# Raiden. Power System Transient Stability Simulation Software Prototype

Eugene Mashalov, Joint Stock Company "Scientific and Technical Center of Unified Power System", Ekaterinburg, Russia

***Abstract*—This paper discusses implementation details of the transient stability analysis prototype software based on the implicit integration scheme. The main advantages of an implicit integration scheme with separate treatment of differential and algebraic state variables as well as an integration process with discrete event handling are considered. The results of test simulations and their comparison with existing software packages are presented.**

***Index Terms*—Transient Stability Analysis, Implicit Integration, Discontinuous DAE**

## I. Introduction

TRANSIENT stability simulation is one of the most complicated tasks of power system analysis. Besides methodological complexity, it requires intense computation with significant time consumption. Parallel implementation of simulation algorithms is difficult due to the sequential nature of the solution method, which is based on integration of the differential-algebraic system of equations (DAE), which is hardly separable into weakly connected parts. Thus, even modern computer systems do not provide the opportunity for real-time simulation, unless model details are reduced. Transient stability is dominantly used in offline applications, such as facility development, and rarely used for power system control on a time scale close to real-time.

However, some near-real-time implementations have been introduced in the last decade. One of the most successful is the stability monitoring system, used by the System Operator of Russia. The system evaluates the safe limits of network load over a period of 40–900 s with respect to static and transient stability. However, some near-real-time implementations have been introduced in the last decade. One of the most successful is the stability monitoring system, used by the System Operator of Russia. The system evaluates the safe margins of network load over a period of 40–900 s with respect to static and transient stability. These margins are used for system automatic or manual control. The duration of the evaluation cycle is mostly determined by the duration of the transient stability simulation. The aim of reducing the computation time of simulation led to the development of the software prototype based on the implicit integration scheme, instead of the former implementation based on explicit methods. The new algorithm consists of three components of integration: the method, the scheme, and the process. However, some near-real-time implementations have been introduced in the last decade. One of the most successful is the stability monitoring system, used by the System Operator of Russia. The system evaluates the safe margins of network load over a period of 40–900 s with respect to static and transient stability. These margins are used for system automatic or manual control. The duration of the evaluation cycle is mostly determined by the duration of the transient stability simulation. The aim of reducing the computation time of simulation led to the development of the software prototype based on the implicit integration scheme, instead of the former implementation based on explicit methods. The new algorithm consists of three components of integration: the method, the scheme, and the process. The integration method defines the numerical properties of the problem solution. The integration scheme defines the computation process of the method to ensure optimal utilization of the problem properties. And finally, the integration process arranges the operation of the integration scheme, considering discrete events and discontinuities in the model.

## II. Transient stability simulation algorithm development

Time spent on transient simulation is proportional to the number of numerical integration steps of the system (1):

$$\begin{cases} \dot{y} &= f(x(t), y(t), t) \\ 0 &= g(x(t), t(t), t) \end{cases} \qquad (1)$$
$$x(t_0) = x_0, y(t_0) = y_0$$

where $g$ and $f$ are smooth vector functions. The time of simulation initialization and computation of the initial conditions is much smaller than the time of the (1) solution. The latter depends on the integration method properties and the system dimension $N_e$. As a rule, solution time $T_{hn} = O(N_e^3)$, since the integration step in one form or another requires the solution of a system of nonlinear algebraic equations. The value of the integration step $h_n$ is determined by the properties of the DAE. The crucial property of (1) with respect to $h_n$ is the stiffness of the DAE, estimated as the ratio of the maximum and minimum values of the real part of the eigenvalues of its linearized matrix.

### A. The integration method

In most cases, the transient stability problem (1) is considered stiff, which encourages using implicit integration methods for its solution. The integration method is a vector function $F(x, y)$ which maps values from the previous step $y_{n-1}$ to current step values $y_n$. Of course, the current step values of

$x_n \leftarrow x_{n-1}$ are also required, but there are several approaches to performing this. The general form of the implicit method for differential variables is:

$$F(y_n, y_{n-1}, t) = 0 \tag{2}$$

The solution of (3) involves the solution of the nonlinear system of equations, which is associated with some technical complications. That was the main reason for the former implementation of transient stability software, which was developed in the 1980s relying on explicit methods. The solution of (2) implies the solution of the nonlinear system of equations, which is associated with some technical complications. Integration step control bounds solution to:

$$\begin{gathered} ||z(t_n) - z_n(t_n)|| <= \epsilon, \\ z_n(t_n) = [y_n(t_n), x_n(t_n)]^T \\ z(t) = [y(t), x(t)]^T \end{gathered} \tag{3}$$

where

$z_n(t_n)$ is the approximated solution;
$z(t)$ is the exact solution;
$\epsilon$ is the error tolerance;

The step control allows one to vary $h_n = t_n - t_{n-1}$ according to (3). For transient phases with rapid changes, the step is reduced. When the transient decays, the step can be increased, which significantly reduces the number of steps required to solve (1).

The implicit method implementation is much more complicated than the explicit one, as it requires a non-linear system of equations solution. In most cases, this solution is provided by the Newton method. Power system equipment models should be implemented with respect to this requirement and provide the ability to construct blocks of partial derivatives of the Jacobi matrix. The use of the automatic implementation system can overcome this problem [1]. Another problem that arises from nonlinearities is the possibility of multiple solutions at the time instants of discontinuities. However, the problems where (1) has discontinuous $f$ and $g$ are well known and there are various approaches developed. Some use rigorous discontinuity modeling [2], while others use integration restart techniques [3].

### *B. The integration scheme*

The DAE (1) solution implies discretization into a pure algebraic system of equations using a chosen integration method. Since the resulting system is nonlinear, an iterative solution method is used. The simplest method is fixed point iteration. This method is usable, but, in many cases, does not provide convergence due to its limitations. The application of the Newton method requires Jacobian matrix formation, which greatly increases the equipment models development complexity. In addition to the system of equations for each model, it is also required to form a block of the Jacobi matrix. However, Newton's method makes it possible to reliably obtain a solution, has good convergence, and, moreover, can be used not only to solve (1) but also to determine the initial conditions at $t = 0$ and at the time points of discrete changes $t_d$. The weak side of Newton's method is its sensitivity to the quality of the initial guess. This problem is partly solved by using the initial estimation from the previous integration step. An additional factor that improves the convergence of Newton's method can be the scheme of the integration method. A well-known integration approach is the predictor-corrector scheme. With values on the available solution $z_{n-1}$ at the time $t_{n-1}$, the method can predict the values of $z_n$ using extrapolation. The resulting prediction is used as an initial estimation for solving the equivalent (1) system, in which the differential equations are discretized in a form that depends on the chosen integration method. During the solution, the prediction is corrected so that the values satisfy the given system of equations. This scheme is effectively implemented using multi-step integration methods, which imply saving the data of several completed steps to perform the next step. To solve DAE, this approach is implemented in the Gear method [4]. This method involves the use of the BDF integration method, both for differential and algebraic equations, and provides a simultaneous solution to (1). Such an integration scheme is fully implicit and preserves integration method stability properties.

A variant of the Gear method is implemented in the well-known software Eurostag [5]. Since the methods of the BDF family have the property of damping high-frequency oscillations with an increase in the integration step, the Gear method exhibits "hyperstability" and erroneously treats unstable cases as stable. The proposed modification, which consists of applying the Adams method for differential variables, eliminates this drawback. A variant of the Gear method is implemented in the well-known software Eurostag [8]. Since the methods of the BDF family have the property of damping high-frequency oscillations with an increase in the integration step, the Gear method exhibits hyperstability and erroneously treats unstable cases as stable. The proposed modification, which consists of applying the Adams method to differential variables, eliminates this drawback since the 2nd order Adams method is A-stable. The use of the BDF for algebraic variables makes it possible to maintain a higher integration step compared to the Adams method with the same overall stability properties of the integration scheme.

All linear multistep methods of order $q$, despite significant differences in their properties, can be expressed in Nordsieck form [6]. That vector has a dimension of $1 \times q + 1$. Since the integration method must be A-stable order is constrained $(q <= 2)$ and the maximum dimension of the Nordsieck vector is fixed at $1 \times 3$.

At the n-th integration step, Nordsieck vector components of differential and algebraic variables are:

$$Y_n = \left[y_n, h_n\dot{y}_n, \tfrac{h_n^2}{2}\ddot{y}\right], \quad X_n = \left[x_n, h_n\dot{x}_n, \tfrac{h_n^2}{2}\ddot{x}\right] \tag{4}$$

The Nordsieck vector is merely a form to express the Taylor's expansion of $x_n$ and $y_n$ at $t_n$. If this vector for $t_{n-1}$ is known, the prediction at $t_n$ is:

$$Y_n^0 = Y_{n-1}A, \quad X_n^0 = X_{n-1}A \tag{5}$$

where

$Y_n^0$ is the differential variables vector extrapolated to $t_n$;

$X_n^0$ is the algebraic variables vector extrapolated to $t_n$;
$A$ is the lower triangular Pascal matrix $q+1 \times q+1$.

The corrected Nordsieck vector at $t_n$ is:

$$Y_n = Y_n^0 + l^Y e_{yn}, \quad X_n = X_n^0 + l^X e_{xn}, \tag{6}$$

where

$l^Y$ is the row vector of integration method for differential variables;
$l^X$ is the row vector of integration method for algebraic variables;
$e_{yn}, e_{xn}$ are the error vectors with respect to the predicted vectors;

The vectors $e_{yn}, e_{xn}$ are normalized:

$$l_1^Y = l_1^X = 1 \tag{7}$$

The corrector equations (6) can be written by components:

$$\begin{array}{rcccl} y_n & = & y_n^0 & + & l_0^Y e_{yn} \\ h_n\dot{y}_n & = & h_n\dot{y}_n^0 & + & l_1^Y e_{yn} \\ \frac{h_n^2}{2}\ddot{y}_n & = & \frac{h_n^2}{2}\ddot{y}_n^0 & + & l_2^Y e_{yn} \\ x_n & = & x_n^0 & + & l_0^X e_{xn} \\ h_n\dot{x}_n & = & h_n\dot{x}_n^0 & + & l_1^X e_{xn} \\ \frac{h_n^2}{2}\ddot{x}_n & = & \frac{h_n^2}{2}\ddot{x}_n^0 & + & l_2^X e_{xn} \end{array} \tag{8}$$

From the $\dot{y}_n$ equation from (8):

$$\begin{array}{l} h_n\dot{y}_n = h_n f(x_n, y_n, t_n) \\ h_n\dot{y}_n^0 + l_1^Y e_{yn} - h_n f(x_n, y_n, t_n) = 0 \end{array} \tag{9}$$

Using (7) define:

$$\begin{array}{c} F_Y(e_{xn}, e_{yn}, t) = h_n\dot{y}_n^0 + e_{yn} - h_n f(x_n, y_n, t_n) \\ F_X(e_{xn}, e_{yn}, t) = g(x_n, y_n^0 + l_0^Y e_{yn}, t_n) \end{array} \tag{10}$$

from which the following system can be obtained:

$$\left\{ \begin{array}{rcl} F_Y(e_{xn}, e_{yn}, t) & = & 0 \\ F_X(e_{xn}, e_{yn}, t) & = & 0 \end{array} \right. \tag{11}$$

with Jacobi matrix ($n$ step index is omitted for simplicity):

$$J = \begin{bmatrix} \frac{\partial F_Y(e_x, e_y, t)}{\partial e_y} + \frac{\partial F_Y(e_x, e_y, t)}{\partial e_x} \\ \frac{\partial F_X(e_x, e_y, t)}{\partial e_y} + \frac{\partial F_X(e_x, e_y, t)}{\partial e_x} \end{bmatrix} \tag{12}$$

The combination of (10) and (12) gives:

$$J = \begin{bmatrix} I - h_n l_0^Y \frac{\partial f(x,y,t)}{\partial y} - h_n l_0^X \frac{\partial f(x,y,t)}{\partial x} \\ l_0^Y \frac{\partial g(x,y,t)}{\partial y} + l_0^X \frac{\partial g(x,y,t)}{\partial x} \end{bmatrix} \tag{13}$$

Note that the Jacobi matrix does not become singular with $h_n \to 0$, which allows it to be used to solve DAE for initial conditions $y_d$, $x_d$ at instants of discontinuities $t_d$. In addition, step reduction does not cause numerical problems, which, for example, the integration scheme [7] is subject to.

The system of equations (11) can be solved iteratively. Variables at iteration $m$ are:

$$\left\{ \begin{array}{c} y_n^m = y_n^0 + l_0^Y e_{yn}^m \\ x_n^m = x_n^0 + l_0^X e_{xn}^m \\ e_{xn}^0 = e_{yn}^0 = 0 \end{array} \right. \tag{14}$$

The iterative process of solving (11) with respect to $e_x$ and $e_y$ is performed according to the recursive formula ($n$ step index is omitted for simplicity):

$$\begin{bmatrix} e_y^{m+1} \\ e_x^{m+1} \end{bmatrix} + \begin{bmatrix} e_y^m \\ e_x^m \end{bmatrix} + (J^m)^{-1} + B^m \tag{15}$$

where:

$$B^m = \begin{bmatrix} h_n f(x^m, y^m, t) - h\dot{y}^0 - e_y^m \\ -g(x^m, y^m, t_n) \end{bmatrix} \tag{16}$$

and:

$$J^m = \begin{bmatrix} I - h_n l_0^Y \frac{\partial f(x^m, y^m, t)}{\partial y} - h_n l_0^X \frac{\partial f(x^m, y^m, t)}{\partial x} \\ l_0^Y \frac{\partial g(x^m, y^m, t)}{\partial y} + l_0^X \frac{\partial g(x^m, y^m, t)}{\partial x} \end{bmatrix} \tag{17}$$

When the process (15) converges, the Nordsiek vectors $X_n$ and $Y_n$ at the $t_n$ are updated using (6).

### C. Local truncation error control

The solution of (1) is formally to find $x(t)$ and $y(t)$ for $t \in [t_0; T_{end}]$. Since the analytical solution of (1) is impossible in most cases, numerical integration is used, which involves replacing differential part of (1) with finite-difference equations and their sequential solution with some integration method. The integration method builds a sequence of approximations satisfying the conditions:

$$||z(t_n) - z_n(t_n)|| \leq \epsilon \tag{18}$$

where $z(t_n)$ is the analytical solution and $z_n(t_n)$ is approximated solution. Since (1) is usually stiff, mostly implicit integration methods are used, requiring the solution of a nonlinear system of equations at each integration step $n$.

Transient trajectories are not smooth and subject to discontinuities at particular time instants due to switching and limiting of state variables. Thus, $f$ and $g$ are not continuous in $t \in [t_0; T_{end}]$ but only piece-wise continuous, as well as their derivatives. This requires a restart integration method at time instants of discontinuities $t_d$ with new initial conditions $z_d(t_d)$. In some cases structure and dimension of (1) also subject to change. Discontinuities are associated with events that can be divided into time events and state events [**?**]. Time events are unconditional and have a known time of occurrence. They can be applied to the solution at scheduled time instants. State events have only conditions of occurrence and their times must be located during the solution.

Based on the above characteristics of the problem, to solve system (1), the following minimum set of procedures should be implemented:

- evaluation of initial conditions at $t_0$;
- evaluation of residuals of equations;
- construction of a submatrix of partial derivatives;
- time location of the state events $t_d$;

- evaluation of switched (1) initial conditions at the discontinuities $t_d$.

The custom model is a subsystem of (1), therefore, a set of procedures identical to the set for the general system must be implemented for it.

When a developer is asked for the creation of equipment model, he starts by analyzing the given block diagram, forms a system of equations, and then implements a program with functions that ensure the interaction of the model with the software core. The result of that work is an integral part of the software. To implement a custom model, a similar process is required with two differences: the developer does not participate in the process, and the result of the work is not included directly in the software, but operates as an external module. The latter is usually, a native machine code executable module in the operating system format. (*.dll for Windows or *.so for Unix systems). Implementing custom model functions in native code eliminates technical differences between custom and built-in models and ensures maximum performance.

The generation of native code in itself is the complicated task, since it is necessary to consider the system architecture features. Therefore, when implementing custom models, to generate native code, standard systems for compiling executable modules are used, for which the source code of the custom model program is preliminarily created in one of the general-purpose programming languages. In this case, such a programming language is called an intermediate. In practice, C/C++ and Fortran compilers are often used. Automatically generated programs in these languages are the penultimate stage of the implementation of the custom model, preceding the compilation of the executable module in native code.

## III. Model representation in AST-form

Consider the model representation in block diagram and textual forms. As an example, a simple model of the automatic voltage regulator is used. Textual form of representation is shown in Fig. 1 and the block diagram in Fig. 2.

Fig. 1. Sample AVR model pseudocode representation

Fig. 2. Sample AVR model block diagram

Both representations use elementary functional blocks: lag, derivative lag and limited lag. Ready-made elementary functional blocks are included in custom model toolbox to simplify model creation and enable efficient implementation by software core instead of letting the user build these blocks from individual differential equations. In addition, the implementation of the elementary blocks in the software core allows to use common procedures for discrete events location and processing and hide these details from the user. If necessary, differential equations can be explicitly specified using the integrator block.

Formally, the block diagram of the model corresponds to a directed graph whose vertices are elementary blocks, and edges are defined by links. Compared to a conventional graph, the order in which the edges in a node are listed matters, because elementary blocks have positional arguments that must be specified in a particular order. This imposes some restrictions on the algorithms used in the implementation of custom models. The block diagram representation of custom models turns out to be easier to use for implementation compared to textual one, since the order of calculations is readily available by traversing the graph from outputs to inputs, and validation of links and parameters is performed by the block diagram editor.

Creating a custom model in a text representation is reduced to describing a system of equations using generally accepted mathematical operators and a set of built-in functions that also include elementary blocks. The system of equations must link the input and output variables. This provides the possibility of using internal variables for structuring the system or introducing feedback. The description of the system is far from an intermediate language, so its transformation is required to implement the model. The textual representation is not as visual as the block diagram, but with some skill, it greatly speeds up the creation of models. At least text editors today remain the basic tool of professional software development and there is no trend toward the transition to graphical environments. It should be emphasized that the description of the model in the textual representation cannot be called a program in the traditional sense of the term. The program sets the sequence of actions, while the solution sequence is not defined for the system of equations in the textual representation. However, the implementation of the model must be nothing more than an ordinary imperative program in the form of an executable module, therefore, in the process of transformations, the order of calculations must be determined.

To transform the custom model source data in text format, some general-purpose parsers are widely used. They are the programs that allow to perform lexical and syntactic analysis of the source text according to a given dictionary and form the required data structure for further processing [**?**]. Almost all programming languages use parsers in one way or another to generate intermediate object code from source code. Existing parsing technologies can also be used to analyze the textual representation of the model's system of equations. In the process of analyzing the original textual representation, the verification of compliance with the specified rules and the detection of syntactic errors are performed. As a result of the analysis, a tree structure is generated that allows to perform the required transformations for further processing by manipulating the tree. To represent the system of equations of the custom model, a structure in the form of the abstract syntax tree (AST) is suitable. The nodes of such a tree are the operators, and the edges define the mutual directed links of the operators. One of the possible options for converting the textual representation of the AVR model from the example Fig. 2, 1 into the AST is shown in Fig 3.

Fig. 3. Sample AVR model AST-tree

Generally, the resulting structure is not a tree. The use of

variables as operands of elementary blocks turns the structure into a graph, which, in the presence of feedback, cannot be reduced to the form of a tree. If there are several output variables, the structure turns into a forest. The traditional use of the term "tree" because the structure is hierarchical and determines the order of links of nodes. For the AST shown in the Fig 3 cycles on system variables are shown by dotted lines. If it were possible to eliminate the edges of the connections of variables, the graph could be considered directed from below nodes to the top. The edges of the variables connections do not have a given directions, but they can be determined if there is no feedback.

The form of AST for a given system of equations is not unique. For ASTs, there is an identity rule, similar to the rule for mathematical expressions. The two ASTs are identical if the values of the root nodes are identical on the entire set of values of the variables included in them in their domains. While maintaining identity, it is possible to perform AST transformations using subtree manipulations, achieving certain properties from the resulting AST. All manipulations of this kind are performed at the symbol level, that is, with variables whose values are unknown. Identity is preserved using the rules of algebra and logic in the transformations. The AST structure is convenient for these transformations from an algorithmic point of view.

Taking as intermediate variables $V_c$, $V_s$ and $U_{sum}$ at the points marked in the block diagram of the model in Fig. 2, by traversing the graph from outputs to inputs, it is easy to obtain an AST identical to Fig. 3. One can also get the AST identical to that shown in Fig. 3 by transforming an arbitrary variation of the AST built from block diagram. Thus, the AST can be used as an intermediate structure that is invariant with respect to the representation of the custom model. To build it, the analysis of the model graph and the analysis of text expressions can be combined. Due to this, the possibilities of describing the custom model in block diagram form can be significantly expanded. In particular, for the inputs and outputs of elementary blocks, mathematical expressions can be additionally specified in the textual representation. They can be used for additional value transformations or for linking to other model blocks without having to explicitly draw those links. At the same time, the block diagram of the model remains compact and clear, and the user has almost unlimited possibilities to expand the functionality of the model. The construction of an AST with such a combined approach begins with a traversal of the model block diagram graph, with a transition to the analysis of mathematical expressions. The results of parsing textual expressions are included in the graph AST as subtrees.

After transformation the representation of the model into the form of the AST, the main stage of the model implementation begins, which includes the following operations:

- classification of variables;
- validation of the initial system of equations;
- simplification and optimization of mathematical expressions;
- generation of expressions for calculating partial derivatives;
- determination of the order of the system of equations solution the by variables causalization.

## IV. Custom model symbolic transformation

### A. Model variables classification

The block diagram or system of equations is the main, but not the only component of the source data for the implementation of the custom model. To link the model with the software core, additional information is required, which, in particular, includes the declaration of variables. With the help of named variables, data exchange is arranged between the model and the software core, as well as with other models that are part of the system (1). In addition to names, the declarations of variables include attributes that allow them to be divided into classes according to certain properties. The one of the most important is the constant attribute. Variables with this attribute cannot be changed during the simulation and are used to obtain source data when the model is initialized and the initial conditions are calculated. The attribute is also necessary for the correct construction of the system of equations and to determine the order of its solution. In the AST example above, the constants are variables for gains, time constants, and limits.

Constant attributes can be explicitly declared for variables in the model source data. But during the analysis of the AST, some variables can receive the constant attribute implicitly. For example, an AST subtree in which only numerical values and constants act as variables is constant. If such an AST subtree resolves a state variable, then it will also be a constant. The definition of an implicit constant attribute affects the structure of the system of equations of the model. Obviously, the number of equations must be equal to the number of unknown variables. Classification of one of the state variables as a constant requires eliminating one of the equations from the system. If the equality of the number of variables and equations is preserved, such elimination reduces the computation burden. However, in some cases, implicitly defining a variable as a constant may result in multiple equations being resolved, in which case, the system becomes underdetermined. In such a situation, the model implementation process should terminate with an appropriate error message pointing to the source data problem.

Variables that are not classified as constants are state variables and, in turn, are divided into internal and external. Internal variables are resolved by model equations. External variables are references to internal variables of other models. In the example above, external to the AST model are the generator state variables $S_u$, $V_g$, $I_g$, and $I_f$ – rotor slip, terminal voltage, stator and rotor currents, respectively. In turn, the internal variable $U_f$ will be interpreted as external in the exciter model.

For state variables that are not constants, there is an intermediate class of discrete variables. It includes variables whose values can change only when the system of equations undergoes discontinuity. These include, for example, variable output values of logical elementary blocks. When the state of such a block changes, the software core must process a discontinuity of the system of equations by restarting the integration method with the calculation of new initial conditions,

including new values of discrete variables. In the time interval between discontinuities, the discrete variable has a constant value. Thus, it makes no sense to include discrete variables into the system of model equations, considering their values in the process of integration as constants. Calculation of the values of discrete variables is performed only when initialization and in the discontinuity processing. The classification of discrete variables is performed according to a principle similar to the implicit classification of constants: if the AST subtree resolves some variable and contains only constants and discrete variables, then the resolved variable is discrete.

Fig. 4. Classification of the custom model variables

State variables not classified as discrete or constant are ordinary continuous variables and are resolved by the system of model equations as part of system (1). The Fig. 4 shows a simplified variables classification chart, indicating how variables can be assigned to classes. It should be noted that the variables are not divided into differential and algebraic, because the integration process in a particular implementation is based on the Gear method [**?**].

### *B. Equation system source data validation*

In the process of automatic custom model implementation, two phases of validation of the source data can be distinguished. The first phase is to detect errors that prevent the correct generation of the AST based on graphical or textual representations. In the graphical representation of the source data, a set of elementary blocks corresponding to the AST nodes is defined, therefore, only topology validation is necessary: checks of unconnected inputs and outputs, the direction of connections from outputs to inputs, etc. To prepare a text representation for generating an AST, it must be first parsed from the text and then checked or syntax errors, most of which prevent correct connection generation between AST nodes and in some cases make it impossible to determine the set of AST nodes.

Once the AST has been generated, a second phase of validation can be performed, which is to detect semantic errors. The second phase does not depend on the way the model is presented initially since it uses only the AST and the results of the variable classification. This information allows validation of AST nodes that have received the constant attribute. In particular, the AST allows to ensure that those elementary block arguments that must be given as constants are defined by a constant subtrees. Since there are no restrictions on the structure of equations in the text representation, it is possible for an equation to explicitly or implicitly change the value of a variable with a constant attribute. Detection of such errors can also be performed by the AST.

If the values of the child nodes of some AST node can be calculated from the given constants, it becomes possible to check the result of calculation for compliance with the acceptable node values in the process of model implementation. Such nodes include, for example, division, square root, inverse trigonometric functions, elementary blocks with constraints. Error analysis in constant expressions, of course, does not guarantee that the model will behave correctly for any set of values of the source data, but allows to identify errors that do not depend on the source data and will inevitably arise during the simulation.

### *C. Simplification and optimization of mathematical expressions*

The possibility of representing mathematical expressions in a tree structure is widely used in computer algebra systems [**?**]. The purposes of such systems go far beyond the simplification of mathematical expressions and is mainly focused on solving problems symbolically, up to automated proof of theorems. Parts of the technology of identical transformations of expressions are in demand in optimization systems for translating program code, which can also include the considered system for model implementation. Expression simplification in computer algebra systems acts as a utility function and is designed to display results in a compact, generally accepted form. There is no universal definition of the result of the expression simplification, so at least two options are considered: the factored form (for example, $(a + b)(a - b)$ or the polynomial form $(a^2 - b^2)$. In the context of the considered application for the implementation of custom models, the term "simplification" takes on a clearer meaning and implies the calculation of the expression with minimum computation cost. If each AST node is assigned a score of the cost of performing an operation, the total cost of resolving the AST can be determined. The task of simplification is reduced to finding an identical AST with minimal total computation cost. Considering that the costs of the operation of multiplication, and even more so of raising to a power, are higher than for the operation of addition, for the difference of squares considered in the example above, the result of simplification will be $(a + b)(a - b)$.

Identity transformations of the AST are performed according to a certain set of rules $r_i$ forming the set $R$. The rule $r_i$ is an identity, on the left side of which the AST in the original form, and on the right - the transformed AST. For example, the rule for the difference of squares has the form shown in Fig 5.

Fig. 5. The difference of squares identity tree transformation rule

Since the rules that define identities are reversible, they can be applied for transformations from the left AST to the right one and vice versa. To apply a rule, it is necessary to find in the source AST a subtree corresponding to the pattern of the AST of the rule and replace it with the AST subtree from the opposite part of the rule. At the same time, both when searching and when replacing, it must be taken into account that any subtrees of the original AST can act as arguments to the rules. In this example, these subtrees are labeled $a$ and $b$. Thus, rules can be applied not only to expression variables, but to subexpressions.

Searching for a rule subtree pattern in the AST can be done recursively from top to bottom or bottom to top. For large ASTs, bottom-up search is more efficient, but it requires tree

preprocessing. Since the AST changes during the application of the rules, the processing must be repeated, therefore, the search for patterns is performed from top to bottom. When searching, one must consider the permutations of arguments in nested AST nodes. For example, an addition operation with three terms yields 6 permutations, each of which must be checked against the pattern. In addition, some operations in AST are equivalent, such as exponentiation and successive multiplication. To reduce searching complexity for pattern rules, the AST is maintained in the so-called canonical form which is described by the following rules:

1) All commutative binary operators are replaced by n-ary ones. In this case, the subtree of consecutive binary operators of the same type is replaced by one operator node with $n$ child nodes.
2) Subtraction nodes are replaced by addition nodes with a unary minus.
3) Division nodes are replaced by multiplication nodes. The divisor expression is enclosed in the exponentiation $-1$.
4) The child nodes are sorted in a lexical order.

Rules 1, 2, and 3 eliminate the operations of subtraction and division and reduce the height of the AST, which simplifies the comparison of subtrees and reduces the required search depth. Rule 4 allows to eliminate the analysis of permutations when searching for rule patterns since all child nodes will be in a defined order. Before generating the implementation of the model in the intermediate language, rules 2 and 3 can be canceled out by converting to the regular operations according to additional rules that are introduced into $R$ at the final stage of system simplification.

To optimize the system of equations for performance, the set of rules $R$ includes elementary of algebra (commutative, associative and distributive laws), rules of addition with zero and multiplication with zero and one, identities with powers, trigonometric and logarithms. Since implicitly specified constants can be determined during the simplification process, the rules also include the calculation of AST nodes by numerical arguments. If a numerical value of a node can be calculated, it is excluded from the AST together with its subtree and replaced with a numerical value. The total number of basic rules is about fifty. All rules are represented in a special form of the AST, extended for patterns representation.

A feature of the simplification of the expressions used in this work is that it is performed for the entire system of equations, and not for individual equations. The simplification algorithm searches for repeated subexpressions, extracts them into a separate equations, and replaces the found expressions with a variable using a substitution rule. Equation representation in canonical form $f(x) = 0$ eliminates the use of rules that must work on both sides of the equation, such as taking logarithms or exponentiation. In addition, this form of equation representation simplifies the generation of expressions for partial derivatives and the determination of the order in which equations are solved.

Simplification and optimization of expressions in a system of equations consist of the sequential application of rules with the tracking of computational cost. The problem in what order and which rules to apply to achieve a minimum of cost does not have an unambiguous solution and in most cases, is solved by the enumeration of possible variants. The solution of this problem is complicated by the fact that the sequence leading to the form of the system with minimal computational costs includes intermediate forms with costs higher than the original form. Therefore, when solving, it is impossible to use objective function indicating the optimal sequence. In addition, as in most problems of finding a global minimum, the solution can be achieved by exhaustive enumeration of all possible combinations, which cannot be applied in practice even for small systems and a reduced set of rules. For the problem of simplification and optimization of the system of equations, two approaches were tested in this work. Both do not guarantee the achievement of a global minimum of costs, but can significantly reduce costs (if possible) for an acceptable time of processing.

The first approach implies the separation of $R$ into two subsets: $R_1$ and $R_2$. The subset $R_1$ includes the rules that generate the polynomial form of expressions, and $R_2$ includes the rules for the factorized form. $R_1$ and $R_2$ are sequentially applied to the original system of equations until the computational cost stops decreasing. This approach is similar to that used for manual simplification of expressions. First, all the expressions are expanded by opening brackets, then they are factorized. This approach reduces computational costs, mainly due to the fact that in the process of sequential application of $R_1$ and $R_2$, some of the AST nodes are converted into constants or become common factors.

The second approach is more general and is based on finding a path in the graph $G(F, T)$, where $F$ are identical forms of the original AST to be simplified, $T$ are transformations performed according to the rules $R$. $A^*$ search algorithm [**?**] is used for path search. The original algorithm is designed to find the shortest path between given nodes in a weighted graph and is used, for example, to build routes on flat maps. The difference from the equation system simplification problem is that the target node is not specified in the source data and must be determined by the minimum cost. A distinctive feature of $A^*$ is that it preserves all paths from the source node to the destination. This allows to track changes in costs and cut off paths where costs do not decrease. However, the time spent on its execution will be exponential. Ways to reduce searching are heuristic pruning of ”dead end” paths and limiting the depth of paths to be searched

Practical studies show that, in most cases, the first approach allows to obtain an acceptable result. The second approach is used optionally since it is more complicated and not well determined, but in some cases it can significantly reduce computational costs compared with the original system because it can “overcome” a sharp increase in computational costs at the initial stages of system simplification and determine the optimal order of rules application.

It should be noted that the purpose of simplification of the expressions of the custom model is not only to reduce computational costs, but also to additionally check the consistency of the given system. During the simplification, a situation may arise in which dependent equations are eliminated, but the number of variables remains unchanged. Therefore, in

the process of simplification, the consistency of the system is verified according to the rules given in subsection IV-B.

### D. Automatic differentiation

At this stage of the implementation of the custom model, the system of equations is reduced to the final version with the minimal achieved computational costs and prepared for linking with the software core. To solve the system of equations of the model, it is necessary to build a block of the matrix of partial derivatives. For models implemented by a software developer, this stage is performed manually according to a given system of equations. To implement a custom model, the generation of symbolic expressions for calculating partial derivatives must be fully automatic.

Fig. 6. The power rule in form of AST

The problem of determining expressions for calculating partial derivatives is solved by symbolic manipulations using the AST of the system of equations. Differentiation is performed analytically and makes transient simulations free of of numerical differentiation errors The expressions for partial derivatives calculation are determined using template rules $d_i$, which form a set of differentiation rules $D$. The rules are defined for each type of AST node. As an example in Fig. 6 shows the rule for the partial derivative of the power function node.

If a differentiation rule is set for each type of AST node, then using a chain rule, an AST of a system of partial derivatives can be obtained, where each equation is assigned to an element of the Jacobi matrix of the custom model. Since the resulting expressions of partial derivatives will also be represented in the form of AST, the same simplification and optimization rules that were described in subsection IV-C can be applied to them. The practical use of symbolic differentiation of functions has shown that many expressions for partial derivatives turn out to be quite cumbersome and expressions can contain subexpressions that occur many times in the system. Simplification and optimization rules make it possible to detect such subexpressions and extract them into separate equations, which significantly reduces the computational costs of calculating partial derivatives.

Since the optimization of expressions of partial derivatives is somehow carried out by introducing additional equations for repeated subexpressions, a variant with a preliminary transformation of the model equations AST into the so-called operator tree was considered. In the operator tree, each original AST node is assigned an intermediate variable $s_i$:

$$s_i = op_i(s, z) \tag{19}$$

where:

$s$ is the intermediate variables vector;
$z$ state variables vector;
$op_i$ is the function defined by AST node;

There are no complicated functions in the operator tree, so only differentiation rules can be used to calculate partial derivatives with no need to use chain rule. The expressions obtained according to such rules are relatively compact and the computational costs for partial derivatives, most often, turn out to be lower compared to optimized expressions for complicated expressions. Of course, the reduction in computational costs is achieved by increasing the number of equations in the system. However, in this work, to solve the system of linear equations with a matrix of partial derivatives, the matrix is transformed into a block triangular form [**?**]. This form is constructed by the permutation of the system so that nonzero values are located above the diagonal, which in turn is free of zeros. After permutation, blocks remain on the diagonal, each of which, in the process of solving of the system, is subjected to LU decomposition with partial pivoting. The solution of the systems reduced to a back substitution in block form. Transformation of the AST into an operator form increases the dimension of the system being solved, but does not lead to the appearance of blocks on the diagonal, since all $s_i$ are resolvable with respect to $op_i$, and (19) can be interpreted not as an equality, but as an assignment. Thus, the addition of equations associated with the transformation of the AST into the operator form is equivalent to the introduction of intermediate variables in the calculation of partial derivatives, and its effect on the performance of the solution of the system is negligible.

The operator form of th AST also has a certain advantage, which allows for each operation in the system of equations of the custom model to obtain a result in the form of $s_i(t_n)$ in the process of simulation. This feature is useful for an in-depth analysis of the custom model during its development and debugging. It should also be noted that the automatic differentiation tool provides significant support in the process of model development in the traditional way – by the software developer, as it eliminates the error prone work of manual writing several tens or hundreds of expressions for partial derivatives.

### E. Determination of the order of the system of equations solution

There is no need to determine the order of solution for the system of equations of the custom model. However, the overall performance of the simulation was slightly affected by the pre-ordering of the system of equations into a block triangular form. This reduces the cost of complete system ordering (1) in the software core. The question of the order of solution arises in the problem of calculating of the initial conditions at the time instants $z_0(t_0)$ and $z_d(t_d)$. If there is no feedback in the model, this order is reduced to a sequence of assignments, usually in the direction from the output to the inputs. In the presence of feedback, the sequence requires the solution of subsystems of equations, in the general case, nonlinear ones. In most cases, the procedures for determining the initial conditions at $t_0$ and at $t_d$ are the same. The only difference is that at $t_0$, it is assumed:

$$\frac{df}{dt}(t_0) = \frac{dg}{dt}(t_0) = 0, \tag{20}$$

while for $t_d$ this condition is not met, but the values of the derivatives at $t_{d-}$ (up to the instant of discontinuity) are known. Differential variables in most cases are considered continuous:

$$\dot{y}(t_{d-}) = \dot{y}(t_{d+}), \tag{21}$$

which gives the ability to resolve $x(t_{d+})$.

The presence in the model of elements with integration constants, the values of which are determined using complicated logic, significantly hampers the calculation of the initial conditions. For example, the initialization of the PID controller or the implementation of bumpless control according to this law, depending on the given coefficients, may require the rejection of (21) and does not allow to determine the initial conditions in a general form [**?**]. In this case, it is required to supply extra source data of the custom model in the form of so-called initialization scheme, which explicitly declares expressions for determining the initial conditions.

In the simplest cases, when (1) does not contain nonlinear feedback and complicated logic, the initialization scheme can be constructed automatically by analyzing the original system. Since the AST in operator form (19) is used to solve this problem, a sufficient condition is the availability of the inverse function $op^{inv}$ to the operator $op$, which allows resolving the variable from the equation of the AST node. The ability to solve an equation with respect to some variable is determined by the classification of variables described in subsection IV-A and the set of already resolved variables. To determine the order of equations and variables resolving, an algorithm for finding the strongly connected components in the graph $G(V, E)$, where $V$ are state variables and $E$ are equations, can be applied. One of the most efficient is Tarjan's algorithm [**?**].

If there is feedback in the original system, then ordering the equations in the form of a simple sequence of calculations becomes impossible. Feedback form the so-called algebraic cycles, which handling requires the solution of a system of equations. Here the ordering algorithm chooses the variable that allows the maximum number of cycle equations to be resolved, and continues ordering the equations as if that variable was resolved. Such technique is known as tearing [**?**]. If there is an inverse function for the equation of such a variable, the variable is introduced into the system of cycle equations using a substitution. This technique allow the system of equations to be ordered so that it acquires a block triangular form with diagonal blocks of the minimum size. Thus, the initial system, during the implementation process takes on a form that does not require additional ordering in the software core.

In case the ordering algorithm copes with the task of resolving all variables, the custom model can be converted into an intermediate language program. Otherwise, the algorithm produces a list of variables and equations that could not be resolved, and the user must supply extra model data with the necessary equations. The main reason that not all state variables can be resolved is the lack of inverse functions for AST nodes or functions used for variable substitutions. In this case, the simplest way to resolve a variable is to explicitly specify an expression to calculate its value in the initial conditions.

Improving the capabilities of algorithms for determining the initial conditions is a subject of further research. The presence of algebraic cycles in itself is not a problem since they are resolvable by a system of equations. The main problems in solving (1) at instants $t_0$ and $t_d$ are related to the ambiguity in determining the integration constants and the states of logical elementary blocks. To obtain a unique and feasible solution of the system, additional options must be used. One of the possible options is to provide ready-made implementations of complicated logical elements as elementary blocks. A more versatile approach implies the addition of extra source data by extending the model representation. Possible extension options are the use of Petri nets or finite state machines.

## V. Conclusion

The process described above is implemented in a custom model compiler for the transient stability modeling package. The compiler accepts a model block diagram graph, textual description of the system of equations, or a mixed description consisting of graph with additional equations for elementary blocks. The compiler supports over 50 elementary blocks can generate source code in an intermediate language for subsequent translation into native code. C++17 is used as an intermediate language. The result of compilation is a dynamic library with the implementation of the functions necessary for the model to interact with the software core. By using the compiler not only a equipment model can be implemented, but also models of automatic devices and parameterized transient stability simulation scenarios.

The use of symbolic manipulations allows to fully automate the implementation of custom models for transient stability simulation. During the implementation of the model, symbolic manipulations allow to optimize models to improve performance and provide advanced error checking. The source data of custom models can be either graphic representation, textual representation, or a mix of both, which allows to combine the capabilities of different modeling approaches without the need to involve software developers.

## VI. References Section

You can use a bibliography generated by BibTeX as a .bbl file. [**?**] BibTeX documentation can be easily obtained at: http://mirror.ctan.org/biblio/bibtex/contrib/doc/ The IEEEtran BibTeX style support page is: http://www.michaelshell.org/tex/ieeetran/bibtex/

## VII. Simple References

You can manually copy in the resultant .bbl file and set second argument of `\begin` to the number of references

## REFERENCES


[1] E. Mashalov, "Automatic implementation of equipment models for transient stability simulation by symbolic manipulations," *Researchgate*, 2022. [Online]. Available: https://rgdoi.net/10.13140/RG.2.2.29110.78404

[2] A. F. Filippov, *Differential Equations with Discontinuous Righthand Sides*, F. M. Arscott, Ed. Springer Netherlands, 1988. [Online]. Available: https://doi.org/10.1007/978-94-015-7793-9

[3] F. E. Cellier and E. Kofman, *Continuous System Simulation*. Springer, New York, 2006.

[4] C. Gear, "Simultaneous numerical solution of differential-algebraic equations," *IEEE Transactions on Circuit Theory*, vol. 18, no. 1, pp. 89–95, 1971.

[5] J. Astic, A. Bihain, and M. Jerosolimski, "The mixed Adams-BDF variable step size algorithm to simulate transient and long term phenomena in power systems," *IEEE Transactions on Power Systems*, vol. 9, no. 2, pp. 929–935, 1994.

[6] E. Hairer, S. P. Nørsett, and G. Wanner, *Solving Ordinary Differential Equations I*. Springer Berlin, Heidelberg, 1993.

[7] P. Linda, "A Description of DASSL: A Differential/Algebraic System Solver, SAND82-8637," Sandia Labs, Tech. Rep., 1982.


## VIII. BIOGRAPHY SECTION


**Eugene Mashalov** Graduated from Electrotechnical Faculty, Ekaterinburg, Urals State Polytechnical University, 1997. Received Ph.D degree in 2000 from the same university. Works for JSC "Scientific and Technical Center of Unified Power System", Ekaterinburg, Russia.
mashalov@gmail.com
www.inorxl.com