascal. Each entity of the power system is represented by a record. The links are modeled by pointers. Pascal is a strongly typed language. Each class of elements has its own type that must be considered when accessing the data. There is still a considerable surface area between the data model and the functions requiring the data. For example, entities such as line, 2-winding transformer have a serial impedance. But since they are represented by different data types this serial impedance has to be accessed differently.

To find improved representational possibilities we examined knowledge representation techniques in AI (see Appendix B). Knowledge representation in AI, however, is mostly concerned with the problem solving knowledge rather than the data knowledge. The only structured AI representation techniques for data knowledge are scripts, semantic nets and frames. Since semantic nets and scripts are closely related to frames we discuss the frame structure [Fik.1].

A frame provides a structured representation for an entity or a class of entities. Each frame contains a set of slots (attributes) that describe the particular class or entity. In power systems such classes can be element, bus, line, compensator, node, observable island, etc. A particular object, such as BUS_15 is called an instantiation of the class, in this case the class bus. In a frame based system each class can be a specialization of a more generic class (superclass). For example, the class element in a power system consists of the subclasses bus, line, transformer, breaker, etc.

A major advantage of such a class system is property inheritance. Instances of objects inherit the slots from their class description and the classes inherit properties from their superclasses. This implies that properties of objects are stored at the highest possible hierarchical level which reduces redundancy.

### 3.2. Reasoning Knowledge

In a modern control center the dispatcher has access to a tremendous amount of data and a host of analysis tools that assist the operation. In fact, there is such a bulk of data that it is very difficult for the dispatcher to identify the critical aspects of the power system. Additionally, the operator should find time to make use of the provided analysis tools which in return produce more data. Exactly these concerns have been expressed in some recent papers [Hor.1, Schu.1]. Horton creates the slogan "We need to produce more intelligence and less data, data, data." By intelligence he means filtered or higher level data that tells us something we don't know but that we should know.

We have defined this function that provides a qualitative description of of the power system the state abstractor. Such functions do not exists yet and there is very limited research in this direction although the knowledge about the critical aspects is crucial for the dispatcher in order to make correct decisions. If the operator has an incorrect idea of the current state of system then his/her actions may lead to a disaster as it happened in the New York blackout [Wil.1]. These problems have lead researchers to apply artificial intelligence techniques to power system problems. Particularly expert systems, a subfield of AI, seem suited for power system problems, since expert systems solve tasks that only qualified human experts, in the power system case the dispatcher, can solve.

An example that addresses the problem of filtering power system data is the intelligent alarm processor [Wol.1]. Typical examples for expert systems applied to other problems are the rule based system used to solve the restoration problem [Sak.1], the blackboard approach for fault analysis in power systems based on OPS5 [Tal.1], and an ES which assists with the decision making of reactive power/voltage control [Liu.1].

If we examine the papers which propose ESs for EMSs, we realize that most of the systems are still relatively simple prototypes and very few are successful in solving real world problems. There are some common stumbling-blocks. ESs have difficulties to efficiently handle the huge amounts of data necessary to describe large scale power systems and analytical tools such as load flow calculations, optimal

power flow, etc. are not easily integrated in an ES (at least not in their current form.)

We are convinced that the major reason for this is the insufficient attention that is paid to the representation of the data as well as the problem solving knowledge. The prototypes presented in the literature make use of general expert system tools or artificial intelligence programming languages such as OPS5, KEE, Prolog and Lisp. As far as the data knowledge is concerned, general expert system applications usually do not have a formal and well defined model as we have in power systems. Therefore the data modeling capabilities of expert system tools are not adequate for power systems. Either they do not offer a structured data model or the access to the information is not efficient enough to implement extensive power system models.

There are many applications in EMS were heuristic reasoning is desirable. After all the operator performs such tasks day by day. For complex problems such as restoration and voltage control it is necessary for the dispatcher to analyze the system first. This means filter the incoming data to obtain a description of the state of the system on a abstract level. Only based on this analysis proper decisions can be made. Since expert system try to model human reasoning it is important that expert systems for problems such as planning and control should work from a qualitative rather than quantitative description of the system. Analysis functions that produce such a symbolic description have not been researched enough. Possibilities of a qualitative description may be area summaries, performance figures for important system aspects such as voltage level, stability, limit violations, closeness to the limits, etc. A first step in this direction is described in [Pao.1]. An inductive learning procedure is applied to acquire a qualitative understanding of the system behavior.

Finally let us examine the problem solving methods used in current EMSs. Surprisingly there is hardly any variety in the way EMSs solve the problem. Each system consists a various number of functions that are mostly algorithms implemented as procedures that gather all the data needed from the database, do their processing and deposit the results back to the data base.

These functions are usually global in the sense that they do not make use of the local properties of the model. They do not exploit the fact that most changes affect the system only locally. For example, a compensator affects the voltage profile mostly in the close neighborhood. Additionally subroutines of power system functions only need very localized information. For example, the factorization step of node requires only information about neighboring nodes and branches. Factorization steps can be naturally executed parallel as long as the nodes are far enough apart.

Furthermore EMSs make hardly any use of temporal problem solving. By this we mean that most functions analyze the power system at a particular time, but do not analyze its behavior over time. One area where temporal processing is useful are update algorithms. For example, if the effects of a change in the system state on the functionality of the system are known, then they can be calculated without complete analysis.

Other applications were temporal reasoning should be applied are alarm processing, bad data detection and contingency analysis:

There has been some recent research in intelligent alarm processing [Wol.1]. The intelligent alarm processor presented examines a sequence of incoming alarm messages, relates them to each other and produces condensed summary alarms. Although this reduction of the number of alarms supports the operator in finding out what is happening in the power system, it is still the operator's task to infer the proper changes.

As far as bad data detection is concerned appropriate temporal solutions are apparent. For example, measurements recognized before as bad data or those with an unusually large deviation compared to the old value are likely to be bad.

The temporal issues that have to be addressed in contingency analysis are cascading effects and how base case modifications, i.e. the power system changes, affect the contingency analysis results.

## 4. Four New Software Concepts

In the previous chapter we showed that one of the main stumbling blocks in getting better, more sophisticated and more flexible EMSs are the data and reasoning representation techniques. To overcome this situation we have to search for new software concepts. Therefore, we present four distinct programming concepts. Their realizations leads to innovative tools for the implementation of present and future EMS-functions [Bob.1].

### 4.1. Data Abstraction: A Data Representation Concept

Data Abstraction provides the means for reducing the interdependencies of software components and increases the reliability and flexibility of software systems [Pas.1]. Data abstraction is achieved by defining abstract data types that consist of an internal representation and a set of local procedures for the access of the information. The concept is also called information hiding since the internal representation is only visible within the scope of the particular software module and completely hidden from the outside. The only way to access the information is over the access procedures which define a fixed communication interface between the software module and its environment.

The surface area between software components is the amount of specifications that is necessary for them to work together. These specifications include the data element names, data type names and function names [Cox.1]. Large surface areas between modules is bad for software productivity and reliability because the modules depend very much on each other. A change in one of them will affect many others. Data abstraction effectively reduces the surface areas by allowing access to the information of a module only over a well defined interface.

Data abstraction is a concept that is exploited in many of the modern programming languages. One example is Modula-2 where the *opaque type* provides a certain degree of data abstraction. Another example is Ada [Buz.1, Hab.1, Sam.1] that defines packages and private types. The package definition corresponds to the visible interface that permits access to the internal data only over the defined declarations. The private types correspond the the hidden part or the internal representation of the module.

The third example of programming tools that provides data abstraction are the object-oriented programming languages. Object-oriented programming is a programming concept that is applied in many languages such as Smalltalk [Gol.1, Kae.1], C++ [Str.1] and Objective-C [Cox.1], to mention only the most prominent examples. The objects are the software modules that define a strict interface to other objects. The access to the objects is only possible over predefined functions, in object-oriented terminology called messages or methods.

In object-oriented languages data abstraction is enhanced by property inheritance. Objects can be structured in classes, subclasses and instances. An instance is a particular element of a class. Property inheritance is a data abstraction service that automatically searches the corresponding classes and superclasses of a particular object if some information is required. This means that although a particular type of information is required from an object, this information is not necessarily stored inside the object, but rather with its class or one of its superclasses. This reduces the redundancy since the data can be contained in the possibly highest appropriate class or category.

### 4.2. Generic Tasks: A Reasoning Concept

The concept of data abstraction improves the programmers efficiency by introducing more sophisticated data building blocks. A similar concept is defined for reasoning: The generic task is a basic reasoning building block. It specifies how to apply problem solving or reasoning knowledge for a well defined class of problems [Byl.1, Sam.1].

In [Cha.1] a more detailed definition is provided: The generic task is characterized by providing information about (1) a task specification in the form of generic types of input and output information (2) specific forms in which the basic pieces of domain knowledge (problem solving knowledge) are needed for the task, and specific organizations of this knowledge particular to the task, and (3) a family of control regimes appropriate for the task.

In order to illustrate the various levels of generic tasks some examples from the computer area are provided below. The lowest level of generic task is found in assembler languages. A given instruction is accepted by the CPU, which in turn infers the basic operations necessary to execute the instruction. In fact each assembler instruction stands for a generic operation that is translated into a specific microcode sequence.

In high-level languages such as FORTRAN, PASCAL, and C there are language paradigms such as DO-loops, REPEAT-UNTIL-loops, WHILE-loops, and CASE-statements besides the basic instructions such as arithmetic, logic and input-output operations. These language paradigms are generic tasks that can be parameterized to perform a specific operation. For example, a WHILE-loop parameterized with a stopping condition and the knowledge of how to get the next entity in a list represents the specific task of searching the list for a specified entity.

More recent languages e.g. ADA [Hab.1], provide tools to extend the set of language defined paradigms by user defined generic tasks. As an example for an ADA generic task, a general purpose paradigm to manage a queue (enqueue and dequeue operation) can be defined. For a particular application, e.g. a specific type of list, we parameterize the ADA generic task. Thus are the very same routine can handle all possible types of of queues.

The highest level of generic tasks are defined in the area of AI. There a generic task specifies a problem solving strategy that is parameterized with the particular problem. Simple generic tasks such a rule matching can easily be separated from a particular application such as in inference engines of rule based expert systems. However, generic tasks such as diagnosis are well known by experts, but cannot be defined generically. In [Cha.1] six different generic task are identified as reasoning building blocks: hierarchical classification, hypothesis matching, knowledge directed information passing, abductive assembly, hierarchical design by plan selection and refinement, and state abstraction.

A generic task consists of some problem solving knowledge which itself makes use of some lower level generic tasks. This refinement process can be stopped as soon as the language or the software system is able to provide these reasoning building blocks. These generic tasks in the extreme case are assembler instructions. In current EMS they are either programming language provided paradigms such as PASCAL set operations, FORTRAN DO-loops or library functions, e.g. for sparse matrices.

Providing higher level reasoning building blocks, has been proved to be effective for reducing software development and maintenance costs. For example this can be readily demonstrated when coding the same function in PASCAL [Ald.1, Jam.1] instead of in assembler. Not only the cost for implementing a function decreases, but also more complex functions can be attacked. Moreover, the assembler implementation of power system functions such as optimal power flow or contingency analysis although theoretically possible, is in reality too complex. We are convinced that the lack of a system providing some higher level generic tasks is the reason why breakthroughs in EMS-functions such as power system restoration are still missing.

The change to a system with higher order generic tasks is traded off with lower run-time efficiency. This can be shown by comparing, for example, an assembler function with the appropriate PASCAL or FORTRAN version. Here we can benefit from current research in computer science to develop appropriate processors for parallel and distributed processing and fast function calls.

Taking a paradigm and parameterize it for a specific task is called instantiation. The generic task instantiation is similar to the class instantiation in data modeling. In both cases we provide an incarnation of a concept, an incarnation of the object class or the generic task, annotated with instance specific parameters.

### 4.3. Local Processing: A Function Implementation Concept

When implementing EMS-functions we are used to split them into a sequence of individual procedures where each of them provides a certain functionality. Whenever a certain functionality is needed the appropriate function can be used provided that the problem is presented in the very same data structure. For example, the load flow calculation is divided into the following tasks: calculation of the linearization, formation of the sparse matrices, determination of the factorization sequence, factorization and backward substitution.

The partitioning of a function into functionality steps with specific data input and output format bears some disadvantages. Particularly, is is hard to restrict the execution on affected model parts or to express non-sequential algorithm steps. Below, a different way of partitioning the functions into smaller units is given. The approach relys on the following claim:

> Every EMS-functions consists of algorithm steps that use only information from the power system entity to be processed and some related entities.

We are not going to prove the claim formally. Instead we give some thoughts which support the claim: EMS-functions process objects in a well defined order. When processing an object, the current object and maybe related object are involved. Whenever an object is not related to the object to be processed it will be not touched. Thus, if an object has information about its own data and about its relation to other objects each algorithm step is local to that object.

The following two examples illustrate the locality of algorithm steps further:

1) A factorization step in load flow calculation involves only the node with its power flow equation, its adjacent nodes and all branches between them.

2) The basic step in a topology algorithm consists of checking whether physically related objects belong to the same node. An appropriate function can be implemented by testing all the neighbors of a certain entity and initiating the same test for these neighbors in a recursive manner.

Global processing is the commonly known programming style where procedures access the data model only in order to store and retrieve information. It separates the code from the data. The data model itself does not have processing capabilities. Local processing associates each algorithm step with a object or more accurately with a class of objects. For example, we can give the class node the knowledge of how to perform a factorization step or the class branch the knowledge of how to provide a linearization. Similar to the data of an object the processing step depends on the realization of the object and can be maintained at the same place as the object itself.

An algorithm basically consists of the processing steps done at the objects and a part that coordinates the sequence of of processing. In global programming this is done by stepping from object to object. However, this coordination is only necessary between related objects. For example, a node can be factorized after some related nodes have been factorized. This relation is provided by the function for optimal node ordering. Note that this ordering is actually a tree (and not just a linear sequence of nodes) where the parent represents a node which can be processed whenever all of its children have been processed [Tin.1].

Implementing this coordination at the object level allows to express only the sequential part of the algorithm and provides the opportunity for the underlying programming system to initiate parallel processing. For example, in the case of the factorization the processing can be started at all leaves of the optimal ordering tree in parallel. Coordination has to take place only when two nodes have the same parent.

There exists an analogy between data abstraction and local processing. Data abstraction provides a model in the form of some objects with some relation in between. This is a distributed data model. Local processing splits an algorithm up into object-oriented parts. The relation between these algorithm steps are modeled by some coordination mechanism, for example, by passing messages between objects.

### 4.4. Declarative Programming: A Modular Knowledge Representation Concept

Declarative programming provides the means for representing problem solving knowledge in a highly descriptive and modular form. As a result of this the knowledge is easily understandable and the programs are very flexible. At this point we give a formal definition of the declarative representation and point out the differences to the procedural representation employed in most conventional programming languages.

*Definition:* Declaratively represented knowledge is knowledge that has to be interpreted in some way in order to be applied.

*Definition:* Procedural knowledge prescribes a sequence of commands that specifies step by step how a problem must be solved. It only needs to be executed to be applied [Nil.1].

It is important to notice that the distinction between declarative and procedural knowledge depends on the representation rather than the knowledge itself. Generally all knowledge can be represented declaratively and procedurally. An example for this is the DC load flow in OPS-5, a declarative (rule based) language (see Appendix A). However, usually algorithms and basic operations are represented in procedural form whereas heuristic knowledge is represented in declarative form. The reason for this is that an algorithm represents the steps for a solution already in procedural form.

Procedurally represented knowledge is implicit. This means it is hidden in the code the procedure or algorithm. Each piece of knowledge is a fixed component of the code and cannot be modified or removed without adjusting the rest of the code. Declaratively represented knowledge, however, is explicit and can be represented in the form of a data structure. Therefore it is modular and directly accessible and understandable for the user. Because of the need of some interpretation applying declarative knowledge is usually more costly than procedural knowledge. Hence a frequently occurring specification is better represented procedurally.

As far as EMSs are concerned we have a well defined mathematical model on which theoretically all problems can be solved in an analytical form. However, in reality such calculations are either too complex to be performed in an on-line environment (stability analysis) or they involve combinatorial search (contingency selection, restoration). Here the application of heuristic knowledge is a necessity, either to simplify complex calculations or to guide the search. For this knowledge the application of a declarative representation is advantageous for the following reason: The heuristic knowledge is usually not well defined or not generally accepted. Therefore it is subject to frequent modifications. A declarative representation facilitates such modifications and allows to tune the system to the desired performance simply by changing and updating the heuristic knowledge.

Computer languages generally use mixed forms of representations with various degrees of declarative and procedural knowledge. Generally the higher the level of a language is, the larger is the extent of the declaratively represented knowledge. Programs in assembler languages are directly executable, and therefore do not contain declarative knowledge. Higher level languages such as Pascal, C, Fortran contain a declaration part that has to be interpreted by the compiler. AI languages such as Prolog and Lisp have the highest degree of declarative knowledge.

# 5. A Data Representation and Reasoning System for EMSs: A Proposal

In the previous chapter we have introduced four programming concepts that increase programming efficiency and software flexibility. Now we propose a data representation and reasoning system that exploits these programming concepts. For a particular implementation we have to choose a programming tool. Object-oriented programming is ideal since it provides data abstraction and local processing. In a first step we describe the object-oriented model, i.e. the objects and their relationships. Then we discuss the implementation of some basic programming building blocks needed in all EMS analysis functions. Finally we explain how our system can support rule based expert systems.

## 5.1. Object-Oriented Programming: An Implementation Tool

Object-oriented programming is a recent programming style that makes use of the concepts of data abstraction and local processing. In object-oriented programming the object is the basic data model unit. The procedures that define the interface between the objects and regulate the access to the internal information are called messages or methods. Messages are not only used for accessing information stored inside the objects, but in object-oriented programming message passing is the only way of information processing.

There are many different languages that belong to the category of object-oriented languages. The most prominent examples are Smalltalk-80 [Gol.1], Object Pascal [Schm.1], Objective-C [Cox.1], and also various extensions to the language Lisp such as Flavors for FranzLisp [Lay.1] and Common Loops for CommonLisp. With the exception of Smalltalk all the above languages are object-oriented extensions built on top of existing programming languages. The idea behind the whole spectrum of object-oriented languages is the same, but the notation is very different from one language to another. Sometimes this notation is very complex and confusing. Therefore we adopt here a kind of pseudo-code, that is mnemonic and therefore easily understandable. This code can readily be translated into any object-oriented programming language.

For the task of sending a message to a particular object the following notation is used:

> [*object-identifier.message-name(parameters)*]
> e.g.
> [SHCAP_1.React_Power]

This piece of code sends the message *React_Power* to the object *SHCAP_1* which is the identifier for a particular shunt capacitor. This tells the object *SHCAP_1* to return the value of its reactive power flow. How the object produces the reactive power flow must be defined in the procedure *React_Power* that is associated with the class of all shunt capacitors.

First a definition of the class of all shunt capacitors is needed:

```
class Shunt_Capacitor:
        superclass:         Physical_Element
        subclasses:         Switched_Capacitor, Fixed_Capacitor
SHCAP_1 = make-instance (Shunt_Capacitor)
```

This defines the class *Shunt_Capacitor* a a subclass of all physical elements. The class itself has two subclasses, the switchable and the fixed capacitors. The second piece of code defines the identifier *SHCAP_1* to be an instance of the class *Shunt_Capacitor*. As an instance of a class it inherits all the methods defined for the class.

For this class the message *React_Power* can be defined in different ways. The simplest possibility is that the message *React_Power* just returns the value of the reactive power flow that is stored somewhere inside all the objects of class *Shunt_Capacitor*. Another possibility is that the message defines how the reactive power flow has to be calculated from other values defined for the object:

Message React_Power for class Shunt_Capacitor:

```
React_Power ()
{ Q = [SELF.Voltage] * [SELF.Voltage] / (2 * PI * [NETWORK.Frequency] * [SELF.Capacitance])
  return (Q) }
```

The identifier *SELF* in the above message is specially defined in object-oriented programming. It always refers to the object that receives the particular message. In the message above another message is sent to the network in order to obtain the frequency and then the reactive power flow is calculated from the voltage and the capacitance.

It is important to note that the message above is just one possible way of obtaining the reactive power flow. Another possibility is that the message contains a request for the information from a data base. This can be useful for less frequently used specifications. However, for the sender of the message the particular implementation is not visible and also of no importance as long as the desired result is obtained, i.e. the reactive power flow is returned. This again illustrates the concept of data abstraction. The message *React_Power* can be modified without affecting the rest of the software.

Object-oriented languages support property inheritance, a service that enhances data abstraction and reduces redundancy of code. In our example, property inheritance makes it possible to send the message *React_Power* also to objects of the classes *Switched_Capacitor* and *Fixed_Capacitor*. Since they are defined as subclasses of *Shunt_Capacitor* they automatically inherit all the messages defined for the superclass *Shunt_Capacitor*.

It is particularly useful that different classes of objects can react quite differently to the same message (polymorphism). Therefore is is possible to define the message *React_Power* with exactly the same name for objects that represent elements such as loads or generators. Obviously there the reactive power flow has to be determined in a quite different way. However, the sender of the message (for example a node that wants to determine the total reactive power injection) does not care whether the element is a generator, load or a shunt capacitor. This feature of object-oriented programming corresponds to operator and function overloading in other languages. For example in ADA and C++ [Str.1] arithmetic operators can be overloaded such that they can be used for user defined objects such as complex numbers, vectors and matrices.

## 5.2. Definition of the System

### 5.2.1. The Object-Oriented Model

#### 5.2.1.1. The Identification of the Objects

In a first step the objects of a data model for EMS functions must be identified. From the previous discussions it is clear that the objects should correspond to the entities in the power system. There are three different kind of entities: physical entities, conceptual entities and generic term entities. Since we have a one to one correspondence between this entities and the objects of the data model we name them accordingly.

**Physical Objects**

Physical objects correspond to the physical entities in a power system. Examples for physical entities are:

| | |
|---|---|
| line | series compensator |
| 2-winding transformer | shunt compensator |
| 3-winding transformer | bus |
| generator | switch (breaker) |
| load | measurement point |

The list above may not be complete and depends on the particular power system. Each of the objects that corresponds to a physical element of the power system has a defined set of messages over which the information can be accessed. Characteristic for these objects is that they are static in the sense that the existence of a particular object does not change. Only the information contained in them changes

and has to be updated constantly, for example, measurement values, breaker status, etc.

**Conceptual Objects**

The conceptual objects consist of sets of objects that are instances of possibly different classes. The set satisfies certain properties that characterize the conceptual objects.

Examples: node
branch
observable island
network group

Characteristic for these objects is that existence of a particular instance depends on the attributes of the physical objects. Therefore these objects are dynamic and have to be updated whenever certain changes occur in the network. This activity is generic in EMS and will be described later.

> For example, a node is defined as the set of entities that are connected without impedance between them. The entity that connects without impedance is the switch. Therefore the node depends on the status of the switches. Whenever a switch is opened a particular node may be split into two nodes. Whenever a switch is closed two nodes may coalesce.

The conceptual objects are used for a particular EMS function. The nodes and branches, for example, are needed for the load flow calculations. The nodes and different measurement-branches and observable islands are needed for the state estimation.

**Generic Term Objects**

Examples: station
area

These objects are similar to the conceptual objects with the difference that their existence is static. They consist of a collection of physical objects. Certain properties of the generic term can be derived from the physical entities. For many power system analysis functions these generic term entities are of minor importance and therefore will not be discussed further.

#### 5.2.1.2. The Relations between the Objects

In this object model relations between the objects must be modeled. These relation correspond to the relations between the elements of the power system model. A power system consists of physical elements such as lines, transformers, generators, loads, compensators, bus-bars etc. These elements are physically connected to each other. In power system also the conceptual entities such as stations, nodes, observable groups play an important role.

Therefore in an EMS the following relations between objects are important:

- class - class : classification (subclass, superclass)
- class - instance : instantiation (class membership)
- instance - instance : conceptual objects (sets)
connectivity

Classification, instantiation and the mechanism of inheritance are provided by the object-oriented programming language and have been discussed above. The relation between the conceptual objects and the corresponding physical objects and the connectivity between objects is discussed below.

**Conceptual Objects**

The conceptual objects are closely related to the objects representing the physical entities. The information they have to provide can be obtained from the physical objects they depend on.

> For example, a branch may represent a line or a 2-winding transformer. When the line gets switched off the branch ceases to exist. However, while it is in existence it has to provide the power flow into both branch ends as a function of the node voltages. The load flow function itself is not concerned how a particular branch produces the branch power flow and its

linearization. However, the branch object contains the knowledge of how to obtain the required information from the physical objects it depends on. This knowledge is contained in messages that are sent to the particular physical objects.

As far as the representation of conceptual objects is concerned it can be handled by defining a special class conceptual objects. All object classes that need the special capabilities of conceptual objects have to be defined as subclasses. The conceptual object must have an property that consists of a set of its members or objects it depends on. Basic operations can be handled by defining a class *Set*. The class set and other useful general purpose classes are offered in class libraries of some object-oriented languages [Cox.1]. However, basic set handling is not enough for the formation and the maintenance of conceptual sets. Whenever an object is included in a conceptual object all the properties have to be updated. The inclusion of object into conceptual objects and the combination of two conceptual objects are basic generic tasks that will be described in 5.3.2.2.

**Connectivity**

The second important relationship between instances of various classes is the physical connectivity. This connectivity can be characterized by an attribute of an element containing all the elements it is connected to. Connectivity is a bidirectional relationship, i.e if element A is connected to element B, then element B is connected to A. This connectivity can again be modeled by a set. Then the system should enforce the consistency between the connectivity sets of the elements. Whenever an element A belongs to the connectivity set of an element B then element B must also belong to the connectivity set of element A. Connectivity sets can be defined as a new class that is a subset of the class set and therefore inherits all the set operations.

#### 5.2.1.3. Solutions to the Frame Problem

We have pointed out that for many tasks in power systems a temporal analysis is necessary. The problem of representing the facts that change as well as those that do not is known in the area of AI as the frame problem [Ric.1]. One rather obvious solution to the frame problem is to associate with each time step a data model as described above. However, this solutions is not very efficient since only a small subset of the data changes from one time step to the next.

Another possibility is to hide the time representations inside the particular objects. The only attributes that evolve over time need a special representation. A specific value would be provided by specifying the access function and additionally explicitly or implicitly (the appropriate world scope is restricted to a given time) the time.

The disadvantage is that because of this additional specification the surface area between data and functions is increased. But there is hardly any way to avoid this problem if the behavior of the power system over time has to be analyzed. The advantage of this representation again is that the implementation is hidden from the user of the object. This implies that new solutions and different implementations of the time tags do not affect the rest of the system.

### 5.2.2. The Implementation of the Generic Tasks

Below, we define a set of basic generic tasks related to EMS analysis functions. They serve as building blocks which are basic enough to be provided directly by the objects, yet complex enough to provide essential benefits such as better software manageability, more efficient programming, preservation of distributed problem solving.

Beside the generic tasks described, object-oriented programming itself provides basic object accessing capabilities such as read and modify object properties. Compared to the access of the data model in current EMS-function even these primitives have a higher level since they are independent of the implementation of the objects.

#### 5.2.2.1. Generic Task: Graph Traversal

Every algorithm on the power system model consists of processing steps that are local, i.e. affect only an object and some directly related ones. These local processing steps have to be performed on some set of objects, sometimes in a specified order. This can be represented by a graph or a tree whose nodes are the objects and whose branches stand for the relations between them. Therefore graph traversal is the workhorse of every algorithm: Graph traversal is a generic task.

**Definition**

Walk through a set of objects of the power system model in a defined order provided in the form of a tree or more generally in the form of a graph and perform a specified operation on each object.

Graph traversal as a generic task needs two parameters. The first one is the graph to be used, i.e. the objects that correspond the the nodes of the graph and relation between them that corresponds to the branches. In the case of a tree additionally the direction of processing has to be defined as described below.

**Discussion**

The realization of the generic task requires the execution of the following steps at each object:

a) accept a message

b) perform the specified operation

c) send messages to all the related objects, specified by the parameter 'graph'.

Note that b) and c) can be interchanged. In case of the sequence a), c), b) the objects get the messages as fast as possible. This enables the highest degree of parallelism however requires also the highest degree of independence of the operation to be performed, the operation is totally local. This may be useful e.g to set some markers at each participating object.

In case of the sequence a), b), c) the operation at an object is performed before sending out the messages. This sequence has to be applied e.g. in a topology algorithm. The current object is inserted into the appropriate node before the physically connected object receive the message to join the node. Otherwise a new node would be created for each object.

**Implementation**

The generic task graph traversing can be implemented in various ways. At this point we want to discuss a particular implementation of the graph traversal algorithm where the graph is a tree. The tree specifies the dependencies for the processing of the nodes, i.e. which nodes have to be processed before a particular node can be processed. An example for such a tree is the optimal ordering tree used for the triangular factorization of the Jacobian in a load flow calculation [Tin.1].

A tree is defined as a set of nodes where each node has a property 'children' that contains the set of its children. By the definition of the tree each node can be the child of only a single node. The node that has no parents is called the root of the tree and is used as a reference for accessing the tree. On a tree the processing can be done in two directions:

- Each child node has to be processed before the parent node. In this case processing starts at the leaves of the tree. This will be called backward processing.
- The parent node has to be processed before the children. In this case processing starts at the root of the tree. This will be called forward processing.

Tree traversing can now be implemented on the object model by defining the following message for the class of objects that can be nodes of a graph:

Message Graph_Traversal for class Node:

```
    Graph_Traversal (children, procedure, direction);
    { if direction = forward then
          [SELF.procedure];
```

```
        send-all ( [SELF.children], Graph_Traversal (children, procedure, direction) );
    else
        send-all ( [SELF.children], Graph_Traversal (children, procedure, direction) );
        [SELF.procedure];
}
```

This is a recursive procedure. In the case of forward processing the processing of an object is done before sending the message, in the case of backward processing the message is sent to all the children and only after the procedure has been executed. The parameter children defines the name of the properties where the next element can be found. The parameter procedure the name of a message that must be defined for the nodes to be processed.

The function send-all needs more explanation. The second parameter is the name of a message that is sent to all the elements of the list given as the first parameter. By definition the messages sent to all the elements are independent from each other. In a multi-processor environment they can be executed in parallel. In a single processor environment the functions send all is simply the processing of a list, i.e. send the message to the first element, then to the next and so on until the list is finished.

It is important to notice that the user does not have to worry about these issues. They affect only the implementation of the generic task 'Graph_Traversal' that is offered by the system.

The implementation of the generic task 'Graph_Traversal' for general graphs is very similar. In a graph objects are processed in an arbitrary order. Therefore the specification of a direction is not necessary. In a graph the traversal algorithm can be initiated by sending the message 'Graph_Traversal' to any node. In graphs a marker is needed to insure that each object is only processed once.

An example how this generic task can be applied is the triangular factorization. Suppose optimal ordering has been performed and has established a property 'Optimal_Order' for all the buses. 'Optimal_Order' contains a list of all the nodes that have to be factorized before the particular bus. This defines an optimal ordering tree as described in [Tin.1]. Then the factorization can be implemented by sending the message 'Graph_Traversal' to the root node:

```
[ Root.Graph_Traversal (Optimal_Order, Factorize, backward) ]
```

The message 'Factorize' must be defined for all buses and performs the local factorization step for a particular bus.

#### 5.2.2.2. Generic Task: Conceptual Object Manipulation Routines

The conceptual objects node, branch, network island and observable group have to be built up and maintained continuously. Since the conceptual objects play an important role in the power system model the basic operations required to build up and maintain conceptual objects are defined as generic.

**Definition**

> The following basic conceptual object manipulation routines are required: Create and delete a conceptual object, and insert and remove an element from its conceptual object.

There are derived operations such as merging two conceptual objects and splitting up a conceptual object. These can be implemented by using the basic operations defined above. The task of creating a conceptual object is offered by the object-oriented language: If the conceptual objects are defined as a class the function 'make_instance' of object-oriented languages performs this task. Also the function 'delete' of an object is provided by object-oriented languages.

**Discussion**

Conceptual object properties are provided either in the form of some procedure to be executed whenever the property is required or the property is stored as some privately defined attributes. In the first case no adjustment is necessary when conceptual object manipulation routines are performed. In the second case each time conceptual objects are modified the appropriate object properties have to be updated. The information how this updating is to be performed depends on the particular conceptual object and therefore can be defined as message for the particular conceptual objects.

For the basic conceptual object manipulation routines conventional set operations have to be used, for example to check whether an object belongs to the set or not. Such operation are usually provided by object-oriented languages. For example Smalltalk and Objective-C offer basic objects such as lists and sets with the corresponding operations.

**Implementation**

In order to show how the basic conceptual object operations can be implemented in object-oriented programming the code for the tasks 'Insert_element' and 'Merge' is given and discussed. Each conceptual object must have and attribute 'Elements' that consists of the set of the elements and a routine to update the properties of the conceptual object. Since it is usually not necessary to update all the properties if a particular element type is included or removed the procedure can parameterized by the element.

Message Insert_Element for class Conceptual_Object:

```
Add_Element ( element )
{ if not member (element, [SELF.Elements]) then
    include (element, [SELF.Elements]);
    [SELF.Update(element)]; }
```

Message Merge for class Conceptual_Object:

```
Merge ( conceptual_object )
{ for each element in [conceptual_object.Elements] do
      [SELF.Add_Element(element)];
  delete (conceptual_object);
}
```

An example for the use of these basic operations is the topology algorithm. The conceptual objects this algorithm builds up are the nodes and the branches. To start the algorithm a first node has to be initialized. The following function then expands the node by including all the elements that are connected with no impedance to some element of the node. In order to define this function again the graph traversal algorithm can be used:

Message Expand for class Physical_Objects:

```
Expand (node)
[SELF.Graph_Traversal (connected_no_impedance, Add_to_Node (node)) ]
```

Message Add_to_Node for class Physical_Objects:

```
Add_to_Node (node)
[node.Insert_Element(SELF)]
```

The message 'Expand' can be sent to the first element of a node with the parameter 'node', the identification of a newly created node. This expands the node according to its key property 'connected_no_impedance' to its full size.

The graph traversal algorithm together with the message 'Expand' can now be used to build up all the nodes and the branches. This time traversing is performed on the node level. The complete procedure for determining all nodes and branches is now:

```
{ FIRST_NODE = make-instance ( Node );
  [FIRST_ELEMENT.Expand(FIRST_NODE)];
  [FIRST_NODE.Graph_Traversal(Neighbor_Nodes, Build_Branches)] }
```

The message Build_Branches of class node must include the following tasks:

- Build all the branches of a particular node, i.e. find all the elements of the node that have an impedance and make instances of the appropriate branches.
- Initialize all the neighboring nodes and include their identifiers in the list Neighbor_Nodes. This means that the list of neighboring nodes is built up during the graph traversal algorithm.

The algorithm discussed builds the nodes and branches from scratch. For update algorithms, however, the same basic building blocks can be used.

**5.2.2.3. Generic Task: Element Power Flow Equation**

Power systems are flow networks. Whether we are interested in the system state based on observed quantities or in the system state based on a set of specified demand or supply quantities, we always deal with the complex flow between nodes, the power flow equations. A power flow equation describes the flow relation between a set of neighboring nodes (n-port). In the simplest and most common form this is represented by a 2-port branch. Although the elements representing these relations may vary, for practical reasons they have to be all alike. Otherwise they cannot be directly implemented into standard EMS-functions. For example, the addition of a model for a line with a distributed load asks for changes in all existing EMS-functions dealing with branches or the changes have to be circumvented by modeling the physical entity artificially or approximatively. The main reason for these problems are the unnecessary strong coupling between the physical model and the functions using the model.

The complex power flow into an element as a function of the state variables is needed in all flow network functions. The particular function does not need any information of the implementation of an element as long as the element power flow and its approximation is provided. The flow network functions then are independent from the element implementation. The element power flow is a generic task.

**Definition**

> Each element provides the complex power flow and an appropriate approximation as a function of the state variables, usually the complex voltages at the ports and the transformer and capacitor taps.

Most implementation of flow functions use a Newton-Raphson type algorithm to solve the nonlinear power flow equations. In this case the approximation is the linearization. Since this approximation is the most usual one it will be considered in the following.

**Discussion**

For the calculation of the load flow, for example, each branch must be able to provide its flow equation. But since a branch is only a conceptual object, it may exist sometimes and may not exist at other times depending on the current topology. Therefore, it is necessary to place the power flow model into the physical objects. The conceptual objects themselves contain only a reference to the physical objects on which they depend.

From the physical objects listed above all the ones that contain impedances must provide the appropriate power flow equations. These objects are: lines, 2- and 3-winding transformers, series compensators and shunt compensators. Additionally, it is possible to include such objects as loads and generators. Usually a load is modeled by specifying a constant P and Q. However, it is also possible that a load is represented by specifying the complex power as an arbitrary function of the voltage. In this case also the load must provide the power flow and its approximations.

**Implementation**

For the implementation three different types of models for the physical elements are discussed in more detail and some examples are given. These are the one-ports: loads and shunt compensators, the two-ports: lines, series compensators and 2-winding transformers and the n-ports: 3-winding transformers.

The modeling of the one-ports is the simplest case. The power flow into the one-port depends only on the voltage magnitude at the single port. Therefore objects corresponding to one ports must be able to return the following functions:

$$P(V)\,,\quad Q(V)\,,\quad \frac{\partial P}{\partial V}(V)\,,\quad \frac{\partial Q}{\partial V}(V)$$

For example, for a shunt capacitor we have the following equations:

$$P(V)=0\,,\qquad Q(V)=\frac{V^2}{\omega C}\,,\qquad \frac{\partial P}{\partial V}(V)=0\,,\qquad \frac{\partial Q}{\partial V}(V)=\frac{2V}{\omega C}$$

Another example is the voltage dependent load. Suppose a polynomial expression is required for the load then the equations are:

$$P(V) = A_0 + A_1 V + A_2 V^2 , \qquad Q(V) = 0$$

$$\frac{\partial P}{\partial V}(V) = A_1 + 2A_2 V, \qquad \frac{\partial Q}{\partial V}(V) = 0$$

Each one port object therefore must have messages defined that provide $P$, $Q$, $\partial P/\partial V$ and $\partial Q/\partial V$. For example, the message P for the object class 'Voltage_Dependent_Load' can be implemented as follows:

Message P for class Voltage_Dependent_Load:

```
P ()
{ P = [SELF.A0] + [SELF.V] ( [SELF.A1] + [SELF.A2] * [SELF.V] )
  return (P) }
```

The modeling of the two-ports is not very different except that each two port object must have more messages in order to provide the element power flow and its linearization. The power flow into both sides depends on the voltage magnitudes and the voltage angles on both sides:

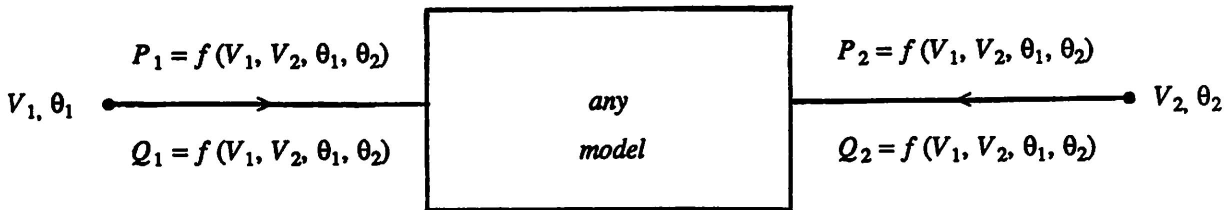


Fig. 1 2-Port Model

The two-port objects must be able to provide the functions:

$$P_1 = f(V_1, V_2, \theta_1, \theta_2) , \qquad P_2 = \cdots , \; etc.$$

$$\frac{\partial P_1}{\partial V_1}(V) = \frac{\partial f}{\partial V_1}(V_1, V_2, \theta_1, \theta_2), \qquad \frac{\partial P_1}{\partial V_2}(V) = \; \cdots , \; etc.$$

These functions must be provided by all two port objects and can be implemented as messages.

For example, if the two port object is a two-winding in-phase transformer the appropriate power flow equation is given by:

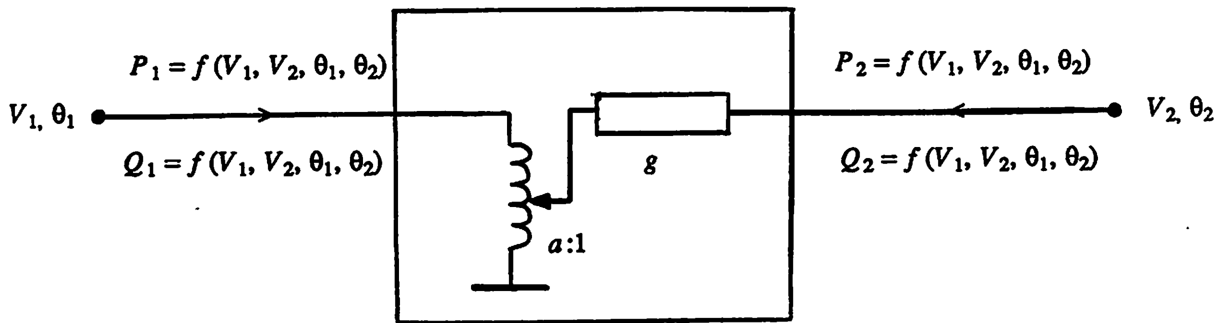


Fig. 2 In-Phase Transformer as 2-Port

$$P_1 = g \left[ \frac{V_1}{t} \right]^2 - g \frac{V_1}{t} V_2 \cos(\theta_1 - \theta_2) - b \frac{V_1}{t} V_2 \sin(\theta_1 - \theta_2)$$

$$\frac{\partial P_1}{\partial V_1} = 2g \frac{V_1}{t} - g \frac{V_2}{t} \cos(\theta_1 - \theta_2) - b \frac{V_2}{t} \sin(\theta_1 - \theta_2)$$

These functions can easily be coded as messages and then represent part of the two port model. It is important to note that the implementation of the function $P_1(V_1, V_2, \theta_1 \theta_2)$ is not visible outside the object. The message [Two_Port.P1] returns the real power flow into the two port independent from the two-port model. Therefore this message works for lines with different models as well as 2-winding transformers and series capacitors.

The extension of the generic task power flow to n-port models is now obvious. The power flow into a port of the n-port depends generally on the complex voltages at all n-ports. Therefore messages are needed to provide all elements of the form:

$$P_i = f_i \ (V_k, \theta_k \colon k=1..n)$$

$$\frac{\partial P_i}{\partial V_k}(V_k, \theta_k \colon k=1..n), \quad \frac{\partial P_i}{\partial \theta_k}(V_k, \theta_k \colon k=1..n), \ \cdots$$

The 3-winding transformer is the only one of the physical objects that requires such a model. Another application for this n-port model is external network modeling.

### 5.2.3. Expert System Tool

Rule based expert systems are a very common way to exploit the advantages of declarative programming. Each rule specifies the conditions for a particular action to take place. Examples for general purpose expert system tools are OPS5 [For.1] and KEE [Fik.1]. Both systems have a data model that is frame based, a structured form of knowledge representation that is very similar to an object-oriented data model. The frames of OPS5 are used only as a data model. All the necessary processing must be described in declarative form (rules). KEE provides some reasoning services such as inheritance, default values and procedural attachments.

In the object-oriented data model, however, arbitrary procedures can be attached to the objects in the form of messages. Declarative and procedural representations of the problem solving knowledge can be mixed as appropriate. Additionally, the object-oriented model in power systems has a special structure (locality) that can be exploited by the expert system. For example, the rules can be directly associated with the objects and therefore variables are not needed. In order to show how a rule based system can make use of this special structure we discuss a rule based, data directed expert system. Data directed reasoning or forward chaining refers to the direction of inference, i.e. from the data to the goal (see Appendix B).

#### 5.2.3.1. Production Rules

Production rules are usually specified in the IF-THEN format, where the IF-part specifies the conditions that must be satisfied in order to execute the THEN-part:

| | | |
|---|---|---|
| IF | {*condition 1*} | AND |
| | {*condition 2*} | AND ... |
| THEN | {*action 1*} | AND |
| | {*action 2*} | AND ... |

In the object-oriented model the only way to access and modify properties of objects is by messages. The conditions and actions will involve messages defined for the objects of the model. This means that a rule describes the conditions for an action to be executed on a particular object. Therefore the rule can be associated with that object. By the local properties of the data model the conditions for the action on an object depend on the object itself and some related objects. Making use of the facilities of object-oriented programming this particular object can be referred to by the variable 'SELF'. Also all its properties and the properties of the related objects can be accessed by means of the variable 'SELF':

E.g. [SELF.Property]
[ [SELF.Relation].Property ]

**a) Condition Part**

Each condition must be a predicate, i.e. either a message returning TRUE or FALSE or an expression involving such messages and boolean operators. The only constraint required for these messages is that they are passive. This means that they only access properties of the objects, but do not modify them. However, the messages may involve complex processing and calculations. Besides the standard boolean operators such as 'AND, NOT, <, >, =', etc. the two logical operators 'THERE EXISTS' and 'FOR ALL' are useful for messages that operate on sets. For example, consider the following rule that is associated with the class node:

> If there is a closed earth isolator
> then the node is earthed.
>
> IF THERE_EXISTS [SELF.Element]
>   SUCH_THAT ( [Element.type] = Earth_Isolator AND [Element.status] = closed )
> THEN [SELF.Earthed] := TRUE

**b) Action Part**

The action part consists of messages of the object that is associated with the rule, messages of the related objects as described before and possibly some messages of objects that serve the particular expert system. An example for the latter is the object that contains the contingency list in an expert system for contingency selection (see example below).

**5.2.3.2. Inference Engine**

The inference engine is responsible for the interpretation of the declarative knowledge given in the form of rules. This is usually done by the recognize-act cycle that involves the following steps for a forward chaining system:

1. Find all the rules whose condition parts are all satisfied. The set of these rules is called the conflict set.
2. Select one rule of the conflict set to be fired (conflict resolution).
3. Perform the actions specified in the action part. In the object-oriented system this means send the messages specified in the action part to the object itself and to the related objects.
4. Goto 1.

The two tasks updating the conflict set and conflict resolution need more discussion. Whereas the first task poses an implementation problem since it involves a search over the involved objects or rules, the second task determines the control strategy and therefore the reasoning behavior of the expert system.

**a) Updating the Conflict Set**

The conflict set contains all the rules whose IF-part are satisfied. Each time the THEN-part of a rule is executed some object properties are modified. This in turn may affect some rule conditions. Hence, each time the conflict resolution set has to be adjusted. This can be done using one of the following strategies:

a) Redetermine the conflict set by testing the IF-parts of the rules starting from scratch.

b) Observe which object properties have been changed and find out which rules are using these properties within the IF-part. Reevaluate the condition part of these rules and add or delete appropriate rules to or from the conflict set.

While the first approach is straightforward but time consuming, the second approach on relies on the fact that an object property is referenced only by rules belonging to that object or to related objects. The rules are associated with the objects and therefore each object (or object class) can determine the rules that are satisfied by itself (locality). By definition the messages in the condition part are passive, i.e. do not modify properties. Therefore for each rule of an object it can be determined from the messages in the condition part on which objects it depends. These dependencies can be determined at compile time. An efficient way of updating the conflict set therefore is to assign messages to those objects

whose properties affect a certain rule. These messages are automatically invoked whenever a property is changed by the expert system. Again we have made use of the special structure of the object-oriented model.

**b) Conflict Resolution**

The strategy for conflict resolution determines which rule of the conflict set is chosen to be executed. The conflict resolution strategy determines the reasoning behavior of the expert system. There are numerous possibilities:

| | |
|---|---|
| Rule ordering: | The rules are arranged in a priority list. The triggering rule appearing earliest in the list is used. |
| Data ordering: | The rules are chosen according to a priority list established for the condition parts. |
| Size ordering: | The rule with the toughest requirements, i.e., longest list of constraints, is fired. |
| Recency ordering: | The rule with the antecedents which have been established most recently is executed. |
| Context limiting: | This strategy tries to reduce the likelihood of a conflict by separating the rules into groups. At each time, only some of the groups are active. An additional procedure activates and deactivates the groups. |
| Specifity ordering: | If the antecedents of one rule are a subset of the ones of another rule, the more specific rule is chosen. |
| Parallel processing: | No rule is chosen; all applicable rules are processed in parallel. |

Any of these strategies can be implemented within the object-oriented system. However, it is useful to exploit the locality and the structure of the model. For example, execute all the rules satisfied for a particular object or object class first, before going to the next object. In this case priorities can be associated to the object classes. Another possibility is to fire all the rules of a class in parallel.

# 6. Use of the System: Three Examples

In this chapter it is demonstrated how the tools defined in Chapter 5 facilitate the implementation of power system analysis functions. The load flow function has been chosen as the basic example of an analytical function. State estimation and bad data detection have been added in order to show how matrix operations (multiplication, sparse inverse) can be broken down into local steps and be implemented with the basic generic tasks. The last example, contingency selection has been chosen because of its heuristic nature. It shows how the rule based system defined above can be used to represent heuristic knowledge in the form of rules and how the expert system can work on the very same data model.

## 6.1. Load Flow Calculation

The basic elements of the load flow model are the conceptual objects: nodes and branches. The creation and the maintenance of the nodes and branches are performed by algorithms that exploit the basic routines for the manipulation of the conceptual objects. Here we assume that the classes 'Branch' and 'Node' are defined and that the relations between them are established such that they form a network.

### The Representation of the Jacobian

The basic step in a Newton-Raphson load flow is the solution of the linearized system:

$$\begin{bmatrix} J \end{bmatrix} \begin{bmatrix} \Delta\theta \\ \Delta V \end{bmatrix} = \begin{bmatrix} \Delta P \\ \Delta Q \end{bmatrix}$$

where $\Delta P$, $\Delta Q$ are given and $\Delta\theta$, $\Delta V$ the state variables to be calculated.

The elements of the Jacobian can be directly related to the branches and the nodes of the model. The Jacobian is reordered such that the elements belonging to the same node are grouped together. Then the 2 by 2 diagonal blocks are of the form:

$$\begin{bmatrix} \frac{\partial P_k}{\partial \theta_k} & \frac{\partial Q_k}{\partial \theta_k} \\ \frac{\partial P_k}{\partial V_k} & \frac{\partial Q_k}{\partial V_k} \end{bmatrix}$$

The diagonal blocks only involve quantities from a single node (Node k) and therefore are associated with that node. The conceptual objects of class 'Node' therefore have messages defined that provide these 4 elements.

Analogously the 2 by 2 off-diagonal blocks are of the form:

$$\begin{bmatrix} \frac{\partial P_{ki}}{\partial \theta_i} & \frac{\partial Q_{ki}}{\partial \theta_i} \\ \frac{\partial P_{ki}}{\partial V_i} & \frac{\partial Q_{ki}}{\partial V_i} \end{bmatrix}$$

These blocks involve quantities from exactly two nodes (k and i) that have a branch between them. Therefore they are associated with that branch and the corresponding conceptual object has messages defined for those 4 elements.

The elements of the Jacobian associated with the conceptual objects must be derived from the linearized power flow that is provided by the physical objects. This is done by the messages for $\partial P_k / \partial \theta_k, \cdots$, etc. of the conceptual objects.

A few examples: If a branch corresponds to a line then no calculation is necessary. The linearized power flow provided by the generic task of the physical object line corresponds to the elements $\partial P_{ki} / \partial \theta_k, \cdots$ of the Jacobian. The message of the object for the branch contains a reference to the object that represents the line.

If the physical object is an n-port then more than one branch is created in the load flow model. Still the Jacobian elements to be provided by these branches correspond exactly to some of the elements of the linearized power flow of the n-port and can be referenced.

For a node the construction of the Jacobian elements is not quite as simple. The power flow into a node is equal to the sum of the power flows into all of the connected n-ports. Therefore the Jacobian elements for the node have to be calculated as the sum of the linearized power flows from the physical elements that are connected to the node.

The references in the messages of the conceptual elements must be defined and maintained by the message 'Up_date' that is defined for all conceptual objects.

**The Solution of the Linearized Power Flow Equation**

The linearized power flow in the Newton-Raphson algorithm

$$Jx = y$$

is usually solved by triangular factorization of the Jacobian. This requires the following 4 steps:

1) Optimal ordering
2) Triangular factorization $J = LU$
3) Forward substitution $Lz = y$
4) Backward substitution $Ux = z$

In the following an implementation of these 4 steps in the object-oriented model is outlined. Since 2 by 2 blocks of the Jacobian can be associated with the nodes and the branches of the network, such 2 by 2 blocks are considered as a single element. Therefore the elements of the Jacobian and the elements of the vectors are 2 by 2 respectively 2 by 1 blocks.

**1) Optimal Ordering**

Usually local optimal ordering is performed before factorizing the Jacobian in order to reduce the number of the fill-in terms. Local optimal ordering is a sequential ordering of the rows and the columns such that at each step of the factorization the next row and column is chosen so that it gives the minimum number of fill-ins in that step size. Notice that in the object-oriented representation no reordering is actually performed, rather the optimal ordering defines in which sequence the nodes (node equations) have to be factorized. Therefore the optimal ordering algorithm therefore establishes a property 'Optimal_Order' for each node. This defines the node that should be factorized next.

The generic task graph traversal can be used to walk through the nodes to establish the optimal ordering:

Graph_Traversal ( Symbolic_Factorize, Least_Number_of_Fill-ins )

The message 'Symbolic_Factorize' for the class Node performs the symbolic factorization of the node by introducing the appropriate cross branches (fill-ins) and removing its branches. The message 'Least_Number_of_Fill-ins' goes through the set of not yet processed nodes and finds the one with the least number of fill-ins. This message again can be implemented by making use of the generic task graph traversal. The message for the optimal ordering must be sent initially to the node of the network that produces the least number of fill-ins. The complete procedure for optimal ordering is therefore:

[ [Any_Node.Least_Number_of_Fill_ins] . Graph_Traversal ( as above ) ]

This initializes the search for the node with the least number of fill ins starting at 'Any_Node'. Then the optimal ordering starts at the node whose factorization step produces the least number of fill-ins.

**2) Triangular Factorization**

This procedure consists of factorizing the node equations in the sequence determined by the optimal ordering procedure. The graph traversal message can be sent to the first node:

[ FIRST_NODE.Graph_Traversal (Factorize, Optimal_Order) ]

The message 'Factorize' performs a factorization step at each node. The procedure is local: It involves only the node itself, the neighboring nodes and the branches between them. It consists of the following steps:

Node: The diagonal element $d$ that is stored with the node is the pivot of the factorization step. It is not changed.

Branches: The branches have attributes for the upper and lower triangular elements $l_b$ and $u_b$. The upper triangular elements of the branches incident into the node have to be updated as follows:

$$u_b \leftarrow d^{-1} u_b$$

Adjacent Nodes: The adjacent nodes are the nodes that are connected to the node to be factorized over a single branch. The matrix elements of the adjacent node is called $d_a$ and the elements of the corresponding branches are $l_b$ and $u_b$. The update equation for the adjacent nodes is:

$$d_a \leftarrow d_a - l_b u_b$$

Cross Branches: The branches between two adjacent nodes are called the cross branches. If such a cross branch does not exist between two adjacent nodes then the message 'Factorize' must create a temporary branch that is only used for the triangular factorization. This branch corresponds to a fill-in term in the factorization of the sparse matrix. The elements of the cross branches are labeled $l_c$ and $u_c$. They are updated as follows:

$$l_c \leftarrow l_c - l_1 u_2$$

$$u_c \leftarrow u_c - l_2 u_1$$

where the labels 1 and 2 denote the branches from the node to be factorized to the nodes of the cross branch.

**3) Forward Substitution**

The forward substitution solves the system

$$L z = y$$

It can be implemented by the graph traversal message since a step at the node is local.

[ FIRST_NODE.Graph_Traversal ( Forward_Step, Optimal_Order ) ]

The local procedure 'Forward_Step' updates the z-values of the node to be processed ($z$) and of the adjacent nodes ($z_a$) as follows:

Node itself: $z \leftarrow d^{-1} ( y + z )$

Adjacent nodes: $z_a \leftarrow z_a - l_b z$

**4) Backward Substitution**

The backward substitution solves the system

$$U x = z$$

The implementation with the graph traversal algorithm is similar, except for the direction that is this time backward.

[ FIRST_NODE.Graph_Traversal ( Backward_Step, Optimal_Order, backward ) ]

The local procedure 'Backward_Step' involves the following updating steps:

Node itself: $x \leftarrow z$

Adjacent nodes: $z_a \leftarrow z_a - u_b\, z$

**Fast Decoupled Load Flow**

For the fast decoupled load flow the following assumptions are made:

a) $\theta_{km}$ small $\rightarrow$ $\sin(\theta_{km}) = \theta_{km}$
$\cos(\theta_{km}) = 1$

b) $g_{km}$ small

c) Jacobian constant

In the object-oriented model decoupling can be easily achieved by choosing a simpler (decoupled) model for the power flow provided by the physical n-ports:

$$\frac{\partial P_{kn}}{\partial V_i} = 0\,, \quad \frac{\partial Q_{kn}}{\partial \theta_i} = 0$$

In the object-oriented model there is no need for uniformity in these simplifications. This means assumptions such as $\theta_{km}$ and $g_{km}$ small can be made for the physical models where they are appropriate, whereas for other physical objects exact models can be used. In order to make use of the savings obtained by the decoupling of the linearized element model the routine that factorizes a node must check the off-diagonal elements of the 2 by 2 pivot block. If they are zero a simpler procedure for the inversion may be applied.

**Advantages of the Approach**

The major advantages of the object-oriented approach using the generic task 'element load flow' are the following:

- The load flow program is completely independent from the models for the physical elements. Therefore these models can be changed and updated without modifications of the load flow program.
- Arbitrary models for the physical objects can be used in the same load flow functions. For each physical element the suitable model can be chosen.
- The extension to n-ports allows easy implementation of elements such as 3-winding transformers, multiple lines between buses etc.
- The n-port model can be used to model external networks. This allows to include arbitrary external network models, not only the ones based on $\pi$-elements.

**6.2. State Estimation and Bad Data Detection**

For state estimation again the power flow equations of the physical objects are the basic equations to be solved. In order to demonstrate how a state estimator can be built up from the generic tasks element power flow, graph traversal and conceptual object operations, the implementation of the basic weighted least square state estimator (WLS) is discussed.

In this case the estimate $\hat{x}$ is obtained from:

$$\hat{x} = \min_x J(x) = \min_x \left[z - h(x)\right]^T W \left[z - h(x)\right]$$

The optimality condition for this is:

$$\frac{\partial J(x)}{\partial x} = 0\,, \qquad i.e. \;\; H(x)^T W \left[z - h(x)\right] = 0$$

where $H(x)$ is the matrix of the partial derivatives

$$H(x) =. \frac{\partial h}{\partial x}(x)$$

The solution algorithm for the basic WLS is to solve the linearized equation iteratively:

$$G(x^k)\,\Delta x^k = y(x^k)$$

where $G(x)$ is the Gain Matrix defined as

$$G(x) = H^T(x)\ W\ H(x)$$

and

$$y(x) = H^T(x)\ W\ \left[ z - h(x) \right]$$

This is a linear equation involving a sparse matrix $G(x)$ and can be solved with the same methods as the for the load flow. It remains to discuss how the vectors and matrices $z, h, H, W, G$ are represented in the object-oriented model.

**Representation of the Quantities for the State Estimator**

The measurements are represented by introducing a class 'Measurement'. Then the measurement value $z_i$ and the accuracy $w_i$ can be stored directly with the measurement object with two attributes 'w' and 'z'.

The values of $h_i(x)$ and the vector $H_i(x)$ can be associated with the measurements. The values for these can be directly calculated from the element power flow equations provided by the physical objects. For example, for a line flow measurement $h_i(x)$ corresponds directly to the element power flows $P_i(x)$, $Q_i(x)$ respectively. An injection measurement is associated with a node. As in the load flow the appropriate element line flows have to be added. Similar procedures work for the linearization of the measurement equations $H_i(x)$.

The vector $y$ has the same dimension as the state variables. It therefore belongs to the object node. It is calculated by traversing the graph on the node level. The local procedure at each node performs the following step:

$$y_i = \sum_{k \epsilon K} \frac{\partial h_k}{\partial x_i}(x)\ w_k\ \left[ z_k - h_k(x) \right]$$

where the sum has to be taken over K, the set of all measurements that influence the particular node. These measurements are the injection measurements of the node, its adjacent nodes and the line flow measurements of all the branches incident to the node.

The graph associated with the gain matrix has the same nodes as the graph for the load flow but different branches. The diagonal elements of $G$ therefore belong to the object node, and the off-diagonal elements to the objects of class 'Measurement_Branch', in the context of the state estimation referred to as branch. For each line flow measurement a branch has to be created that corresponds to the branch in the load flow network. For each injection measurement two sets of branches are created. The first ones are branches between the node with the measurement and the adjacent nodes. The second ones are the branches between the adjacent nodes.

The calculation of the elements of

$$G(x) = H^T(x)\ W\ H(x)$$

can be done, for example by traversing the graph on the measurement level:

[ FIRST_MEASUREMENT ( G_Update, Next_Measurement ) ]

The local procedure 'G_Update' is different for line flow and injection measurements. For line flow measurements the following steps are performed:

1) Create a branch corresponding to the load flow branch if it does not exist yet. For reference let this branch be between node k and l.

2) Update its $g_{kl}$:

$$g_{kl} \leftarrow g_{kl} - \frac{\partial h_m}{\partial x_k} \frac{\partial h_m}{\partial x_l}\ w_m$$

where $h_m$ corresponds to the appropriate linearized measurement equation provided by the object for measurement m.

For an injection measurement:

1) Update the $g_d$ of the node corresponding to the measurement and the $g_{da}$ of the adjacent nodes as follows:

$$g_d \leftarrow g_d - \left[\frac{\partial h_m}{\partial x}\right]^2 w_m$$

$$g_{da} \leftarrow g_{da} - \left[\frac{\partial h_m}{\partial x}\right]^2 w_m$$

2) Create the branches incident to the node corresponding to the load flow model if they do not exists yet.

3) Update $g_{kl}$ of those branches:

$$g_{kl} \leftarrow g_{kl} - \frac{\partial h_m}{\partial x_k}\frac{\partial h_m}{\partial x_l} w_m$$

4) Create the cross branches between the adjacent nodes and update them analogously.

**Solution of the Nonlinear State Estimator**

The gain matrix G is represented in the same way as the Jacobian for the load flow. Therefore the solution of the linearized system is the same as the one presented for the load flow in the previous section.

Also decoupled versions are easily implemented with the object-oriented model. For the state estimator there are two possibilities for decoupling: algorithm and model decoupling. Algorithm decoupling introduces approximations into the gain matrix. This is obtained by neglecting the elements of $G$ involving $P$ and $V$ or $Q$ and $\theta$. Model decoupling is achieved by defining decoupled functions for the element power flow.

**Sparse Inverse for Bad Data Detection**

Bad data detection is the task of identifying grossly erroneous measurements and eliminate the effects of the bad measurements on the estimate. A reliable approach is to calculate the normalized residuals and compare them with the normal distribution [Mon.2, Gar.1]. If only a single bad data is present, its corresponding normalized residual is the largest. For multiple, interacting bad data the normalized residual approach can be combined with systematic search [Mon.3]. Identified bad data has to be deleted and the state reestimated. In order to avoid refactorization of the gain matrix the bad measurements are replaced by pseudo-measurements [Mon.2].

For all these techniques the diagonal elements of the residual covariance matrix $\tilde{R}$ are needed. $\tilde{R}$ is obtained from the linearized analysis:

$$\tilde{R} = W - H\,[\,H^T\,W^{-1}\,H\,]^{-1}\,H^T = W - H\,G^{-1}\,H^T$$

In order to calculate the diagonal elements of $\tilde{R}$ only those elements of $G^{-1}$ are needed that occupy the same positions as the elements of $G$ itself. Therefore the method for calculating the sparse inverse can be applied.

It is a fact that the elements of the sparse inverse $Z = G^{-1}$ can be computed recursively using only the elements of $LDU$. The matrix G is symmetric. We assume that the triangular factorization of $G$ is given as $LDL^T$ and that the diagonal elements $d$ are properties of the objects of class 'Node' and the of diagonal elements $l$ are properties of the objects of class 'Branch' (fill-ins included). The elements of $G$ are associated with the same objects. These properties have been established by the state estimator.

The sparse inverse can be calculated by traversing the nodes of the network in the opposite order of the factorization and performing the following operations at each node i:

1) Denote the set of all processed nodes that are adjacent to the node to be processed by $A$. Calculate the z-elements ($z_{ij}$) of all the branches incident to node i that lead to adjacent nodes j that have not been processed yet ($j \varepsilon A$) as follows.

$$z_{ij} = - \sum_{k \epsilon A} z_{ik} \; l_{kj}$$

The sum has to be taken again over the set of the not yet processed nodes.

2) Calculate the element $z_{jj}$ of the node as

$$z_{jj} = d_{jj}^{-1} - \sum_{k \epsilon A} z_{jk} \; l_{kj}$$

This concludes the description of the implementation of the analytical functions. It shows how these functions can be broken down into local steps that can be implemented as messages of the objects. Next we discuss an example of a rule based expert system and its relation to the object-oriented data model.

### 6.3. Rule Based Contingency Analysis

#### 6.3.1. Introduction

The task of identifying the critical contingencies is solved either by searching the possible cases and ranking their severity e.g. by the use of a performance index, or in a more heuristic manner by using the operator's judgement given in the form of heuristic contingency selection rules [Schu.1, Sob.1].

The contingency ranking approach is systematic and exhaustive. However, this method is not feasible for the analysis of multiple contingencies, i.e. the outage of two or more (non-)related elements, because it represents a very demanding combinatorical problem.

The heuristic approach relies on the existence of contingency selection knowledge given in the form of rules. In current implementations the rules are often not written down and the operator directly provides a set of contingencies to be examined. This heuristic approach is a typical application for ES: The knowledge is given in the form of rules because the complete problem is too complex to be solved in a systematic way. Hence, contingency selection is chosen as example to show how ES problems are solved using the system proposed above.

Note that the advantage of the systematic contingency ranking approach for single contingencies can be combined with the heuristic solution by simply providing rules which propose the n highest ranked outages evaluated previously by a contingency ranking algorithm.

Hence, the appropriate ES-task can be summarized as follows: Given a power system state and additionally the changes from the the previous to the current system state, apply the given contingency selection rules in order to come up with a set of critical single and multiple contingencies.

A contingency itself is either an element outage (generator, some types of branches) or a bus-bar split. In order to perform the analysis properly, the protection system has to be modeled: The outage of an element may cause the tripping of some switches resulting occasionally in a configuration change which is ultimatively different than the configuration when just cutting the contingency element out of the node-branch model. An illustrative example is given in [Schu.1].

#### 6.3.2. Contingency Selection Rules

As an example of how contingency selection rules can be coded within the object-oriented system we have chosen the rules presented in [Schu.1]. The purpose of this example is not to present an operational contingency selection scheme but to demonstrate how easy the rules and the messages can be combined in the object-oriented system. In the approach presented in [Schu.1] single contingencies are provided by a ranking algorithm or by the user. Multiple contingencies are selected only from network parts where switching occurred recently.

Basically, a rule consist of the left-hand side, a sequence of conditions, and the right-hand side, the action part, a sequence of procedures (messages). A condition involves the properties of an object and/or some related objects (locality). The action part for our application conists of the following functions.

a) propose a single or multiple contingency (Add)

b) evaluate a list of parallel branches to a given object (Parallel_Branches)

c) define the changed topological structure after an object outage (Simulate_Outage)

d) provide a list of connected branch-generator pairs where the branch capacity is lower than the generator capacity (Gen_Bra_Overflow)

e) run contingency ranking for a network group (Con_Rank)

Below the seven rules as listed in [Schu.1] using an object-oriented notation are given. These rules can be associated with the classes of 'Outage_Element' a subclass of the class 'Physical_Object',

Class Outage_Element:

Rule 1.0 IF [ SELF.Con_Flow(Time=Previous) ] > [ SELF.Flow_Limit ]
THEN [ Cont_List.Add(SELF) ]

The contingency caused a flow violation in the previous analysis.

Rule 2.0 IF [ SELF.Must_Run_Operator ]
THEN [ Cont_List.Add(SELF) ]

The contingency was declared "must run" by the system operator.

Rule 3.0 IF [ SELF.Must_Run_Staff ]
THEN [ Cont_List.Add(SELF) ]

The contingency was declared "must run" by the programmer/planning staff.

Rule 4.1 IF NOT [ SELF.Con_Rank ]
THEN [ Network_Group.Con_Rank ]

If no the contingency rank is not determined yet for the current time initiate the contingency ranking algorithm.

Rule 4.2 IF [ SELF.Con_Rank ] <= 10
THEN [ Cont_List.Add(SELF) ]

The contingency is among the 10 worst of the contingency filter process.

Rule 5.2 IF [ SELF.Mul_Con_Reqest ]
THEN [ SELF.Simulate_Outage(Time=[SELF.Outage]) ]

If there is a multiple contingency request for this element, simulate the outage of the element locally. The message "Outage" produces a unique time label over which the values of the simulation can be referenced.

Rule 5.3 IF [ SELF.Gen_Bra_Overflow(Time=[SELF.Outage]) ] <> EMPTY
THEN [ Con_List.Add( [SELF.Gen_Bra_Overflow(Time=[SELF.Outage])] ) ]

Class Node:

Rule 5.1 IF [ SELF.Switch_Ind_Change ]
THEN [ [SELF.Element].Mul_Con_Request ]

If the status of a switch associated with the node changes, request multiple contingency analysis for all outage elements of the node.

Class Generator:

Rule 6.0 IF [SELF.P_Flow] > [SELF.P_Specified] THEN [Con_List.Add(SELF)]

The contingency must be run due to generator output greater than a predetermined level.

Class Branch:

```
Rule 7.0  IF      [SELF.Con_Rank] <= 10
          AND     [SELF.Parallel_Branches] <> EMPTY
          THEN    [ Con_List.Add(SELF, [SELF.Parallel_Branches]) ]
```

If a branch that is a contingency has a parallel branch, then both together form a multiple contingency.

The reasoning for the rules above is relative shallow. Therefore the rules have a high degree of independence and can easily modified and updated. For example, to add a new type of contingency to the list of critical contingency only a appropriate rule has to be added. Also the capability of the system to perform rule based as well as procedural programming been exploited. For example, the simulation of an outage is a procedure, because the sequence of steps is clearly defined.

## 7. Conclusion

In this report we have presented a conceptual design for a data representation and reasoning system for EMSs that makes use of modern programming concepts and rule based reasoning. The most important quality of the system is the integration of conventional algorithms and heuristic reasoning into a single system that takes the particularities of the power system area into account. The system applies the four software concepts: data abstraction, generic tasks, local processing and declarative programming (expert system).

Data abstraction has been achieved by defining an object oriented data model that corresponds directly to the power system. We have described the differencies between physical and conceptual objects and how the relations between them are modeled.

Generic Tasks provide abstraction on the reasoning level. We have defined three generic tasks that are essential to EMS. "Graph Traversal" addresses the task of performing a local operation at each node. It is the work horse of any algorithm since the power system model corresponds to graphs on many different levels. The power system elements form the nodes, and the relations between them the branches of the graph. The generic task "Conceptual Object Manipulations" addresses the problem of how to create, maintain and delete the dynamic (conceptual objects). The task is essential e.g. for topology algorithms and observability analysis. The third generic task is the "Element Power Flow". We have demonstrated how this function makes the power flow used in EMS functions independent from the element model.

We have applied declarative programming by defining a rule based expert system. The advantage of the system is that no interface is required between the expert system and analytical software because both have exactly the same data model. Additionally the rule based system exploits the structure of the object oriented model.

The ideal tool to implement such a system is object-oriented programming. In our presentation we did not concentrate on a particular object-oriented language but we presented the ideas in general terms. Therefore any object-oriented language is suitable for an actual implementation [Schm.1]. The differences within the spectrum of object-oriented languages are considerable, yet they can be divided into two groups:

One group consists of the symbolic languages such as Smalltalk and the object-oriented languages based on Lisp [Lay.1]. The languages in this group are usually interpreted and very user friendly. Therefore they facilitate the development and the debugging of the software. However, the languages in this group perform dynamic binding on all levels which increases the run time overhead considerably. The inefficiences introduced by dynamic binding are not tolerable for large software packages, particularly when they include extensive number crunching such as EMS applications.

The other group are object-oriented extensions of conventional languages such as C and Pascal. These languages have the advantage that the modularity of the object-oriented approach can be combined with relatively low level and therefore efficient implementations. This is the major reason why we have chosen C++ [Str.1] an extension of the language C for the implementation of the system. The disadvantage of this language is that it is very recent and therefore software tool such as debugger and libraries are not yet available. However the object-oriented approach makes it possible to implement libraries that address the specific problems of power system applications. Example for such functions are automatic storage allocation for the static (physical) and dynamic (conceptual) objects, the implementation of the generic tasks and the rule based expert system. Once these tools are specified application programming can be done on conceptual level because all the implementation issues are taken care of.

Currently we are in the process of implementing a prototype written in C++ in order to prove the feasibility and the efficiency of the approach. In a first stage some standard EMS functions such as load flow calculation and state estimation are implemented in an object-oriented environment in order to evaluate their performance. This includes the management of dynamic and static objects and the proposed generic tasks. The final step is the implementation of an object-oriented, rule based expert system.

## 8. References

[Ald.1] R. T. H. Alden and R. Krishnamurlhi, "FORTRAN vs PASCAL for Power Engineering Programming under MS-DOS," *IEEE Trans. Power Systems*, vol. PWRS-1, pp. 179-185, Nov. 1986.

[Alv.1] F. Alvarado, R. Lasseter and Y. Liu, "An Integrated Engineering Simulation Environment," *Proc. PICA Conf.*, Montreal, pp. 213-221, May 1987.

[Alv.2] F. Alvarado and Y. Liu, "General Purpose Symbolic Simulation Tools for Electric Networks," *Proc. PICA Conf.*, Montreal, pp. 222-229, May 1987.

[Bas.1] V. Basili, E. Katz, N. Panlilio-Yap, C. Ramsey and S. Chang, "Characterization of an Ada Software Development," *IEEE Computer*, vol. 17, pp. 53-65, Sept. 1985.

[Bob.1] D. G. Bobrow and M. J. Stefik, "Perspectives on Artificial Intelligence Programming," *Science*, vol. 231, pp. 951-957, Feb. 1986.

[Buz.1] G. D. Buzzard and T. N. Mudge, "Object-Based Computing and the Ada Programming Language," *IEEE Computer*, vol. 18, pp. 11-19, Mar. 1986.

[Byl.1] T. Bylander and S. Mittal, "CRSL: A Language for Classificatory Problem Solving and Uncertainty Handling," *The AI Magazine*, vol. 7, pp. 66-77, Aug. 1986.

[Cha.1] B. Chandrasekaran, "Generic Tasks in Knowledge Based Reasoning: High-Level Building Blocks for Expert System Design," *IEEE Expert*, vol. 1, pp. 23-30, Fall 1986.

[Cle.1] K. A. Clements, G. R. Krumpholz and P. W. Davis, "Power System State Estimation with Measurement Deficiency: An Algorithm that Determines the Maximal Observable Subnetwork," *IEEE PES 1982 Winter Meeting*, paper 82 WM 043-8, Feb. 1982.

[Cox.1] B. J. Cox, *Object-Oriented Programming*. Reading, MA: Addison-Wesley, 1986.

[Chr.1] R. D. Christie and S. N. Talukdar, "Expert Systems for On Line Security Assessment, A preliminary design," *Proc. PICA Conf.*, Montreal, pp. 114-119, 1987.

[Fik.1] R. Fikes and T. Kehler, "The Role of Frame-Based Representation in Reasoning," *Communications of the ACM*, vol. 28, pp. 904-920, Sept. 1985.

[For.1] C. L. Forgy, *OPS5 User's Manual*, Department of Computer Science, Carnegie-Mellon University, 1981.

[Gar.1] A. Garcia, A. Monticelli and P. Abreu, "Fast Decoupled State Estimation and Bad Data Processing," *IEEE Trans. Power App. and Syst.*, vol. PAS-98, pp. 1645-1652, Sept./Oct. 1979.

[Gol.1] Adele Goldberg and David Robson, *Smalltalk-80*, Reading, MA; Addison-Wesley, 1983.

[Hab.1] A. N. Habermann and D. E. Perry, *Ada for Experienced Programmers*. Reading, MA: Addison-Wesley, 1986.

[Har.1] D. Hartzband and F. Maryanski, "Enhancing Knowledge Representation in Engineering Databases," *IEEE Computer*, vol. 17, pp. 39-48, Sept. 1985.

[Hor.1] J. S. Horton, B. Prince, A. M. Sasson, W. T. Wynne, F. Trefny and F. Cleveland, "Advances in Energy Management Systems," *IEEE Trans. Power Systems*, vol. PWRS-1, pp. 226-234, May 1986.

[Imh.1] K. Imhof, "Power System Modeling and Analysis Based on Graph Theory," *Doctoral Dissertation*, Swiss Federal Institute of Technology, Zurich, 1985.

[Jam.1] A. K. Jampala and S. S. Venkata, "Think PASCAL," *IEEE Trans. Power Systems*, vol. PWRS-1, pp. 185-192, Feb. 1986.

[Kae.1] T. Kaehler and D. Patterson, "A Small Taste of Smalltalk," *Byte*, vol. 11, pp. 145-159, Aug. 1986.

[Lay.1] K. Layer, R. Fateman and S. Haflich, *Franz Lisp Reference Manual*, Berkeley: University of California, 1985.

[Liu.1] C. C. Liu and K. Tomsovic, "An Expert System Assisting Decision Making of Reactive Power - Voltage Control," *IEEE Trans. Power Systems* vol. PWRS-1, pp. 195-201, May 1986.

[Liu.2] C. C. Liu, K. Tomsovic and S. Zhang, "Efficiency of Expert Systems as On-Line Operating Aids," *PSCC Conference*, Lisbon, Portugal, 1987.

[Mon.1] A. Monticelli and Felix F. Wu, "Observability Analysis for Orthogonal Transformation Based State Estimation," *IEEE Trans. Power Systems*, vol. PWRS-1, pp. 201-208, Feb. 1986.

[Mon.2] A. Monticelli and A. Garcia, "Reliable Bad Data Processing for Real-Time State Estimation," *IEEE Trans. on Power App. and Syst.*, vol. PAS-102, No. 5, May 1983.

[Mon.3] A. Monticelli, Felix F. Wu and Maosong Yen, "Multiple Bad Data Identification for State Estimation by Combinatorial Estimation," *IEEE Trans. Power Systems*, to appear.

[Nil.1] Nils J. Nilsson, *Principles of Artificial Intelligence*, Palo Alto, Calif: Tioga Pub. Co., 1980.

[Pao.1] Y. Pao, T. DyLiacco and I. Bozma, "Acquiring a qualitative understanding of system behavior through AI inductive inference," *IFAC Symp. on Planning and Operation of Electric Power Systems*, Rio de Janero, July 1985.

[Pas.1] Geoffrey A. Pascoe, "Elements of Object-Oriented Programming," *Byte*, vol. 11, pp. 139-144, Aug. 1986.

[Pis.1] Sergio Pissanetzky, *Sparse Matrix Technology*, London: Academic Press, 1984.

[Ric.1] E. Rich, *Artificial Intelligence* New York: McGraw-Hill Book Company, 1983.

[Sak.1] T. Sakaguchi and K. Matsumoto, "Development of a Knowledge Based System for Power System Restoration," *IEEE Trans. Power App. and Syst.*, vol. PAS-102, pp. 320-329, Feb. 1983.

[Sam.1] J. E. Sammet, "Why ADA is not Just Another Programming Language," *Communications of the ACM*, vol. 29, pp. 722-732, Aug. 1986.

[San.1] S. Mittal, B. Chandrasekaran and J. Sticklen, "Patrec: A Knowledge-Directed Database for a Diagnostic Expert System," *Computer*, vol. 16, pp. 51-58, Sept. 1984.

[Scha.1] G. Schaffer, "User-Oriented Power System Control," *Proc. IFAC Symp. on Power Systems*, Belgium, pp. 152-157, 1986.

[Schm.1] Kurt J. Schmucker, "Object-Oriented Languages for the Macintosh," *Byte*, vol. 11, pp. 177-185, Aug. 1986.

[Schu.1] R. Schulte and Current Operational Problems Working Group, "Survey Report on Current Operational Problems," *IEEE Trans. Power App. and Syst.*, vol. PAS-104, pp. 1315-1320, June 1985.

[Schu.2] R. P. Schulte, G. P. Steble, S. L. Larsen and J. N. Wrubel, "Artificial Intelligence Solutions to Power System Operating Problems," *IEEE PES Summer Meeting*, 1986.

[Sob.1] D. J. Sobajic and Y. Pao, "An Artificial Intelligence System for Power System Contingency Screening," *Proc. PICA Conf.*, Montreal, pp. 107-113, May 1987.

[Str.1] Bjarne Stroustrup, *The C++ Programming Language*. Reading, MA: Addison-Wesley, 1986.

[Tal.1] S. N. Talukdar, E. Cardozo and T. Pery, "The Operator's Assistant - An Intelligent, Expandable Program for Power System Trouble Analysis," *IEEE Trans. Power Systems* vol. PWRS-1, pp. 182-187, May 1986.

[Tes.1] L. Tesler, "Programming Experiences with Object-Oriented Languages," *Byte*, vol. 11, pp. 195-206, Aug. 1986.

[Tin.1] W. Tinney, V. Brandwajin and S. Chen, "Sparse Vector Methods," *IEEE Trans. on Power App. and Systems*, vol. PAS-104, pp. 295-301, Feb. 1985.

[Van.1] H. P. Van Meeteren and J. M. Thorson, Jr., "Network Analysis and Scheduling Functions in State-of-the-Art Energy Management Systems," *Int. Conf. on Large High Voltage Ele. Systems*, Toronto, Canada, paper no. CC 85 07.

[Wil.1] G. L. Wilson and P. Zarakas, "Anatomy of a Blackout," *IEEE Spectrum*, vol. 15, pp. 38-46, Feb. 1978.

[Wol.1] B. F. Wollenberg, "Feasibility Study for an Energy Management System Intelligent Alarm Processor," *Proc. PICA Conf.*, San Francisco, pp. 249-254, 1985.

[Woo.1] William A. Woods, "Important Issues in Knowledge Representation," *Proceedings of the IEEE*, vol. 74, pp 1322-1334, Oct. 1986.

## APPENDIX A: DC-Load Flow in Declarative Form

It is possible to represent most knowledge in declarative or procedural form. As an experiment, a DC load flow has been formulated in a declarative form. It is not expected that this unusual approach is efficient since a declarative representation needs some interpreter which will definitely slow execution down. The goal of the experiment is to exemplify the advantages and disadvantages of such a representation. Furthermore some specific requirements on a declarative representation and inference engine for power system applications can be derived.

As an example the relation between angles and active powers of a decoupled DC load flow are considered. They satisfy the following linear equation.

$$P = B^* \theta$$

B is the real part of the node admittance matrix. B is very sparse. The active powers P are given, the angles $\theta$ have to be calculated. This corresponds to solving a sparse linear equation. Usually triangular factorization is applied:

$$P = L^* U^* \theta$$

| | |
|---|---|
| 1) Factorization | $B = L^* U$ |
| 2) Forward Substitution | $L * z = P$ |
| 3) Backward Substitution | $U^* \theta = z$ |

Commonly the data for this problem is stored in the vectors $P, z, \theta$ and the sparse matrices $L, U$. In a computer implementation the latter are stored in vector form to make use of the sparsity. An index system is used to access the elements of $L$ and $U$.

However, it has been shown that for power systems an object-oriented approach is more useful [Imh.1]. An object-oriented representation does not need a particular data structure for the calculation of the load flow. The diagonal elements of B can be associated with the nodes and the off-diagonal elements with the branches of the power system. Such an approach is necessary to formulate the knowledge for the load flow declaratively.

To find a declarative representation it is necessary to describe the solution to the problem. This leads to the following rule:

Rule 1: The load flow is complete if all the nodes have been factorized and forward and backward substitution are complete.

Next consider the factorization process. The factorization of a particular node affects only the node itself, the branches connected to it, the nodes at the other side of those branches (called adjacent nodes) and finally the branches between adjacent nodes.

Rule 2: A node can be activated for the factorization and the new diagonal element calculated if none of its adjacent nodes is active.

After the initialization of the factorization for a particular node, the process of factorizing that node is captured in the following rules:

Rule 3: The L and U terms of a branch can be recalculated if it is connected to an active node.

Rule 4: A diagonal element of a node can be updated if its adjacent node is active for factorization and the L and U terms of the connecting branches have been recalculated.

Rule 5: A branch between two adjacent nodes can be updated if the branches to the active node have been recalculated.

Rule 6: A new branch has to be introduced between two adjacent nodes and its L and U terms can be calculated if there was no branch and the terms of the branches from the adjacent nodes to the active node have been recalculated (fill in).

Rule 7: A branch should be removed for the factorization process if it is connected to an active node and the rules above cannot be applied any more to this branch.

The rules for the forward substitution can be included in the rules above. Whenever the diagonal element of a node has been recalculated, the elements of the vector z can be updated.

The backward substitution is more complicated to describe declaratively since it has to be performed in a reverse order of the factorization [1]. The rules for forward and backward substitution are not given here since their structure is similar to the ones above.

The description of the rules above is not precise enough for an actual implementation with a rule based language. For such an implementation status slots are needed for the objects (branches, nodes) to keep track which operations have been already executed for that particular object. The DC load flow as described above has been implemented in OPS5 [For.1].

### A.1 Results and Problems of a DC Load Flow in OPS5

Connectivity of the Power System:

The elements of a power system form a network. In the representation of the system knowledge the connection between the elements have to be modeled. In the load flow example in OPS5 each branch has a vector attribute that consists of the name of the two nodes the branch connects. This representation has two disadvantages:

1) It is indirect. This means the access a node connected to a particular branch uses the name of the branch. Such an access involves search.

2) The representation is directed. This means that

   *connect <Bus1> <Bus2>*
   and
   *connect <Bus2> <Bus1>*

   are different. Therefore more than a single rule is needed to perform simple tasks such as finding the adjacent nodes of a particular node. For example, in the OPS5 implementation and Rule 5 had to be represented by four different OPS5 rules.

These two facts make programming tedious and execution inefficient. Since in all power system applications such connections have to be modeled, a specialized AI-system should provide efficient facilities for modeling and access of connectivity.

Sequential and Parallel Execution:

It is noted that the rules for the DC load flow above do not specify a sequence of steps directly. They rather state the conditions for a particular operation. Therefore operations can be executed in an arbitrary sequence or even parallel as long as they do not interact. In a declarative representation parallel execution does not have to be specified explicitly.

It is more complicated to describe a sequence of steps declaratively. In this case each object must have various attributes that keep track of the operations that have been completed on this object. Such an attribute has been used to ensure that the backward substitution is performed in the correct sequence.

Efficiency:

As expected the DC load flow implemented in OPS5 is not efficient. For a 30 bus system more than 500 seconds of CPU time is needed and more than 1000 rules fire. One reason for this inefficiency is that a particular rule about a node may match all the nodes of the system and therefore will be fired many times. The overhead for rule firing and for changing a working memory element is rather expensive [Liu.2].

### A.2 Conclusions

From the results of the implementation of the DC load flow the following conclusions can be drawn:

- Knowledge concerning algorithms and methods where the sequence of steps is indisputable, are more efficiently represented in the form of a procedure.

[1] In fact the order does not have to be exactly reverse depending on the chosen factorization path (see [Tin.1]).

- A declarative representation is not adequate for problems involving numerous low level decisions. The additional overhead from the inference engine makes execution slow.
- A declarative representation is appropriate for problems involving a large pool of heuristic knowledge and when the sequence of steps for the solution cannot be determined. In this case the ease of developing and modifying a knowledge base outweighs the computational inefficiencies.
- An AI system for power engineering applications needs efficient facilities to represent the relations and connections between the power system entities.

# APPENDIX B: Knowledge Representation

Below we give a short overview of the knowledge representation methods and reasoning techniques commonly applied in AI. Additionally, we discuss when the particular methods are appropriate.

## B.1 Introduction

Knowledge representation consists of the following two tasks [Ric.1]:

- Representation of the facts in some chosen formalism (data structure).
- Definition of procedures that allows the 'intelligent' manipulation of the represented knowledge (inference mechanism).

Although knowledge can be represented using virtually any method, the form of knowledge representation is crucial for any AI system. A summary of the commonly known techniques is presented below. To evaluate the most suitable representation techniques the following four aspects are used [Ric.1].

**Representational Adequacy:** The ability to represent all kinds of knowledge that is needed in that domain.

**Acquisitional Efficiency:** The ability to acquire and insert new information most easily.

**Inferential Adequacy:** The ability to manipulate the knowledge in such a way as to derive new structures corresponding to new knowledge inferred from old.

**Inferential Efficiency:** The ability to incorporate knowledge that can be used to focus the attention of the inference.

Usually, the knowledge is very diverse. For different types of knowledge different representation techniques are required. In a first step representation techniques are evaluated with respect to representational adequacy and acquisitional efficiency only. In a second step inference procedures are identified that manipulate different representations and then compared with respect to inferential adequacy and efficiency.

## B.2 Summary of Knowledge Representation Techniques in AI

Below a overview of the various techniques is given. For a more detailed list see [Nil.1, Ric.1]. The methods listed below are realized as underlying knowledge representation techniques in the various AI programming languages.

### B.2.1 Logic

| | |
|---|---|
| Purpose: | representation of simple facts and reasoning with them |
| Types: | propositional logic, predicate calculus, temporal logic |
| Inference: | resolution for propositional logic, first order predicate calculus, none for higher order predicate calculus |
| Characteristics: | uniform, unstructured representation, purely declarative (no possibilities for procedures), resolution is inefficient but complete and exact, information must be consistent, highly modular (knowledge pieces are independent) |
| Realization: | Prolog |

### B.2.2 Procedural Representation

| | |
|---|---|
| Purpose: | representation of executable knowledge |
| Types: | conventional programming languages, attached procedures in frames, procedures in LISP |
| Inference: | compilers and interpreters |
| Characteristics: | not very modular, very situation specific, inflexible, efficient if compiled because only a very low level interpretation is needed |

Realization: FORTRAN, PASCAL, C, ADA

### B.2.3 Semantic Networks

| | |
|---|---|
| Purpose: | representation of highly related facts |
| Inference: | no formal inference engine |
| Characteristics: | structured representation, related to frames, not used for computer implementations because of lacking inference mechanism, no formal semantics. |
| Realization: | no computer implementation, used in psychology and linguistics |

### B.2.4 Production Systems

| | |
|---|---|
| Purpose: | reasoning with knowledge represented in the form of condition-action relations (if-then rules) |
| Types: | production systems using variables, certainity factors, fuzzy logic |
| Inference: | modus ponens used in forward or backward chaining |
| Characteristics: | modular, uniform but unstructured, natural, inefficient, flow of control opaque |
| Realization: | OPS5, OPS83, KEE |

### B.2.5 Frames

| | |
|---|---|
| Purpose: | representation of highly structured knowledge |
| Inference: | no inference engine, however, a frame system provides reasoning services such as procedural attachments, default values and property inheritance. |
| Characteristics: | highly structured form of representation, efficient access of data, low redundance, hierarchical structure |
| Realizations: | part of KEE, KLR |

### B.2.6 Scripts

| | |
|---|---|
| Purpose: | representation of common scenarios and sequences of events |
| Inference: | no formal inference engine |
| Characteristics: | specialization of frames |
| Realization: | no implementation so far |

## B.3 Summary of Reasoning Techniques in AI

Knowledge based reasoning deals with concepts and techniques how the problem solving knowledge can be applied to the data knowledge. in order to produce new data knowledge. There are a wealth of reasoning techniques. Below we provide a summary of reasoning techniques or inference methods. However, we do not take into account that some methods are closely related nor do we try to group the methods. For a more detailed description of methods see [Ric.1].

Resolution:

Resolution operates by taking two clauses each containing the same literal, in positive form in one and in negative form in the other. The resolvent is obtained by combining all of the literals of the two parent clauses, except the ones that cancel. This reasoning technique is used in logic such as propositional logic, temporal logic, first order predicate calculus.

Heuristic Search:

Heuristic search methods builds up a graph where each node proposes a (part) solution in order to search for a goal. Basically, search is the work horse that gets exploited when other techniques

are not available. Variants of these techniques, different ways how the problem graph is explored, are generate-and-test methods, hill climbing, breadth-first search, depth-first search, best-first search, problem reduction approaches, constraint satisfaction, mean-end analysis, minimax search and alpha-beta pruning.

Nonmonotonic Reasoning:
During problem solving it is often the case that an expert cannot proceed further without making assumptions. But later the process might encounter contradictions. This means that some assumptions are wrong. In a next step the wrong assumptions have to be identified and all the deductions based on them must be discarded.

Statistical and Probabilistic Reasoning:
To each rule or fact some probability figures such as measures of (dis)belief, certainity factors are assigned. At every reasoning step appropriate probability changes must be supported.

Production Systems / Modus Ponens:
A production system consists of a set of IF-THEN rules, facts about a specific domain, and a control strategy which says how rules are applied. For example, the OPS-5 programming language is based on the reasoning concept modus ponens. It is interesting to note that production systems can be used to model any computable procedure.

**B.3.1 Some Reasoning Issues**

There is a wide variety in the way how the reasoning is performed. Some styles are exact (resolution) others are inexact (probabilistic reasoning). Often the completeness of a method is sacrificed for the efficiency. For example, resolution is complete, i.e. it is guaranteed that an solution is obtained if one exists, whereas the modus ponens is much more efficient but not complete.

Most of these methods can perform in both directions. Forward changing starts from facts and works towards the usually unknown goal. In the other direction backward chaining tries to confirm a specified hypothesis or tries to establish a given goal.

The reasoning methods are closely related to some representation forms. In fact some knowledge representation forms are designed to serve an intented reasoning technique, e.g. clause representation for the first order predicate calculus based inference engine. Note that an inference engine is just the realization of a reasoning method.

The AI reasoning techniques have in common that the problem solving knowledge is represented in a more abstract, human like way than in conventional languages. The advantage of these methods is that the problem can be formulated in a natural way. Therefore, programming is easier and thus more efficient. The interpretation of the knowledge can be left to the computer. On the other hand we have to pay for this with additional CPU-time used to interpret the declarative knowledge.