A DOMAIN ARCHITECTURE FOR SOLVING SIMULTANEOUS NONLINEAR NETWORK EQUATIONS

Peter Bosshart
Peter.Bosshart@eeh.ee.ethz.ch
Swiss Federal Institute of Technology (ETH), CH 8092 Zürich, Switzerland

Rainer Bacher
Rainer.Bacher@eeh.ee.ethz.ch

Abstract This paper describes an approach of systematic software reuse by defining a domain architecture. An appropriate domain model formulates network and connected entities with behavior leading to systems of simultaneous nonlinear equations. An object-oriented framework is derived from the domain model and represents a core reusable asset. The domain architecture provides an environment for an adaptive usage of this framework. The person who reuses the domain architecture specifies an application by sets of symbolical expressions assigned to the fundamental nport types and functions representing sums of any variable type. Further the reuser selects or provides a code generating function to adapt the framework. A power flow example with a power interchange area is presented.

Keywords - Software Engineering, Software Reuse, Object orientation, Framework, Code Generation, Newton-Raphson Solution, Network Simulation Environment

1 Introduction

There is a permanent demand for new application and simulation software required for different purposes such as research, planning and operation. These systems become larger and increasingly complex. As a consequence creating system software is more difficult to complete on time and within the constraints of budget. It has become very difficult to create these new applications with traditional software development technology. When finished, they are difficult to understand. Thus they are hard to maintain and to modify for new requirements.

Studies in computer science have shown that reusability can improve software development productivity and quality. Productivity increases as previously developed assets can be used in current applications, which saves new development time. Quality can be increased as frequently reused assets have been tested and corrected in different use cases.

Despite the benefits of reuse there are several reasons why software is not reused. They range from lack of training to create reusable assets to lack of tools to support reuse. A main obstacle for software reusability is the learning curve of a reusable asset.

Successful reuse has been achieved within mature and stable domains. A domain means a distinct functional area that can be supported by a class of software systems with similar requirements and capabilities [1].

In this project we focus on the domain of network and connected entities with behavior leading to systems of simultaneous nonlinear equations.

Domain analysis and modeling are key activities to create reusable assets. Based on modeling in the same domain boundary several reusable concepts were presented. In [2, 3] node-edge oriented modeling approaches are proposed. They are suited in many typical power flow cases but they are not general enough to cover the wide area of real-world models which go beyond node and edge based models. A generic power network container has been shown in [4]. Based on the same modeling approach as in [2] it uses object-oriented concepts. Reuse is done by specializing derived classes and by using templated classes.

The error prone coding of Jacobian and mismatch terms has to be done by the software reuser. A general purpose symbolic simulation tool is presented in [5]. There the symbolic processing of user specification allows a very high degree of coding flexibility, however, needs much execution time and can lead to performance problems required for on-line software.

This paper presents a reusable asset which is called a framework supporting the domain. The final product of the framework is C++ source code. However, its behavior cannot be completely adapted within the framework. Additional elements such as Jacobian and mismatch term generation are necessary which are organized in a matching domain architecture. Due to the object-oriented framework together with the proposed modeling approach we strongly believe that one advantage of the proposed domain-architecture is that the specification and adaption part have become smaller and thus easier to use and to change by a reuser.

The framework is far more than a library. It contains control flow and protocols, allowing to reuse the design of the solution for a class of problems. The framework itself contains much genericity, which has no longer to be provided by the adapting part. The proposed modeling approach is better suited for the object-oriented framework than the approach based on buses and branches. Due to decoupling the information of topology and behavior of entities the setup of Jacobian terms is much easier, simplifying the adaption part further. A second advantage is the enhanced modeling flexibility of the proposed approach by nports and sumsetters. The main disadvantage
2 Applied Software Concepts

The reuse of existing software assets seems to be an attractive way to reduce cost and time effort of software development. Reuse software systems inherently use a software engineering process that is specifically structured for reuse. Systematic reuse must be organized [6, 7] and means that software development is guided by an organized use of domain engineering products (including a domain model, domain architecture and other assets) during successive stages of a software engineering process [1]. An architecture means an organizational structure of a system or asset [1]. In [8] an architecture for distribution control systems is presented.

Today diverse approaches of reuse exist: Class-Library, code generator, code-skeleton and framework. A framework is a collection of cooperating classes that make up a reusable design solution for a given domain [9]. The framework is the reusable asset within a domain architecture. Frameworks allow programmers to take advantage of preassembled complex relationships among objects. There is no need for a reuser to understand all details of those relationships. [10] describes the use of an object-oriented framework for the development of EMS (Energy management systems) software. Frameworks rely heavily on object-oriented concepts. Data abstraction and encapsulation by a class are the most important contributions to allow easier modeling. Object-oriented modeling is advantageous because there is little gap between domain modeling and implementation. Subclassing, inheritance and dynamic binding are further enabling key features for building reusable components. Polymorphism is an important ability to hide different implementations behind the same name. This greatly simplifies communication among objects. In the concept presented in this paper the language C++ [11] was used as the object oriented implementation language.

3 Domain Model

This paragraph presents the result of domain analysis. This was done by analyzing the power flow problem which is a typical instance of the domain. A common data format definition of a solved power flow was used [12] to figure out requirements. The structure of data and flow of information within the domain are identified and described. It is shown how the different entities contribute to the overall nonlinear equation system \( g(\vec{x}, \vec{p}) = 0 \) where \( \text{dim } \vec{x} = \text{dim } \vec{g} \), \( \vec{x} \) is the vector of unknown variables, \( \vec{p} \) is the vector of known parameters.

3.1 Network and Entities

A network is a set of connected entities called “nports” and “sumsetters”. These entities are always connected to each other by ports. An “nport” entity has exactly \( n \) ports where \( n \) is an integer number greater than zero. Note, however, that a physical device which has 3 physical connections is not necessarily a 3port device (\( n=3 \)). Its number “\( n \)” is identical with the number of real variables of the modeling approach is the higher dimension of the linearized equation system.

![Fig. 1: Network composed of nports and sumsetters entities](image)

which are accessed by “sumsetters”.

![Fig. 2: Connections as routing path to variables](image)

3.1.1 Port

Sumsetters access variables located in nports. The access path to variables is called a connection. Each connection is established by adding a port to a sumsetter. Each port holds two data items: the appropriate nport identifier \( k \) and an index \( j \), which is part of a key to a variable located in an nport \( k \).

3.1.2 Variable

A variable is an entity related to a component \( x_m \) of unknowns \( \vec{x} \) within the system \( g(\vec{x}, \vec{p}) = 0 \). It is represented on the computer as a real type value. One key idea of this model is, that the numerical value \( x_m \) is accessible from two different view points. The first view point is associated with the order of variables used in the linear system solver of the Newton-Raphson solution. Access to the variable value is made by the index \( m \) which is stored on variables. The second view point is related to equations and variables local to nports and sumsetters. The function \( V_k(\text{basename}, \text{index}) \) returns the numerical value of the variable identified by a string \( \text{basename} \) and an optional integer value \( \text{index} \). Both arguments build the key to a variable in the naming space of the nport \( k \).

3.1.3 Parameter

A parameter is an entity related to a known value stored in nports and sumsetters. It is a component of known values \( \vec{p} \) within the system \( g(\vec{x}, \vec{p}) \). It is represented on the computer as a real type value.
3.1.4 Nport

Nport is the entity typically associated with an electrical element such as generator, load, line etc. It can be seen as a finite state machine with a behavior \( B N_\sigma(\mathcal{V}_\sigma, \mathcal{P}_\sigma, \mathcal{E}_\sigma, \mathcal{T}_\sigma) \) for each state \( \sigma \). Opposite to sumsetter the behavior of the nport is known at specification time. The \( n_\sigma \) equations ranging from \( q \) to \( q+n_\sigma-1 \) are specified independently of the embedding network. They are formulated with a known set \( \mathcal{V} \) of \( n_\sigma \) variables ranging from \( s \) to \( s+n_\sigma-1 \) and parameters. The behavior \( B N_\sigma \) in a state \( \sigma \) contains the following sets:

\[
\begin{align*}
\mathcal{V}_\sigma &= \{ x_s, x_{s+1}, \ldots, x_{s+n_\sigma-1} \} \\
\mathcal{P}_\sigma &= \{ p_1, p_2, \ldots, p_{n_\sigma} \} \\
\mathcal{E}_\sigma &= \{ g_q(\mathcal{V}_\sigma, \mathcal{P}_\sigma) = 0, \quad g_{q+1}(\mathcal{V}_\sigma^{q+1}, \mathcal{P}_\sigma) = 0, \ldots, \quad g_{q+n_\sigma-1}(\mathcal{V}_\sigma^{q+n_\sigma-1}, \mathcal{P}_\sigma) = 0 \} \\
\mathcal{T}_\sigma &= \{ \quad v - p_u \geq 0 \Rightarrow \text{next state} = \sigma_1, \quad p_u - v \geq 0 \Rightarrow \text{next state} = \sigma_2 \}.
\end{align*}
\]

Parameter \( p_u, p_l \in \mathcal{P}_\sigma \), where \( p_u > p_l \) and variable \( v \in \mathcal{V}_\sigma \). For a state \( \sigma \) there are a maximum of two next states \( \sigma_1 \) and \( \sigma_2 \). If no condition for a transition is satisfied it is assumed that the state does not change. If a condition for a transition is satisfied, the nport propagates a change request to the owner object (the network). This object in turn enables the nport to go to the specified next state \( \sigma_1 \). Then, the new behavior will be \( B N_{\sigma_1}(\mathcal{V}_{\sigma_1}, \mathcal{P}_{\sigma_1}, \mathcal{E}_{\sigma_1}, \mathcal{T}_{\sigma_1}) \).

These state descriptions allow the formulation of changes of sets of equations for certain nport conditions. This is especially important in a practical power flow formulation where for example generators can change from PV- to PQ-type due to limited Q or where remotely controlled branch quantities such as the active power flow can become active, etc.

3.1.5 Sumsetter

A sumsetter \( s \) accesses variables located in nports to create further subsets of equations \( \hat{g}^{(s)}(\hat{x}^{(s)}, \hat{p}^{(s)}) = 0 \). These equations typically express topological information such as incidence relations within networks. A sumsetter has \( n_\sigma \) instances of sum types and a set \( \hat{\mathcal{A}} \) of \( n \) connected ports. All sum types generate equation(s) containing \( n \) variables with a common basename. Optionally a single variable \( l_v \) and a parameter \( \text{par} \) may be added. There are two sum types:

1. **OneRowSum** (basename, \([l_v, [\text{par}]\]), \( \hat{\mathcal{A}} \)):
   \[
   \sum_{i=1}^{n} \mathcal{V}_k(\text{basename}, j_i) + l_v + \text{par} = 0
   \]

2. **MultiRowSum** (basename, \( \hat{\mathcal{A}} \)):
   \[
   \begin{align*}
   \mathcal{V}_k(\text{basename}, j_1) - \mathcal{V}_k(\text{basename}, j_2) &= 0 \\
   \mathcal{V}_k(\text{basename}, j_2) - \mathcal{V}_k(\text{basename}, j_3) &= 0 \\
   & \ldots = 0 \\
   \mathcal{V}_k(\text{basename}, j_{n-1}) - \mathcal{V}_k(\text{basename}, j_n) &= 0
   \end{align*}
   \]

The behavior \( B S(\hat{\mathcal{A}}, \mathcal{E}) \) of a sumsetter has the following sets, whereby Sum stands for both types of sums:

\[
\hat{\mathcal{A}} = \{ \text{port}_1, \text{port}_2, \ldots, \text{port}_n \}
\]

\[
\mathcal{E} = \{ 
\begin{align*}
g_q &= \text{Sum}(\text{basename}_q, [l_v], [\text{par}], \hat{\mathcal{A}}), \\
g_{q+1} &= \text{Sum}(\text{basename}_{q+1}, [l_{v+1}], \\
&\quad \quad \quad [\text{par}_{q+1}], \hat{\mathcal{A}}), \\
&\quad \quad \quad \ldots, \\
&\quad \quad \quad g_{q+n_\sigma-1} = \text{Sum}(\text{basename}_{q+n_\sigma-1}, [l_{v+n_\sigma-1}], \\
&\quad \quad \quad [\text{par}_{q+n_\sigma-1}], \hat{\mathcal{A}}) \}
\end{align*}
\]

In contrast to nport the behavior \( BS \) is not completely known at specification time. The actual set of ports \( \hat{\mathcal{A}} \) is known only after the network data has been read.

3.2 Entities and system of equations

The overall system \( g(\hat{x}, \hat{p}) = 0 \) contains \( n_g \) simultaneous nonlinear functions, depending on unknowns \( \hat{x} = \{x_1, x_2, \ldots, x_{nv}\} \) and knowns \( \hat{p} \). Subsets of \( g, \hat{x} \) and \( \hat{p} \) are stored and managed only in nports and sumsetters. For the system the following assumptions are made:

1. All components of \( \hat{x} \) are continuous variables
2. The total number of variables \( nv \) is equal to the total number of functions \( n_g \). This is a necessary but not a sufficient condition for the solution of a non-linear equation set.
3. The linearized components of \( g \) are linearly independent functions.
4. \( \exists \hat{g}(\hat{x}, \hat{p}) \quad \forall \{i, j\} \): All first partial derivatives of all equations with respect to all unknown variables exist.

4 Domain architecture

This paragraph presents a generic, organizational structure for applications in the domain. It contains a design that satisfies the requirements specified in the domain model. Thus the domain model has impact on the elements of the architecture. The architecture can be adapted to create different applications within the domain. It defines a structure and a development process for configuring the object-oriented framework. An application means a set of specified behavior for each used type of nport and sumsetter. Further specifications define the correspondence to a selected network data interchange format [12].

![Fig. 3: Architecture](image-url)
The proposed architecture has three layers, see Fig. 3. In the first layer, the variability of applications are captured in specification data. This data controls the adaptation of generic parts of the framework. The second layer contains the commonality of applications in the domain. They are formulated by hard-coded classes within the object-oriented framework. The framework is C++ source code containing hard-coded and generated classes. Within the third layer, a runtime system is built. An instance of the framework is created which reads user defined network data and outputs the result of one simulation step.

4.1 Specification

The behavior $BN$ of each nport and $BS$ of each sumsetter type is specified in a symbolical way by an appropriate Maple V [13] procedure. The procedure has the same name as the nport or sumsetter type plus the name part “Def”. This naming convention is used for later processing. The procedure returns the behavior in two unified Maple V record formats: one for the $BN$ and one for the $BS$. These records define an intermediate format to the adaptation part of the architecture. Further parts of the specification are the set of all required types (sumsetters and nports) and information of how to read network data from a selected network data interchange format (for example the one of [12]).

4.2 Adaptation

A code-generating part customizes the generic parts of the framework. It is done by creating classes for each required sumsetter and nport type. Again Maple V is used to do this job and to generate Jacobian and function mismatch calculating C-source code. In addition, the constructor of the network class is built, depending on the selected interchange format. It reads network data from this interchange format and instantiates objects.

4.3 Framework

By applying object modeling techniques [14] the domain model was translated into ten hard-coded and two generic classes, anynport and anysumsetter. Translation is complicated by two constraints:

- execution runtime efficiency
- ease of adaptation of generic parts: anynport, anysumsetter and constructor of the network class

The strongly coupled classes demonstrated in Figs. 4 and 5 together build the key elements of the domain architecture.

On the top of Fig. 4 the most general classes are shown.

Lower positioned classes inherit data and behavior (methods) from the upper classes. Multiple inheritance was used to accumulate characteristics of parent classes. Description of classes not mentioned in the model:

- **NetworkObj**: Provides a name for objects.
- **NewtonRaphson**: Virtual base class to declare the behavior of objects involved in the domain solution

Fig. 4: Class hierarchy

Fig. 5: Has-Relations between classes

process. They all act on a solver object which encapsulates a linear sparse solver. The appropriate methods are:

```
virtual void EvalLinSysSize(int& size){};
virtual void CountNonZeroes(int& nZ){};
virtual void PutSolver(LinSolver& solver){};
virtual void GetFromSolver(LinSolver& solver, double& maxMism){};
virtual void SetFunStartLfNr(int& fnr){};
virtual void PrintVarsAndPars(){};
RVar* GetVar(char* name){...};
```

- **Network**: is a container that stores objects derived from nport and sumsetter. There are methods to create and insert these instances and connections (ports) between them. Further there is a method to start the calculation with a given mismatch tolerance and a maximum number of iterations. The constructor has to be customized to interpret a chosen interchange format of network data.

4.3.1 Mapping of locally defined behavior to Newton-Raphson process

The nport and sumsetter assemble the overall system of simultaneous nonlinear equations $\mathbf{f}(\mathbf{x}, \mathbf{p}) = 0$. The solution of this system is obtained by the well known Newton-
Raphson process. The symbolic expressions of the linearized equation terms $A_{ij}$ and $b_i$ of a Newton-Raphson iteration are discussed in the following. Fig. 6 demonstrates the following assignments:

\[ A_{ij} = \begin{cases} \sum_{k} g_k(v_k,p_k) : q_k ≤ i < q_k + ng_k \\ \frac{\partial g_k}{\partial x_j} : s_k ≤ j < s_k + nv_k \\ 0 \end{cases}, \quad b_i = \begin{cases} g_i(v_k,p_k) : q_k ≤ i < q_k + ng_k \\ 0 \end{cases} \]

\[ \begin{cases} 1 : q_k ≤ i < q_k + ng_k \\ x_j \text{ part of OneRowSum}_{i}(\ldots) \end{cases}, \quad b_i = \begin{cases} OneRowSum_{i}(\ldots) \text{ or MultiRowSum}_{i}(\ldots) \\ 0 : \text{else} \end{cases} \]

$u_i$ is the negative correction value of the associated variable $x_i$. Nport and sumsetter update their maximum mismatch and variables.

C++ was chosen as main programming language. The sparse linear solver is taken from the FORTRAN “umfpack” [15]. Templates from the C++-LEDA library [16] are used for linked lists.

### 4.4 Environment

Within this environment (using UNIX as operating system) an executable file is generated. There is a makefile for the whole application generating process. After creation of a library containing all Maple V procedures needed for specification and code generation, all parts of the framework are customized accordingly. Together with the hard-coded classes of the framework all parts are included in a default main program. The default main program simply creates an instance of a network and starts the methods Calculate and PrintVariables. Depending on the arguments the main program reads in the network data file of the selected network data interchange format.

### 5 Application example

An application example is given in Fig. 7. It is an AC power flow problem for steady state conditions. Thus the unknowns are formulated as complex numbers. The complex cartesian coordinate variables $e + j \cdot f$ (voltage), $ie + j \cdot if$ (current) and $P + j \cdot Q$ (power) are related to each electrical terminal of the electrical elements SlackGen, SlackArea, PVGen, PLoad, ImpLoad and Z. These elements are mapped to nport-types. PVGen can change from PV- to PQ-type due to limited $Q$. Since $Q_{min} ≤ Q ≤ Q_{max}$ three states are necessary to cope with the different behavior.

Bus1 ... Bus5 are the interconnecting points of the network. The interchange area defines a region for which the sum of exported real power is a given value $P_{schedule}$. All entities called bus and power interchange area are modeled as types of nsetters.

---

**Fig. 6: Structure of sparse linear equation system**

Nport\_k related equation terms:

\[ A_{ij} = \begin{cases} \frac{\partial g_k(v_k,p_k)}{\partial x_j} : q_k ≤ i < q_k + ng_k \\ \frac{\partial g_k}{\partial x_j} : s_k ≤ j < s_k + nv_k \\ 0 \end{cases} \]

$B_i = \begin{cases} g_i(v_k,p_k) : q_k ≤ i < q_k + ng_k \\ 0 \end{cases}$

Sumsetter\_k related equation terms:

\[ A_{ij} = \begin{cases} 1 : q_k ≤ i < q_k + ng_k \\ x_j \text{ part of OneRowSum}_{i}(\ldots) \end{cases} \]

\[ B_i = \begin{cases} OneRowSum_{i}(\ldots) \text{ or MultiRowSum}_{i}(\ldots) \\ 0 : \text{else} \end{cases} \]

**Fig. 7: Network example**

**Specification of nport types:**

**SlackGen (Network slack generator):**

\[ \forall = \{ e_1, f_1, i_{e1}, i_{f1}, P, Q \} , T = \{ \} \]

\[ P = \{ esl, fsl \} \]

\[ E = \{ e_1 - esl = 0 , f_1 - fsl = 0 , P - esl i_{e1} - fsl i_{f1} = 0 \} \]

**SlackArea (Area slack bus):**

\[ \forall = \{ e_1, f_1, i_{e1}, i_{f1}, P, Q \} , T = \{ \} \]

\[ P = \{ U \} \]

\[ E = \{ i_{e1} - e_1 P + (i_{f1})^2 = 0 , i_{f1} - \frac{e_1 P + i_{f1}}{e_1^2 + (i_{f1})^2} = 0 \} \]

**PLoad (PV-load):**

\[ \forall = \{ e_1, f_1, i_{e1}, i_{f1} \} , T = \{ \} \]

\[ P = \{ P, Q \} \]

\[ E = \{ i_{e1} - e_1 P + i_{f1} = 0 , i_{f1} - \frac{e_1 P + i_{f1}}{e_1^2 + (i_{f1})^2} = 0 \} \]

**PVGen (PV-generator):**

\[ \forall = \{ e_1, f_1, i_{e1}, i_{f1}, P, Q \} \]

\[ P = \{ P, U, Q_{min}, Q_{max} \} \]

\[ E = \{ i_{e1} - e_1 P + i_{f1} = 0 , i_{f1} - \frac{e_1 P + i_{f1}}{e_1^2 + (i_{f1})^2} = 0 \} \]

\[ T = \{ Q ≥ Q_{max} \Rightarrow 2 \}

\[ Q_{min} - Q ≥ 0 \Rightarrow 3 \} \]
\( V_2 = \{ e_1, f_1, i_e, i_f, 1 \}, T_2 = \{ \} \)

\( P_2 = \{ P, Q_{\text{max}} \} \)

\( E_2 = \{ i_e - \frac{P_i}{e_i + j f_i} Q_{\text{max}} = 0, i_f - \frac{-Q_i}{e_i + j f_i} P_{\text{max}} = 0 \} \)

\( V_3 = \{ e_1, f_1, i_e, i_f, 1 \}, T_3 = \{ \} \)

\( P_3 = \{ P, Q_{\text{min}} \} \)

\( E_3 = \{ i_e - \frac{P_i}{e_i + j f_i} Q_{\text{min}} = 0, i_f - \frac{-Q_i}{e_i + j f_i} P_{\text{min}} = 0 \} \)

**Z (Impedance branch):**

\( V = \{ e_1, f_1, i_e, i_f, 1, e_2, f_2, i_e, i_f, 2, P_1, P_2 \} \)

\( P = \{ R, X \}, T = \{ \} \)

\( E = \{ i_e - R \left( \frac{e_1 + j f_1}{R + j X} - \frac{e_2 + j f_2}{R + j X} \right) = 0, i_e - 3 \left( \frac{e_1 + j f_1}{R + j X} - \frac{e_2 + j f_2}{R + j X} \right) = 0, i_e + i_f = 0, i_f + i_f = 0, P_1 - R ((e_1 + j f_1) (i_e + j i_f))^* = 0, P_2 - R ((e_2 + j f_2) (i_e + j i_f))^* = 0 \} \)

**Specification of sumsetter types**

**Bus:**

\( S = \{ \text{OneRowSum}(\text{basename} = i_e), \text{OneRowSum}(\text{basename} = i_f), \text{MultiRowSum}(\text{basename} = e), \text{MultiRowSum}(\text{basename} = f) \} \)

**PowerInterchangeArea:**

\( S = \{ \text{OneRowSum}(\text{basename} = P, \text{par} = P_{\text{schedule}}) \} \)

Specification of network instantiation, parametrization and results are omitted for space reason. The specification uses 160 lines of Maple V code, the generated C++ code has a total length of about 1400 lines of which 570 lines are generated in the adaptive part of the Class “Network” (see Fig. 4). 250 lines of code in all classes of “Anysumsetter” and 580 lines of C++ code in all classes of “Anynport”. In contrast to these 1400 lines of adaptive C++ code, the total C++ code which includes also the hard-coded parts has 1900 lines. This C++ code does, however, not include the source code for the solution of the sparse linear system of equations and general purpose classes such as lists. The Maple V code to generate the 1400 lines of adaptive C++ code has a total length of 700 lines including the 160 lines of specification. After compilation, the application creates 69 variables and functions. It converges after 4 Newton-Raphson iterations.

Note that Maple V allows the use of \( \Re \) (Real part) or \( \Im \) (Imaginary part) within the specification part.

Specifications of nport- and sumsetter types have been implemented for other power flow problems such as a polar coordinate power flow formulation.

The proposed domain architecture was tested in a power flow application by applying a 2500-bus network. Thereby the system has 5600 nports and creates about 45000 variables. To start applying the proposed techniques the following tools and libraries are needed: An operating system supporting the make-tool, GNU g++-compiler/linker, a FORTRAN 90-compiler, a FORTRAN to C converter (F2C) and Maple V. Further the publicly available libraries LEDA, UMFPACK, HARWELL and BLAS are needed. Both source-code of the framework, the elements of the architecture and the specifications of nport/sumsetter suited for a loadflow application are ASCII-files.

## 6 Conclusions

This paper proposes an object-oriented framework as a key component of a reuse-oriented architecture for the domain. Both the framework and the architecture have evolved in an iterative development process including steps of analysis, abstraction, model enhancement and update. It turned out that a network can be modeled by instances of the two generic types nport and sumsetter. The adaptation of generic parts has been done in two steps. In a first step the symbolical behavior specification of nport and sumsetter types is captured using Maple V as environment and language. Then code-generating functions customize the generic parts of the framework. This is done by creating classes for each required sumsetter and nport type.

The second step adds the constant parts of the domain. This allocation of responsibilities simplifies the development of the code-generating part which is a task complicated by manifold aspects. The payoff of this approach is a high reduction in the effort to develop software within this complex domain.

Thereby a first advantage is the enhanced modeling flexibility of the proposed approach by nports and sumsetters:

- ease of modeling of entities with more than two terminals, such as three phase networks, sub-stations, mutually coupled lines etc.

- ease of real zero impedance modeling of closed switch

The main disadvantage of the modeling approach is the higher dimension of the linearized equation system.

- more equations and variables

- increased number of instantiated objects

Due to the object-oriented framework together with the proposed modeling approach we strongly believe that a second advantage of the proposed domain-architecture is that the specification and adaption part have become smaller and thus easier to use and to change by a reuser.

The framework is more than a class-library. It contains control flow and protocols, allowing to reuse the design of the solution for a class of problems. The framework itself contains much genericity, which has no longer to be provided by the adapting part. The proposed modeling approach is better suited for the object-oriented framework than the approach based on buses and branches. Due to decoupling the information of topology and behavior of entities the setup of Jacobian terms is much easier, simplifying the adaption part further.
REFERENCES


Peter Bosshart received the Dipl.El.-Ing. degree in electrical engineering in 1983 from the Swiss Federal Institute of Technology (ETH) in Zürich, Switzerland. After ten years industrial experience at ABB Process Automation Corporation, Baden, (Switzerland) he is currently finishing his PhD in the Energy Management Systems group at the department of electrical engineering at the ETH Zürich. He can be reached at Internet e-mail: Peter.Bosshart@eeh.ee.ethz.ch.

Rainer Bacher received the Dipl.El.-Ing. degree in electrical engineering in 1982 and the Dr.sc.techn. degree in 1986, both from the Swiss Federal Institute of Technology (ETH) in Zürich, Switzerland. After his doctorate he joined the Energy Management Systems Division of Control Data Corporation in Minneapolis, U.S.A. In 1989 he joined Colenco Ltd., a Swiss consulting company. Also in 1989 he started as a lecturer and part-time senior researcher at the power system group at the ETH. In 1992 he became a full-time senior assistant and lecturer at the ETH. In 1993 he was appointed assistant professor of Energy Management Systems at the department of electrical engineering at the ETH Zürich. He is a member of technical committees for PSCC 93, 96 and PICA 93, 95, 97. He can be reached at Internet e-mail: Rainer.Bacher@eeh.ee.ethz.ch. His WWW home page is: http://www.eus.ee.ethz.ch/~bacher.