Automatic generation of optimization code based on symbolic non-linear domain formulation

Rainer Bacher
Swiss Federal Institute of Technology (ETH), CH 8092 Zürich, Switzerland
E-mail: bacher@tech.ee.ethz.ch; FAX: +41 1 632 12 52; Tel.: +41 1 632 41 94

Abstract: A concept for the automatic generation of optimization code for a class of non-linear optimization problems is described and realized at the example of an electric power system optimal power flow problem. The equations are structured based on a node and edge structure given from a network. The goal of this domain engineering approach is the high-level symbolic formulation of this structured optimization problem and the subsequent complete automatic code generation of the solution algorithm in Matlab. The main algorithmic step is the iterative solution of a sparse linear system of equations applied to the Karush-Kuhn-Tucker optimality conditions of the optimization problem. The matrix elements of this linear system to be solved during the solution process consist of sums of first and second order optimality conditions of the optimization problem. The equations are assumed to be twice differentiable and the variables are assumed to be continuous variables. The number of real variables and equations is far more than thousand.

In this paper software and algorithmic engineering for the following smooth non-linear optimization problem is discussed:

\[ \begin{align*}
\text{Minimize} & \quad F(x) \\
\text{subject to} & \quad g(x) = 0 \\
& \quad h(x) \leq 0
\end{align*} \]  \quad (1)

All bold variables indicate vectors or matrices. All equations and objective function terms in this optimization problem are assumed to be twice differentiable and the variables are assumed to be continuous variables. The number of real variables and equations is far more than thousand.

This problem formulation (1) is a very general formulation. A compact high level definition from which a general, robust and fast solution code can be automatically derived is not possible since no characteristics of the equation and objective function parts can be seen from the given problem definition. Also, because high non-linearity is assumed as the underlying characteristics of (1) only a more rigorous problem definition helps to specify solution algorithms which satisfy certain robustness, speed, code maintenance and code extensibility requirements. Optimization problems coming from a practical real-world environment always follow some structured form which can be applied to more than one problem instance.

This paper does not intend to provide a solution algorithm and problem entry architecture which solves "all" partially separable optimization problems. The problem is more special (or less generic) and represents a subset of the definition in [8]. However, from a domain expert point of view (in this case a power system engineer) it represents a domain whose efficient formulation and algorithmic solution has merits in itself.

In this paper we do not intend to provide a solution algorithm and problem entry architecture which solves "all" partially separable optimization problems. The problem is more special (or less generic) and represents a subset of the definition in [8]. However, from a domain expert point of view (in this case a power system engineer) it represents a domain whose efficient formulation and algorithmic solution has merits in itself.

To show the main characteristics of the electric power system optimal power flow domain to domain outsiders, this paper has the following structure:

Section 2 describes the domain framework including the mathematical solution approach, the hard-coded, non-user entenable domain characteristics and the design of the formal entry of instances of this structured formulation with Maple V [7]. Section 3 explains with enough generality one instance of the domain at an example of a simple five node electrical power system network. I.e. the optimal power flow domain definition is explained at a simplified problem instance. This should allow to conclude on the generalized instances of the domain in areas other than the electrical power system. In section 4 the most interesting aspects of the symbolic program are discussed. This program reads the

SIF: Standard Input Format; a format for partially separable optimization problems
domain definition and generates fast problem solver code. The paper ends with conclusion in section 5 and the references.

When comparing to similar approaches [10] must be mentioned. There a knowledge based approach is described which allows to specify scientific computing problems. This specification results in a high-level language description that is transformed by domain-independent rules. Although being conceptually similar the present paper concentrates more on the formal entry of a domain and the associated solution of an optimization problem which fits to the domain characteristics.

[9] summarizes the strengths of problem solving environments (PSE). In a more general sense the approach of the present paper is one representative of a PSE.

Other approaches which have some links to the domain engineering phases presented in this paper are described in [5] and [11]. There, the effort is concentrated on automatic code differentiation and automatic code generation of first order partial derivatives of any FORTRAN 77 (in [5]) and first and second order partial derivatives of Maple V (in [11]) code representing functions. The parts which specify the code functions for which first and second order partial derivatives are needed, are more formalized in this paper. The formalization is, however, enough general to cover many cases of interest within the chosen domain boundaries.

Also, a domain engineering has been reported in [3] where a well-determined system of structured equation types is formally entered. The solution of these non-linear equations is based on the automatic generation of FORTRAN 77 code using automatic symbolic computation of first order derivative terms. This structured approach has been extended to the solution of a structured equality constrained optimization problem in [2]. These papers, however, are written for domain experts and assume a deep knowledge of the domain by the reader.

Going back even further in the area of domain engineering papers related to power systems, [1] and [6] are papers pioneering the field of code generation.

2 DOMAIN FRAMEWORK

2.1 Mathematical solution framework

The optimization as formulated in very general form in (1) is solved with a Newton algorithm based on an equality constrained augmented objective function approach. The intent of this paper is not so much a discussion of the solution approach of the optimization problem, but rather the discussion about the formal derivation of the “exact” matrix and “exact” right-hand side vector of the Newton-solution approach within the domain defined later in this paper.

The code which has been developed solves a slightly modified problem formulation as compared to (1):

\[
\begin{align*}
\text{Minimize} & \quad \mathcal{J}(x) \\
\text{subject to} & \quad g(x) = 0 \\
& \quad x^{\min} \leq x \leq x^{\max}
\end{align*}
\]  

(2)

Any general formulation (1) can be easily transformed into a formulation (2) by introducing additional variables into the inequality constraints \( h(x) \leq 0 \) such that the inequality is transformed into an equality constraint \( h(x) + x_h = 0 \) and \( x_h \geq 0 \). Conceptually, this has the disadvantage of a larger number of equality constraints and an increased number of variables. From a formal point of view, however, this formulation has fewer parts to be defined in the sense that only two inequality types exist for variables, i.e. upper and lower variables limits.

(2) is transformed into an equality constrained optimization problem by adding the inequality constraints as weighted penalty terms to the objective function and creating the new objective function \( f(x) \):

\[
\begin{align*}
\text{Minimize} & \quad f(x) = \mathcal{J}(x) + \frac{\delta}{2} \sum_i \left( w_{i,\min} \left( x_i^{\min} - x_i \right)^2 \right) \\
& \quad + \frac{\delta}{2} \sum_i \left( w_{i,\max} \left( -x_i^{\max} + x_i \right)^2 \right) \\
\text{subject to} & \quad g(x) = 0 \\
\end{align*}
\]  

(3)

It is assumed that the numeric values for the weight variables \( w_{i,\min}, w_{i,\max} \) are set based on some logic built into the solution algorithm. This allows to apply a standard Newton algorithm to the Lagrange function of this modified optimization problem:

\[
\mathcal{L} = f(x) + \lambda^T g(x)
\]  

(4)

The first derivatives of this function are equivalent to the Karush-Kuhn-Tucker optimality conditions. The main step of the exact Newton algorithm is the solution of the Karush-Kuhn-Tucker optimality conditions as a sparse linear system of equations:

\[
\begin{bmatrix}
H^{(k)} & J^{(k)^T} \\
J^{(k)} & 0
\end{bmatrix}
\begin{bmatrix}
\Delta x^{(k)} \\
\Delta \lambda^{(k)}
\end{bmatrix}
= -
\begin{bmatrix}
r^{(k)} \\
g^{(k)}
\end{bmatrix}
\]  

(5)

with

\[
H^{(k)} = \frac{\partial^2 f(x)}{\partial x^2} \bigg|_{x^{(k)}} + \sum_{i=1}^{m} \left( \lambda^{(k)} \frac{\partial^2 g_i(x)}{\partial x^2} \right) \bigg|_{x^{(k)}}
\]  

(6)

and

\[
J^{(k)} = \frac{\partial g_i(x)}{\partial x} \bigg|_{x^{(k)}} ;
\begin{bmatrix}
r^{(k)} \\
g^{(k)}
\end{bmatrix}
= \left[ \frac{\partial f(x)}{\partial x} \bigg|_{x^{(k)}} + J^{(k)^T} \lambda^{(k)} \right]
\]  

(7)

The problem (5) is solved as follows with an iterative NR approach: Set \( k = 0 \); Assume some value for the weights in the augmented objective function. With given values for \( x^{(k)}, \lambda^{(k)} \), the linear system (5) is solved. The solution vector is then added to the previous solution: \[ [x^{(k+1)}, \lambda^{(k+1)}] := [x^{(k)}, \lambda^{(k)}] + [\Delta x^{(k)}, \Delta \lambda^{(k)}], \] \( k \) is increased by 1. Adapt the augmented objective function weights based on some algorithmic logic (This logic is not the issue of this paper). With the new solution \( x^{(k)}, \lambda^{(k)}, (5) \) is solved again, variables are updated, etc. Finally a converged solution is obtained, if the maximum absolute rhs-value of (5) is below a predefined tolerance \( \epsilon \).

2.2 Hard-coded, non user-enterable domain characteristics

The mathematical framework as discussed in the previous subsection describes a mathematical optimization problem domain which is still too general to formalize. Instead of choosing the definition of a partially separable optimization problem as used in Lancelot, another way is taken:

We define node and edge types, see Fig. 1 and assume that an edge has a node at each end. Edges are usually directed, i.e. an edge from node i to node j is not the same as the edge from node j to node i.
Each instance of a node and edge type owns the same set of associated elementary functions, unknown variables and numerically given constants. The numerical values of these unknown variables and constants can be different for each instance of these types. However, the number and types of variables will be identical for all instances of the same type.

Looking at the optimization problem framework as defined with (3) all unknown variables $x$ must be instances of node or edge type variables. The objective function $f(x)$ must be expressed as a sum of functions associated to all instances of specified node or edge types. The equality constraint set $g(x) = 0$ must be formulated as instances of functions of specified node or edge types.

In addition the domain includes two mandatory variable types for each node or edge type with associated domain laws. They have the following characteristics: The first variables type is called a "flow" variable type. All "flow" variables must obey the first domain law that the sum of all "flow" variable values of the same "flow" variable subtype at any node must be zero. In addition a second variable type exists which is called a "potential" variable. All "potential" variables of the same "potential" variable subtype must obey the second domain law: Any number of instances of edge and node types which are connected to each other must have identical "potential" variable values.

These two mandatory domain variable types are illustrated in Fig. 2.

"Flow" variables represent some good whose sum of inflow and outflow must be balanced at any node. "Flow" variables can represent any fluid or transportation good or in the area of electrical networks the electrical current.

The "potential" variables represent some form of balance at connected nodes: They must show identical values. "Potential" can be seen as height, pressure or in the area of electrical networks a measure for "voltage".

Figure 2: Mandatory flow and potential variables and domain laws

To summarize, the domain is restricted to a user defined number of "flow" and "potential" variables. Each node or edge type must own the same "flow" or "potential" variables subtypes. In addition the user defined node and edge types must have user defined associated node or edge type functions (called elementary node and edge functions) which must be formulated as mathematical expression of "internal" variables, "internal" constants and the above mentioned "flow" and "potential" variables.

All these subsets of "flow" and "potential" variables, all node and edge types, all "internal" variables and constants associated to these node and edge types, all elementary functions types associated to node and edge types must be defined by the user in a formalized way.

These domain rules together with the mandatory domain laws allow to derive a generic, robust and fast solution algorithm because all terms of the optimization problem must be written as (sums of) functions of only a few variables. This fact allows the automatic derivation of relatively simple first and second order derivative terms to be used in a Newton-type solution approach. Also it leads to a sparse linearized system representation which allows to simulate very large problem instances of the domain.

2.3 Software engineering phases

Fig. 3 gives an overview about the various software engineering steps during the development process.

Figure 3: From a symbolic domain definition to the domain solver code

The domain engineering approach presented in this paper is done in three phases:

Phase A The above defined node and edge type dependent domain characteristics must be entered in a high level form. The person who enters these domain parameters is typically a domain specialist who knows the domain both from a conceptual and mathematical point of view. In other words, this person specifies the problem parameters within the hard-coded, non user-changeable domain restrictions. The result of this phase A is called a "symbolic domain definition data set". In this paper Maple V is used as the high level tool which allows to enter these user defined domain parameters.

Phase B A Maple V program (called "symbolic domain code generation program") is executed which reads or interprets the "symbolic domain definition data set" given in phase A. It generates the algorithmic solution code within
the domain parameters as defined in phase A. This generated program code (called “domain problem solver”) is a fast Matlab code. Matlab has been chosen as the language in which this “domain problem solver” is coded (both the static, hard-coded and also the generated parts). Matlab is an excellent prototyping environment which can handle sparsity efficiently both from a code execution speed (relatively fast code) and also from the program developer point of view (easy sparsity and pointer coding). It is a code which can solve many problems within the chosen domain definition of phase A. Since the domain definition of phase A allows to set up concrete values for node type and edge type instance related parameter sets, these parameters data sets (called “formatted domain problem data”) can be user defined before the execution of this “domain problem solver”.

Phase C Now that the “domain problem solver” is available and ready to be executed, various “formatted domain problem data” sets for a concrete domain definition done in phase A can be given. At this point, the size of the network, the number of nodes and edges and associated types are given by concrete parameter values. Note, however, that these parameter values must fit to the “symbolic domain definition data set” given in phase A and the problem solver code generated in phase B. Conceptually a program is possible for phase C which checks this compatibility between these phase C input and phase B output files.

This formal definition of a domain oriented structural optimization problem has merits in itself since it allows a formal comparison between optimization problem formulation characteristics within the chosen domain. Without a formal approach, no structured discussion about features and problem solution capabilities of a presented algorithm is possible even among domain experts.

However from an end-user point of view (i.e. the institution which sets up the “formatted domain problem data” sets) the value of the formal domain specification is highest if there exists a robust and fast solution algorithm code for any “symbolic domain definition data set” and any associated “formatted domain problem data” sets.

In this paper we do not give details about the chosen solution algorithm. We restrict ourselves to the realization of the underlying domain solution characteristics: It is a Newton-type algorithm which corresponds to the solution of non-linear Karmans-Kuhn-Tucker optimality conditions of the optimization problem. The algorithmic challenge lies in the correct coding of first and second order derivatives of the Lagrangian, the setup and efficient solution of a large sparse linear system of equations, see (3).

Thus these first and second order terms characteristics must be automatically derived from a formal problem definition.

To clarify the concepts behind domain definition (what is user enterable during what phase of development process), generation of the algorithmic code parts dependent on the user entered domain definition (how and where does the user entered domain definition influence the hard-coded parts of a Matlab optimization problem solution code which are based on the solution of (5)) a concrete example of a 5 node electrical power transmission network is discussed in the following.

The number of nodes, i.e. 5, and the type of connections are much smaller than real-world examples (more than 1000 nodes with more than 1500 connections are typical) and should only help to understand the domain definition concepts. Also, the next subsection does not describe the data for the 5 node example but the node and edge types, elementary functions, variables which are modelled in the 5 node example.

3 PHASE A: THE OPF PROBLEM DOMAIN DEFINITION

OPF: Optimal Power Flow

3.1 OPF domain characteristics overview

In order to show the characteristics of the optimization problem in the power systems domain an example 5 node network is chosen. This network has the network topology, node and edge types as shown in Fig. 4.

![Figure 4: OPF 5-node network example](image)

In this network nodes can be either generator (symbol G) (providing electric power) or load nodes (symbol L) (consuming electric power). Nodes with both a generator and a load are assumed to be generator nodes. Nodes 1, 2 and 5 are generator nodes, nodes 3, 4 are load nodes.

From a modelling point of view a similar characterization is necessary for the edges connecting the nodes: An edge can represent an electric power transmission line, a voltage transformer and other power system elements. These edge types transport the generator-produced electric power to the loads. Fundamental laws (the Kirchhoff laws) dictate how much of the generated power is transported by every edge of an electric power transmission network.

In Fig. 4, all connected nodes represent transmission lines, i.e. 1-2, 1-4, 4-5, 2-5, 2-3.

The network model is only a model of the real network and usually we restrict ourselves to the so-called AC (Alternating Current) model which allows to use complex variables for the model formulation.

3.2 Symbolic domain definition

3.3 OPF domain laws, flow and potential variable definitions

As mentioned in the previous section the domain has hard-coded restrictions which are based on two domain laws for flow and potential variable types. These laws do not need to be explicitly given. They are automatically realized in the final problem solver code. However, for the chosen domain instance (in this case the OPF domain) we have to specify the names of flow and potential variables.
3.3.1 OPF domain flow and potential variable types

<table>
<thead>
<tr>
<th>Mandatory variable type</th>
<th>Symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node flow variable types</td>
<td>$L, I_f$</td>
</tr>
<tr>
<td>Node potential variable types</td>
<td>$\epsilon, f$</td>
</tr>
<tr>
<td>Edge flow variable types</td>
<td>$I_{eij}, I_{fij}$</td>
</tr>
<tr>
<td>Edge potential variable types</td>
<td>$e_i, f_i, e_j, f_j$</td>
</tr>
</tbody>
</table>

This formal definition implies that all node and edge types own these mandatory node or edge variables automatically. Also, node and edge related function expression (to be defined later) must use these symbols when referring to flow or potential variables.

These mandatory variables imply the following domain laws which are autogenerated for the problem solver:

$$L + \sum I_{eij} = 0 \quad \text{for all nodes}$$
$$I_f + \sum I_{fij} = 0 \quad \text{for all nodes}$$
$$\epsilon - e_i = 0 \quad \text{(if i is start node of edge ij)}$$
$$e - e_j = 0 \quad \text{(if j is end node of edge ij)}$$
$$f - f_i = 0 \quad \text{(if i is start node of edge ij)}$$
$$f - f_j = 0 \quad \text{(if j is end node of edge ij)}$$

In the OPF domain these laws are the so-called Kirchhoff laws. They are fundamental to all aspects of power transmission and must be considered in any model dealing with electrical current or power transfer.

3.3.2 General principles in setting up node and edge types

The high-level domain definition process consists of setting up all node types ($N_1, N_2, \ldots$), all edge types ($E_1, E_2, \ldots$). This conceptual definition is the key to the associated setup of elementary function, unknown internal variable and constant parameter sets.

For each node type ($x$), set up the following information: Elementary node function types ($g_n, g_{n2}, \ldots$), unknown node variable sets ($x_{n1}, x_{n2}, \ldots$), constant node parameter sets ($c_{n1}, c_{n2}, \ldots$). The unknown node variable sets do not need to include the mandatory “flow” and “potential” node variables. These mandatory node variables are automatically included as variables of the node types.

For each edge type ($y$), set up the following information: Elementary edge function types ($g_e, g_{e2}, \ldots$), unknown edge variable sets ($x_{e1}, x_{e2}, \ldots$), constant edge parameter sets ($c_{e1}, c_{e2}, \ldots$). The mandatory “flow” and “potential” variables of edges are also automatically included as unknown edge variables and do not need to be specified here.

These sets of unknown node and edge variables (in addition to the mandatory node and edge variables) in the chosen domain are used

1. to allow the symbolic computation of first and second derivatives of the associated elementary function and the subsequent Matlab code generation of all “problem solver” code parts which are dependent on these elementary functions, their first and second order partial derivatives.
2. to generate Matlab code with computes correct pointers and values for instances of all node and edge types with associated variables and elementary functions. Instances of variables and functions of these variables can be found both in the linear system matrix and the right hand side terms of (5).

Thus all variable sets and all elementary functions of each node and edge type (including the mandatory node and edge variables) are assumed to be instantiated for all individual instances of the associated node or edge type.

The term “constant parameter set” is used to describe those symbols in an elementary function which are assumed to be numerically given for all instances of the associated node or edge type.

The elementary node and edge functions are functions of only node or only edge related variables and constant parameters for which a symbolic tool such as Maple V can compute first and second derivatives.

It is assumed that all node and edge types are exclusive: For example, no node can be associated to two different types, no elementary function of a node type 1 can be instantiated for another node type 2.

3.3.3 OPF domain node and edge types

Node type sets:

$N_1, N_2, N_3 = \{\text{load nodes, generator nodes, swing nodes}\}$

Edge type sets: $E_1 = \{\text{transmission lines}\}$

3.3.4 OPF domain elementary node and edge function types

| Type | $|m|$ | Elementary function type |
|------|-----|--------------------------|
| $g_{n1}$ | 1 | -$P_n$ - ($e_i$ + $f_i$) |
| $g_{n2}$ | 1 | -$Q_n$ - ($f_i$ - $e_i$) |
| $g_{n3}$ | 1 | $P$ - ($e_i$ + $f_i$) |
| $g_{e1}$ | 1 | $I_{eij} - (g_{ij}^\ell + b_{ij}^\ell f_i + b_{ij}^s e_j + g_{ij}^f f_j )$ |

Note that the symbols in these elementary functions are either unknown variables (including flow or potential variables) or constant parameters. Throughout this paper the symbols representing the constant parameters are marked with an upper index $^0$. Also, an index $x$ indicates node related sets, an index $e$ indicates edge related sets. The second column of (10) marked with $[m]$ shows the numbers of the elementary function types for each node or edge type. This number $m$ will be referred to later in the so-called domain group functions.

3.3.5 OPF domain unknown variable sets and constant node parameter sets

The sets are as follows: $x_{n1}$: $\{V\}$, $x_{n2}$: $\{P, Q, V\}$, $x_{n3}$: $\{P, Q, V\}$, $c_{n1}$: $\{P_L, Q_L^\ell\}$, $c_{n3}$: $\{\}$ and $c_{n2}$: $\{\}$ are empty sets, $x_{e1}$: $\{Iabs_{ij}\}$, $c_{e1}$: $\{g_{ij}^\ell, g_{ij}^s, b_{ij}^\ell, b_{ij}^s\}$.
3.4 Definition of domain optimization problem

After specifying all node and edge type related information in a formal way (Maple V has been chosen as the language in which this information is specified) the optimization problem terms must be synthesized and formally specified. To allow certain flexibility in setting up the optimization problem and still observing a strong formalism, so-called group function structures must be defined. Only instances of these group function types will be used for the formal definition of the optimization problem.

3.4.1 Hard-coded domain group function structure

As will be discussed later, these domain group function types are structurally hard-coded. However user defined parametrization is possible within these hard-coded structures.

These group domain functions allow to describe optimization problem instances in a high level formalized way: In this domain design the optimization problem can only be defined in terms of these parametrizable group functions. At the current stage of the domain development process we restrict ourselves to four domain group function types whose structure is hard-coded:

<table>
<thead>
<tr>
<th>Type</th>
<th>Instance structure</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G_1$</td>
<td>$G_{1[N_i]}[m][k] = g_{N_i}[m][k]$</td>
<td>${i, m}$</td>
</tr>
<tr>
<td>$G_2$</td>
<td>$G_{2[E_i]}[m][k] = g_{E_i}[m][k]$</td>
<td>${i, m}$</td>
</tr>
<tr>
<td>$G_3$</td>
<td>$G_{3[N_i]}[m] = \sum_{k \in N_i} g_{N_i}[m][k]$</td>
<td>${i, m}$</td>
</tr>
<tr>
<td>$G_4$</td>
<td>$G_{4[E_i]}[m] = \sum_{k \in E_i} g_{E_i}[m][k]$</td>
<td>${i, m}$</td>
</tr>
</tbody>
</table>

(11)

${i, m}$ must be understood as follows: Several sets of indices $\{i, m\}$ can be given to indicate the existence of instances of this type. In each $\{i, m\}$, $i$ refers to the i-th node or edge type (node type association is only possible with $G_1$ and $G_3$, edge type association only with $G_2$ and $G_4$) and $m$ refers to the m-th instance of the associated node or edge elementary function. All possible numbers for $i$ and $m$ of which selections can be made are specified in (10): $i$ points to the first column in (10) indicating the i-th node or edge type, $m$ points to the second column of (10) indicating the m-th elementary function of this node type $i$.

$G_{1[N_i]}[m][k]$ represents a domain group function for every individual node instance $k$ of type $N_i$ of the m-th elementary node functions of type $g_{N_i}[m]$. Each instance of this domain group function $G_1$ is uniquely given by an index set $\{i, m\}$. $G_{2[E_i]}[m][k]$ represents a domain group function for every individual edge instance $k$ of type $E_i$ of the m-th elementary edge functions of type $g_{E_i}[m]$. Each instance of this domain group function $G_2$ is uniquely given by an index set $\{i, m\}$. $G_{3[N_i]}[m]$ represents a domain group function for the sum of the m-th elementary node function of type $g_{N_i}[m][k]$ over all instances $k$ of nodes of type $N_i$. Each instance of this domain group function $G_3$ is uniquely given by an index set $\{i, m\}$. $G_{4[E_i]}[m]$ is a domain group function for the sum of the m-th elementary edge functions $g_{E_i}[m][k]$ over all instances of edges of type $E_i$. Each instance of this domain group function $G_4$ is uniquely given by an index set $\{i, m\}$.

Parametrized instances of these hard-coded domain group functions will be used in both the objective function part and also the equality constraint parts of the optimization problem.

In a formal definition of the domain instance we need to know what group function types are instantiated. To achieve this only index pair related information $(i, m)$ is required. The meaning behind the index values is hard-coded and known by the "symbolic domain instance code generation program".

It is obvious that the set of domain group functions together with the flow and potential variable and associated domain laws limits any domain instance. The code generator can only generate code based on these four hard-coded group function types. Thus the functionality of the domain is restricted to instances of these group functions. However, it is free with respect to the underlying elementary functions for which only first and second order partial derivatives must exist.

These group function restrictions and fundamental domain laws correspond to the conceptual boundary of the chosen domain. An expansion of the domain would require an expansion of the group functions and a subsequent realization of autogenerated domain solver code which can handle these new group functions.

In the following, if the term $G_0\{o\}$ is used, the o-th pair of indices defined in $G_i$ is meant.

User-defined OPF domain group function instances

<table>
<thead>
<tr>
<th>Type</th>
<th>User data ${{i, m}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G_1$</td>
<td>${{(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3), (3, 4)}}$</td>
</tr>
<tr>
<td>$G_2$</td>
<td>${{(1, 1), (1, 2), (1, 3)}}$</td>
</tr>
<tr>
<td>$G_3$</td>
<td>${{2, 4}, [3, 5]}$</td>
</tr>
<tr>
<td>$G_4$</td>
<td>${{1, 3}}$</td>
</tr>
</tbody>
</table>

For the OPF domain the parametrized group functions are needed as defined in (12).

In a pair $(i, m)$ specified in the second column of (12), the index $i$ specifies the associated node or edge type $i$ and the m-th elementary function of this node type $i$.

These formal group function definitions must be used to define the objective function and equality constraint types as discussed in the next two subsections.

3.4.2 Hard-coded domain objective function structure

$$F(x) = \sum_o G_3[o] + \sum_o G_4[o] \Rightarrow \text{Minimum}$$

An objective function must be represented by the sum of user-defined instances of domain group function $G_3$ (for node types) and the sum of user-defined instances of domain group function $G_4$ (for edge types). This fact of using the sum is hard-coded and cannot be changed by the domain user. Also, the fact that a minimum (and not a maximum) is optimized is hard-coded. However, the user can specify which instance of the $G_3$ or $G_4$ group function defined in (12) will be used in the autogenerated problem solver code. For example, the parameter 2 in $G_3[2]$ means to use the second pair of indices defined for $G_3$ in (12), i.e. $(3, 5)$.

In a formal definition given by the user only the numbers $[x]$ and the associated group function number 3 or 4 must be given.

User-defined OPF objective function instances

Two typical objective function types for the total network active power network losses are given. These two objective functions are formally defined as follows using the group functions defined in (12):
The empty entries indicate that no instances of this group function are needed in the OPF domain objective function definition. Note, that only either the first two or only the second two group functions can be specified in a concrete OPF domain specification (For domain specialists it is clear, however, that although both objective functions are functionally completely different, both lead to the same optimization problem minimum if the problems converge.).

3.4.3 **Hard-coded domain equality constraints structure**

Any domain instance is restricted to user-defined instances of the group function types 1 and 2. I.e. the user can define the equality constraints of the optimization problem only as $G1[m] = 0$ or $G2[m] = 0$ where the user enters the desired numbers for $m$.

The first equality constraint type ($G1[m] = 0$) allows to specify instances of node type oriented domain group functions $G1$ as equality constraints. The second type ($G2[m] = 0$) allows to specify instances of edge type related domain group functions $G2$ as equality constraints. The domain laws which in the case of the OPF are the Kirchhoff laws are hard-coded domain equality constraints and must not be specified here.

In a formal definition of these equality constraints we need to know what group function types of ($G1$, $G2$) are to be instantiated as equality constraints for the domain.

**User-defined OPF equality constraints parameters**

One (or many) set of equality constraints for the OPF domain can be formally defined as follows using the OPF group functions defined in (12):

<table>
<thead>
<tr>
<th>Group function type</th>
<th>User data $m$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G1[m] = 0$</td>
<td>${(4), (2), (3), (4), (5), (6), (7), (8), (9), (10)}$</td>
</tr>
<tr>
<td>$G2[m] = 0$</td>
<td>${(4), (2), (3)}$</td>
</tr>
</tbody>
</table>

3.4.4 **Hard-coded domain inequality constraints structure**

Since we restrict ourselves to upper and lower limits of only variables as inequality constraints the only information for the domain is the type of variables $x$ and the type of limit (upper, lower or both). The unknown variable sets which have been formally defined are a superset of the variables which can be limited. Thus we have to give all those members of the sets of unknown variable which are limited either by a maximum, a minimum or both limits.

**User-defined OPF inequality constraints parameters**

<table>
<thead>
<tr>
<th>Node/Edge type</th>
<th>User: Limit type</th>
<th>User: Limited symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N_1$</td>
<td>$\text{min}/\text{max}$</td>
<td>$V_i$</td>
</tr>
<tr>
<td>$N_2$</td>
<td>$\text{min}/\text{max}$</td>
<td>$P_i, Q_i, V_i$</td>
</tr>
<tr>
<td>$N_3$</td>
<td>$\text{min}/\text{max}$</td>
<td>$P_i, Q_i, V_i$</td>
</tr>
<tr>
<td>$E_1$</td>
<td>$\text{max}$</td>
<td>$\text{tab}, s_{ij}$</td>
</tr>
</tbody>
</table>

$\text{min}/\text{max}$ indicates that both upper and lower limit values must be respected, $\text{max}$ alone indicates the needed upper limit observation.

Note that the variables given in (16) must be subsets of the unknown variable sets of the associated node or edge types, see section 3.3.5.

4 **PHASE B: SYMBOLIC DOMAIN INSTANCE CODE GENERATION PROGRAM**

The complete “symbolic domain definition” as given in the previous section is defined in Maple V language. In addition a Maple V based “symbolic domain code generation program” has been developed which can read and/or interpret any user-defined “symbolic domain definition” (see Fig. 3). This Maple V program knows the syntax and semantics of the “symbolic domain definition” data file. With the given symbolic data this Maple V program generates all those Matlab code pieces of the “domain problem solver” code which are dependent on the user entered domain parameters. In addition to this code generation there must also be a strong correspondence between the “formatted domain problem data” and the parameter sets as given in the symbolic “domain definition”. The generated code must also be merged with those hard coded domain parts which do not change for new “symbolic domain definition” sets.

This “domain problem solver” consist of the following main code pieces whose programming logic is independent on the actual user entered domain data. Independent means that the high-level functionality of the code blocks specified below does not change for different domain definition sets. The code blocks, however, must be generated dependent on the parameters the user enters for the chosen domain. The most important code generation steps are as follows:

1. Generate the code which reads the node and edge types and associated parameters and initial values for all unknown variables (including flow and potential variables).

2. Generate the code which computes the sparse linear system matrix elements and right-hand-side vector values of (5):

   (a) Generate the code which computes the group function terms associated to the equality constraints and to the objective function including the code which computes first and second order partial derivatives.

   (b) Generate the code which compute the equations and all associated terms associated to the flow and potential variable domain rules including the code for first order partial derivatives (no second order partial derivatives exist).

The experience gained during the development process of Maple V code which generates Matlab code can be summarized as follows:

The process of writing this Maple V code is most complex for those problem solver code parts which link node function types with edge functions types. This affects mainly the code parts 2(a) and 2(b). The generated problem solver code must be able to compute instances of functions of node and edge related variable sets correctly. Thus the code must compute correct pointers from node or edge related element instances to variable and equation locations within the large,
sparse linear system of equations to be solved by the linear system solver.

First and second order derivatives of all elementary functions are computed by the Maple V program. These parts are not so critical during the domain development process since these functions are decoupled and do not depend on each other. Maple V code has been developed which translates the computed first and second order derivative terms into correct Matlab code. The generated Matlab code uses vector-type operations, i.e. the principle to apply the same node and edge type autogenerated first and second order derivatives to all instances of a node or edge type is followed throughout the whole code generation phase. The generated Matlab code includes not a single “case” or “if” statement. The only place where cases are allowed is during phase A where the domain node and edge types are defined.

The development of Maple V code instantiating group functions as Matlab code is quite complex because group functions can be used more than once and it is up to the user to define where what group functions will be used as equality constraints or objective functions parts.

One observes that code errors in this part often propagate in many places in the autogenerated problem solver. This has the effect that the generated code is either totally wrong or basically perfect. There is not much space for intermediate solutions because of the automatic propagation of errors in many different places of the solver code.

As long as variables and node/edge type symbols in the domain definition are named in such a way that they have meaning to the domain expert the generated code parts can be checked for algorithmic correctness.

This process of defining domain parameters sets on a high conceptual level and generating code from this high level definition influences both the code quality of the autogenerated code and the code quality of the “Symbolic domain code generation program” in a very positive way. According to our experience the software engineering process has a very high chance of generating perfect code within the chosen domain and the chosen first and second order derivative terms based solution algorithm.

5 CONCLUSIONS

Domain engineering as presented in this paper can be seen as a process to specialize or instantiate a structured general description or framework of a problem. The domain is called “power systems optimal power flow domain”. This name is based on the fact that the design of this domain includes on one side the hard-coded characteristics of the fundamental laws of static power systems, the Kirchhoff laws. On the other side typical optimization features of power networks are also included.

The key aspect of the presented domain engineering is a formal description of the generic design and a phased engineering process of this domain which allows to specify parameterized domain instances and also allows to get an algorithmic solution of these instances based on a few, compact information pieces. Once the generic design for a domain is coded a user has to learn to use this design code by specifying domain parameter data. This learning process is especially hard because the material which allows to learn about the domain design is very difficult to formulate by the domain designer and no standards how to do this exist yet.

According to the simulation results obtained with autogenerated code and large power system optimal power flow networks this domain engineering concept seems to represent a practically useful compromise for the following often conflicting requirements:

1) High quality code by using code generation and code quality checks inherent in the domain development process.
2) Robust algorithmic code by applying a Newton-type algorithm.
3) Fast execution times by using code generation optimization features of the symbolic computation tool (Maple V), by generating sparse matrices and using fast sparse linear system solvers (Matlab 4.2).
4) Flexible and manageable domain software development, domain expansion and maintenance by introducing several separated phases into the domain engineering approach.

References


