Object Oriented Design of a Transient Analysis Program

Taku Noda

***Abstract*— This paper presents a design methodology of an electromagnetic transient analysis program based on an object-oriented approach. Two important features of the object oriented programming are (i) abstraction of data and their procedures using classes and (ii) derivation of a new class from an existing class using the concept of inheritance. In the proposed design methodology, first the base class, from which all kinds of circuit components are derived using inheritance, is defined. When a specific circuit component is derived from the base class, the parameters, the internal variables, and the behavior of the component are defined and described in the code defining the derived class. This means that each circuit component manages its data and knows how to initialize and update itself, that is, the circuit solver does not have to manage and know them. In this way, we can separately write the code of the circuit solver and that of circuit components, and thus, coding is simplified and code maintainability is enhanced. Especially, adding a new component does not require modifications in the circuit solver code. Using the proposed approach, a new transient analysis program named "XTAP" (eXpandable Transient Analysis Program) has been created. Some practical implementation issues are also presented in the paper.**

***Index Terms*— Circuit simulation, Electromagnetic transient analysis, Object oriented programming, Power systems, and Software maintenance.**

I. Introduction

POWER engineers have started using object-oriented methods for power system simulations in the late 80's. Object-oriented methods are now used in various aspects of power system simulations [1]–[13]. In [1], an object-oriented implementation of a load-flow program using the Objective-C language is illustrated. How power system components such as transmission lines, transformers, and etc. are modeled using classes is shown, and it is concluded that the object-oriented approach reduces the efforts of programming. Similar results are obtained in [2]–[4] using the C++ language. In addition, [2] shows how to handle sparse matrices using C++, and [4] deals with large-scale networks including FACTS controllers. Object-oriented modeling of a power system for dynamic stability type simulations is proposed in [5], and for obtaining the steady-state solution of a nonlinear power network in [6], both using C++. An object-oriented modeling methodology, called DCOM (Distribution Circuit Object Model), for a distribution analysis system is reported in [7]. As an application to other than programming, an object-oriented concept was used to develop a common database which generates input data files for different simulation programs ranging from load flow to electromagnetic transients [8]. The database stores parameters of power system components in various aspects and appropriately processes the parameters to generate an input file for a specific type of power system simulations. Reference [9] reports a development of graphical user interfaces for power system simulations using object-oriented programming. The development of a fault locating system using an object-oriented approach is described in [10].

Two important features of the object-oriented programming are (i) abstraction of data and their procedures using "classes" and (ii) derivation of a new class from an existing class using the concept of "inheritance". An efficient implementation of an electromagnetic transient program is reported in [11] based on a partly object-oriented approach using Fortran 95. The approach only uses (i) for optimizing the execution speed.

This paper presents a design and programming methodology of an electromagnetic transient analysis program based on a fully object-oriented approach. The proposed object-oriented methodology first defines a base class called "Branch", from which all kinds of circuit components are derived using inheritance. The Branch class has data and their procedures which are common to all circuit components but no specific model parameters and behaviors are defined. Using inheritance, specific model parameters and behaviors are added to the Branch class, and as a result a specific circuit component is derived. From the Branch class, not only simple components like resistors, inductors, capacitors, and etc. but also complex components are derived. In order to carry out an electromagnetic transient simulation, each circuit component has to be initialized preceding the time step loop and then updated at each time step (if the component is nonlinear, then updated at each iteration step). The proposed design methodology significantly simplifies these processes. Since all circuit components are derived from the common base class Branch, all circuit components can be put into an array of the Branch class. Once all components of a circuit are stored in the array, the circuit solver can issue an initialization or updating message to each element of the array. When issuing these messages, the circuit solver does not have to specify how to initialize or how to update and even does not have to know what is each element in the array. On the other hand, each element knows how to initialize or update itself, since the initialization and updating processes are defined when each component has been derived from the Branch class. This enables separately writing the code of the circuit solver and that of circuit components and thus significantly

T. Noda is with the Electric Power Engineering Research Lab., CRIEPI (Central Research Institute of Electric Power Industry), 2-6-1 Nagasaka, Yokosuka-shi, Kanagawa-pref. 240-0196, Japan (e-mail: takunoda@ieee.org).

Presented at the International Conference on Power Systems Transients (IPST'07) in Lyon, France on June 4-7, 2007
Paper No. IPST07 – (Paper ID: 111)

simplifies the entire code structure. Code maintainability is also enhanced. Especially, adding a new component or a user-defined component becomes an easy task. When adding a new component, the circuit solver does not have to be revised and only the code for deriving the new component from the Branch class is required.

Using the proposed object-oriented design methodology, a new transient analysis program named "XTAP" (eXpandable Transient Analysis Program) has been created. The C++ language is used in the implementation, and some practical implementation issues are also presented.

II. Object-Oriented Programming

Conventional procedural programming languages are used to place commands which are successively executed to process data. Thus, procedures (commands) are of main concern. On the other hand, the object-oriented programming languages consider that the target objects themselves are equally important as well as their procedures. To use an existing object, we do not have to know how the object works. We just send a message to the object, and the object understands the message and works. This situation is similar to that when you want to watch television you only have to know how to use a remote controller (how to send a message) and you don't have to know how the electronic circuits work inside the television set.

To realize this way of thinking in programming, the object-oriented programming languages introduce the following concepts:

- abstraction by class structuring
- message passing
- encapsulation
- derivation of a new class by inheritance

We use class structuring to define an object as a "class". In the class definition, not only the data structure of the object but also procedures for the data, called "methods", are defined. The data structure and their procedures are now represented by an abstract entity called a class. To use an object, an instance is generated from the class definition of the object and actual computer memory is allocated for the instance. A method of the object is invoked by passing a message to the instance from outside. The internal data of an instance are not accessible from outside, and this structurally prevents from modifying the internal data by mistake and is called encapsulation. In order to define a new class which is a revised or expanded version of an existing class, we use inheritance. Rather than defining such a class from scratch, the revised or expanded class is derived from the original class by adding the revised or expanded portion of data and methods, and this is called inheritance.

In the proposed design methodology, circuit components are modeled using the object-oriented approach mentioned above.

III. Design

A. Overall View

Fig. 1 shows the overall view of the proposed transient analysis program design. The circuit solver itself is an object, and the design of the class defining the circuit solver is described in Section III-D. The circuit components, including not only simple ones like resistors, inductors, capacitors, and sources but also complicated ones like transmission lines and generators, are also considered to be objects. They are derived from the common base class "Branch" as described in Sections III-B and III-C. The circuit solver has an array of pointers to the instances of the circuit components. The array is named "pBranches". Since all circuit components have been derived from the common class Branch, they can be put into the single array. Actually, pBranches is an array of pointers to Branch instances.

B. Circuit Component Class

List 1 shows the C++ code defining the "Branch" class. First comes the string "Name" which contains the name of this circuit component. Next comes the integer "NumPhases" indicating the number of phases, that is, the number of terminal pairs attached to this component. "LhsNodes" and "RhsNodes" are the indices respectively of the left-hand side and the right-hand side nodes connected to this component. The Branch class has some other miscellaneous variables in the part (A). They are omitted here, since they are irrelevant to the main features of Branch. The double-precision floating-number variables "v_br" and "i_br" respectively contain the branch voltage and the branch current of this component at the previous calculation step. They are necessary for numerical integration of dynamic elements such as inductors and capacitors. "g_sub" is the container of the equivalent conductance value of this component, if exists. "s_sub" is used to hold the value of a current source if this component has such a current source. If the component is a voltage source, the voltage value is held in "s_sub" instead. The matrix/vector versions of v_br, i_br, g_sub, and s_sub are defined in the part (B) for the multiphase case, but they are omitted here. In this implementation, the template class "matrix" has been prepared separately for representing matrices and vectors and easily performing matrix algebra. "NodeNames" is a vector of strings defined as a static variable. According to C++ rules, the memory for NodeNames of all instances are shared. This is used for keeping a record of all node names connected to all circuit component instances. "pBranches" is a vector of pointers to Branches. Since this is

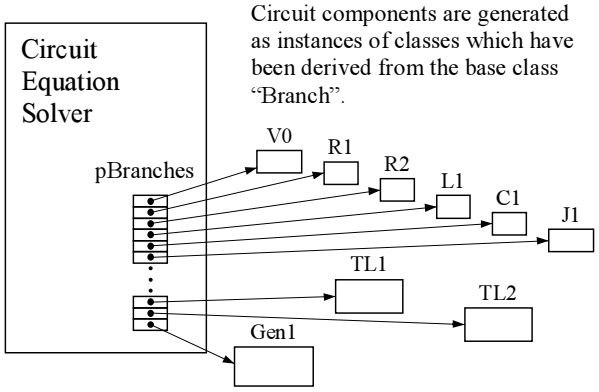


Fig. 1. Overall view of the proposed transient analysis program design.

defined as static, this is also shared by all instances and used for keeping the addresses of all instances.

The next part is the definition of methods. The constructor and the destructor are defined first. The “Initialize()” method is invoked by the circuit solver before entering the time step loop for the initialization of this component. The “Update()” method is called at each calculation step in the time step loop for updating the internal state of this component. If this is a nonlinear component, Update() is called at each iteration step for solving nonlinear circuit equations. Otherwise, at each time step. At this stage, since no specific behavior of this component can be defined, these methods are empty and defined as virtual. By defining as virtual functions, they will be overwritten when a class for a specific circuit component is derived from this class by inheritance, and overwriting methods will properly be called at run time. The part (C) includes miscellaneous methods common to all kinds of circuit components, but not shown here.

It should be noted that in the definition of this base class Branch only variables and methods which are common to all kinds of circuit components are defined. At the time of inheritance, variables used by a specific circuit component will be added and the virtual functions will be overwritten to implement specific component behavior.

List 1. Definition of the Branch class.

```
class Branch
  {
  protected: // ---------------------------------

  string
    Name; // name of this branch used as id.
  int
    NumPhases; // number of phases.
  int
   *LhsNodes, // left-hand side node indices.
   *RhsNodes; // right-hand side node indices.
  ... (A) ...

  double // for single-phase branches.
    v_br, // branch voltage at the previous step.
    i_br, // branch current at the previous step.
    g_sub, // equivalent conductance.
    s_sub; // current or voltage source value.
  matrix<double>  // for multiphase branches.
  ... (B) ...

  static vector<string>
    NodeNames;  // contains all node names.
  static vector<Branch*>
    pBranches;  // pointer to branches.

  public: // -----------------------------------

  Branch();

  virtual ~Branch() {}

  virtual void Initialize() {}

  virtual int Update( double, int ) {
    return( 0 );
    }

  ... (C) ...
  };
```

C. Deriving a New Class for a Specific Circuit Component

In the proposed design, the circuit solver has no pre-defined components. Even simple components like resistors are not pre-defined. In this Section, how to derive a specific circuit component from the base class Branch is illustrated using the code for the linear inductor component as an example.

List 2 shows the code for the linear inductor component. The code derives the class “Inductor” from Branch using inheritance. It adds the double-precision variable “L_Val” to keep the inductance value. The constructor is overwritten to set necessary information such as component category, number of phases, instance name, and inductance value. The destructor is overwritten simply by an empty method, since there is nothing to do when the memory allocated to this object is released. The “Initialize()” method is overwritten, and the equivalent conductance value for a given inductance value and a given time step interval is calculated and set to “g_sub”, where the code assumes the trapezoidal rule of integration [14]. In the “Update()” method, the equivalent source value “s_sub” is updated based on the trapezoidal rule. The variable L_Val that is specific to this component has been added, and the methods Initialize() and Update() have been overwritten to describe the model specific behavior.

In the same way, other circuit components are defined. Fig. 2 shows the class hierarchy of the circuit components. All kinds of components are derived directly from the base class Branch. Multiple inheritance is possible to derive a circuit component, but it is not recommended due to the overhead of execution time especially in the case when the component is used many places in a circuit. However, if the following three conditions are satisfied, multiple inheritance may be used.

- The component to be developed is considerably complicated.

List 2. Code for the linear inductor component.

```
class Inductor : public Branch
  {
  private: // ----------------------------------

  double
    L_Val; // Inductance value.

  public: // -----------------------------------

  Inductor( string Name1, double L_Val1 ) {
    SetCategory( LinDyn ); // linear, dynamic comp.
    SetNumPhases(1); // number of phases = 1.
    SetName( Name1 ); // instance name.
    L_Val = L_Val1; // inductance value.
    }

 ~Inductor() {}

  void Initialize( double Delta_t ) {
    g_sub = Delta_t/( 2.0*L_Val );
    }

  int Update( double Time ) {
    s_sub += 2.0*g_sub*v_br;
    return( 0 );
    }
  };
```

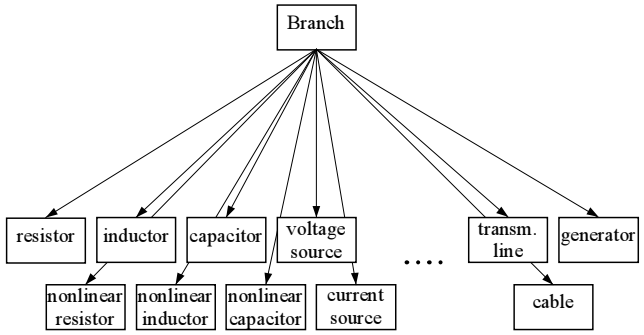


Fig. 2. Class hierarchy of the circuit components.

- The class of the component can be defined by adding small amount of code to an existing component class.
- The component is used in one place or a few places in the circuit to be simulated.

For instance, if the class definition of an elaborate synchronous generator model is available and if you would like to slightly change the excitation method, then deriving a new class for the modified generator model from the existing synchronous generator class could be preferable, rather than from the Branch class. When this model is used in one place or in just a few places in the circuit to be simulated, the overhead of execution time can be small. Furthermore, if an error exists in the original model and it is fixed, the same problem is fixed at the same time also in the derived model. Thus, use of multiple inheritance for deriving a new circuit component should be decided with the considerations above.

D. Circuit Solver Class

List 3 shows the code defining the class of the circuit solver. The class is named "SimuCase" which stands for a Simulation Case. In the class definition, at first, the string "Name" is defined to keep the name of a simulation case. The next section of the definition consists of the containers of the network information. The vector "NodeNames" of strings contains the names of all nodes in the circuit. The array "pBranches" is the same as the one used in the class Branch. The integer variables "NumBranches" and "NumNodes" are the number of branches (circuit components) and the number of nodes in the circuit respectively. Next comes the section of network matrices and vectors. "F" is the matrix obtained as a result of linearization of circuit equations. The modified nodal analysis [15] is used for the formulation of the circuit equations in this implementation. The class "SparseMatrix" has separately been prepared for efficiently storing the elements of a sparse matrix in memory utilizing indexing and also for calculations (see Section IV-A). The column vectors (n by 1 matrices) "x" and "y" are the solution vector and the right-hand side vector of the modified nodal equations respectively. Some other miscellaneous member variables are defined in the part (A).

The next section consists of member function definitions. Only important methods are shown here, and the other methods are omitted in the part (B). After the definition of the constructor and the destructor, the method "OutputRequest()" is defined. It assigns which simulation results are stored in the output file. The method "Initialize()" first gets the address of pBranches from the Branch class and then initializes all circuit components, that is, the Initialize() methods of all Branch instances are called, preceding the time step loop. It also calculates the initial value of the matrix F. Finally, the method "Start()" starts the time step loop of a transient simulation. In the time step loop, the Update() methods of all circuit components (Branch instances) are called to update their internal states. The circuit equations are solved by the Newton-Raphson iteration in this implementation. Linear components are updated only once at each time step, while nonlinear components are updated at each iteration step. This time sequence showing how the methods of the SimuCase class are called is illustrated in Fig. 3.

List 3. Code defining the class of the circuit solver.

```
class SimuCase
  {
  private: // ----------------------------------

  string
    Name; // name of this simulation case.

  // Network Info.
  vector<string>
    NodeNames; // strings of all node names.
  Branch**
    pBranches; // array of pointers to Branches.
  int
    NumBranches, // number of branches.
    NumNodes; // number of nodes.

  // Network Matrices and Vectors.
  SparseMatrix
    F; // MNA matrix.
  matrix<double>
    x,  // solution vector.
    y;  // RHS vector of the MN equations.

  ... (A) ...

  public: // ----------------------------------

  SimuCase( string, double, double, string );

 ~SimuCase();

  ... (B) ...

  void OutputRequest( ... ); // select outputs.

  void Initialize(); // initialize all components.

  int Start(); // start the time step loop.
  };
```

E. Overall Flow of a Simulation

Currently, the developed program XTAP reads input data in the form of a text file. Simulation parameters such as time step, maximum calculation time, and etc. are read first, and then data of circuit components follow. Each time the data of a component have been read, its instance is created from the class of the component. After reading the data of all

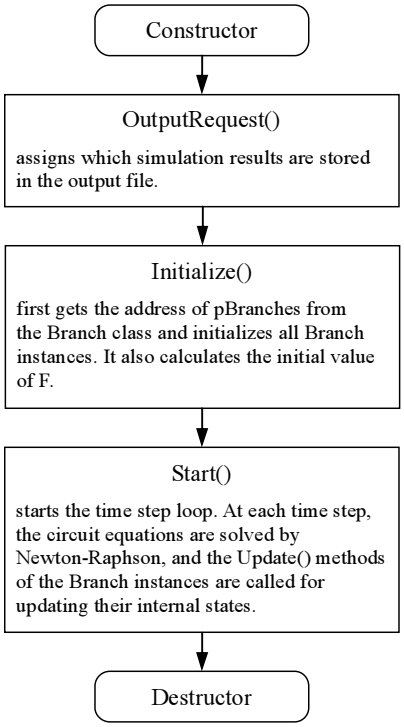


Fig. 3. Time sequence showing how the methods of SimuCase are called.

components, the instance of SimuCase is generated and the pointers pBranches to the circuit components are transferred from Branch to the instance of SimuCase. Next, the section specifying output variables is read from the input file, and the instance of SimuCase gets the output specification using the OutputRequest() method. The input file ends here. Then, the instance of SimuCase performs the initialization process by the Initialize() method and starts the transient simulation by the Start() method as shown in Fig. 3.

IV. Implementation Issues

A. Handling of Sparse Matrices

In this implementation, a sparse equation solver called PARDISO [16] is used to solve the system of circuit equations

$$Fx = y.$$

PARDISO, developed at the University of Basel, can be used for solving a large sparse symmetric and unsymmetric linear system of equations on shared-memory multiprocessors. Currently, XTAP is used on a computer with a single-core processor, but tests on a multi-core (possibly 8 to 16 cores) shared-memory computer is scheduled in 2007. Since PARDISO is written in FORTRAN, the following techniques are important to efficiently pass the matrix F and the vectors x, y from the main routine of XTAP to PARDISO. Passing the vectors x and y is straightforward. The pointers to x and y are passed to PARDISO. As mentioned above, F is stored in the class SparseMatrix. It has, as its member variables, arrays that store the elements of F using indexing and the pointers to the arrays. The byte sequence (format) of the arrays is designed so as to be exactly the same as that of PARDISO. Thus, just by passing the pointers, the matrix F can be passed to PARDISO. The memory allocated by SparseMatrix is not copied and directly used by PARDISO.

B. Variable-Size Arrays

The Standard Template Libraries (STL) of C++ provides useful data containers such as “vector”. A vector container (not a mathematical vector) is a variable-size array of any type, whose memory is linearly allocated. The memory allocation for a vector container is dynamic and automatic, and the user does not have to take care of it. Before reading an input file, XTAP cannot know how many nodes and how many circuit components exist in the circuit to be simulated. Thus, vector containers are useful to store these data. However, if it is used inside the time-step loop, the execution speed may be slowed down depending on the implementation of the vector class of a compiler used. To avoid this, only when reading an input file, vector containers are used to store data. Then, before entering the time-step loop, the pointers to the contents of the vector containers are copied to ordinary arrays, and they are used in the calculations inside the time-step loop. The array “pBranches”, which contains the pointers to the instances of all circuit components, is a good example for this. In this way, we can avoid the use of fixed-dimension arrays, which have been problematic in old versions of EMTP, and we can enjoy the ease of programming due to the variable-size feature at the same time not losing its execution speed in the time step loop.

A list container is a variable-size array to which a new entry is inserted at any position. List containers are used in the XTAP code for analyzing input data, for instance, handling parameterized subcircuits. Since they are not used inside the time-step loop, the speed-up process mentioned above is not required.

V. Expandability vs. Execution Speed

The design methodology mentioned in Section III enables separately writing the code of the circuit solver and that of circuit components. This simplifies the entire code structure and enhances code maintainability. Adding a new component or a user-defined component becomes an easy task, and the circuit solver does not have to be modified for this. As a result, the program becomes expandable, in other words, flexible for adding new components. This has mainly been realized by the use of inheritance. However, the use of inheritance always involves inevitable overhead of execution time, although it is minimized by the one-time inheritance described in Section III-C. This must be kept in mind.

If the ultimate objective is to achieve the highest execution speed, the design strategy described in [11] that does not use inheritance should be recommended. According to this author’s view, however, execution speed is not always of the highest priority, and expandability is also important because the demand of adding user-defined models is increasing for design and test purposes. In addition, computers are becoming faster and faster and compilers of object-oriented languages

are continuously improved. Therefore, if expandability and code maintainability are considered to be key points of design, the proposed design methodology is preferable.

VI. Conclusion

This paper has presented a design methodology of an electromagnetic transient analysis program based on a fully object-oriented approach. The proposed design methodology uses an object-oriented approach for the modeling of circuit components, and this enables that the code of the circuit solver and that of circuit components can be written separately. Thus, coding is simplified and code maintainability is enhanced. A resultant transient analysis program becomes expandable, in other words, flexible for adding new components. Some implementation issues have also been discussed in this paper.

References

[1] A. F. Neyer, F. F. Wu, and K. Imhof, "Object-oriented programming for flexible software: Example of load flow," *IEEE Trans. on Power Systems*, vol. 5, no. 3, pp. 689–696, Aug. 1990.

[2] B. Hakavik and A. T. Holen, "Power system modeling and sparse matrix operations using object-oriented programming," *IEEE Trans. on Power Systems*, vol. 9, no. 2, pp. 1045–1051, May 1994.

[3] E. Z. Zhou, "Object-oriented programming, C++ and power system simulation," *IEEE Trans. on Power Systems*, vol. 11, no. 1, pp. 206–215, Feb. 1996.

[4] C. R. Fuerte-Esquivel, E. Acha, S. G. Tan, and J. J. Rico, "Efficient object oriented power systems software for the analysis of large-scale networks containing FACTS-controlled branches," *IEEE Trans. on Power Systems*, vol. 13, no. 2, pp. 464–472, May 1998.

[5] A. Manzoni, A. S. e Silva, and I. C. Decker, "Power system dynamics simulation using object-oriented programming," *IEEE Trans. on Power Systems*, vol. 14, no. 1, pp. 249–255, Feb. 1999.

[6] A. Medina, A. Ramos-Paz, R. Mora-Juarez, and C. R. Fuerte-Esquivel, "Object oriented programming techniques applied to conventional and fast time domain algorithms for the steady state solution of nonlinear electric networks," *Proc. 2004 IEEE Power Engineering Society General Meeting*, pp. 342–346, June 6–10, 2004.

[7] J. Zhu and D. L. Lubkeman, "Object-oriented development of software systems for power system simulations," *IEEE Trans. on Power Systems*, vol. 12, no. 2, pp. 1002–1007, May 1997.

[8] D. G. Flinn and R. C. Dugan, "A database for diverse power system simulation applications," *IEEE Trans. on Power Systems*, vol. 7, no. 2, pp. 784–790, May 1992.

[9] M. Foley, W. Mitchell, and A. Faustini, "An object based graphical user interface for power systems," *IEEE Trans. on Power Systems*, vol. 8, no. 1, pp. 97–104, Feb. 1993.

[10] M. Kezunovic and Y. Liao, "Development of a fault locating system using object-oriented programming," *Proc. 2001 IEEE Power Engineering Society Winter Meeting*, pp. 763–768, Jan. 28 – Feb. 1, 2001.

[11] J. Mahseredjian, B. Bressac, A. Xemard, L. Gerin-Lajoie, and P.-J. Lagace, "A Fortran-95 implementation of EMTP algorithms," *Proc. Int. Conf. on Power Systems Transients (IPST) 2001*, pp. 686–691, Rio de Janeiro, Brazil, 2001.

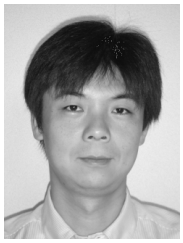
[12] S. Pandit and S. A. Khaparde, "Object-oriented design for power system applications," *IEEE Computer Applications in Power*, Oct. 2000, pp. 43–47.

[13] M. P. Selvan and K. S. Swarup, "Object methodology," *IEEE Power & Energy Magazine*, Jan./Feb. 2005, pp. 18–29.

[14] H. W. Dommel, "Digital computer solution of electromagnetic transients in single- and multi-phase networks," *IEEE Trans. Power Apparatus and Systems*, vol. PAS-88, no. 4, pp. 388–399, Apr. 1969.

[15] C.-W. Ho, A. E. Ruehli, and P. A. Brennan, "The modified nodal approach to network analysis," *IEEE Trans. Circuit and Systems*, vol. CAS-22, no. 6, pp. 504-509, Jun. 1975.

[16] http://www.computational.unibas.ch/cs/scicomp/software/pardiso/



Taku Noda (M'97) was born in Osaka, Japan in 1969. He received his Bachelor's, Master's, and Doctoral degrees all in engineering from Doshisha University, Kyoto, Japan in 1992, 1994, and 1997 respectively. In 1997 he joined Central Research Institute of Electric Power Industry (CRIEPI). He is currently a Research Scientist there. From 2001 to 2002 he was a Visiting Scientist at the University of Toronto, Toronto, Ontario, Canada. From 2005 he has been serving also as an Adjunct Professor at Doshisha University.

His research interest includes electromagnetic transient analysis of power systems and analysis of surges using electromagnetic field computations.