Journal of Applied Sciences

ISSN 1812-5654

science alert

ANSInet
an open access publisher
http://ansinet.com

Journal of Applied Sciences 11 (15): 2723-2733, 2011
ISSN 1812-5654 / DOI: 10.3923/jas.2011.2723.2733
© 2011 Asian Network for Scientific Information

Object-oriented Programming Strategies for Numerical Solvers Applied to Continuous Simulation

Gustavo Boroni and Alejandro Clausse
CONICET and Universidad Nacional del Centro,7000 Tandil, Argentina

Abstract: The study of problems and inconveniences appearing in the Object-oriented (OO) implementation of continuous simulation systems based in Differential-algebraic Equations (DAE) is presented. It was found that the numerical treatment of the equations is crucial to find a design compatible with OO programming practices which lead naturally to implicit schemes. The family of Backward Differential Formulas (BDF) was found particularly appropriate to achieve high levels of software flexibility and reusability. A series of numerical studies were carried out comparing numerical performances with software quality metrics. It was found that BDF implementations improve substantially the software quality, although the computer costs also increase significatively which ultimately calls for pondering the importance of each software characteristic (i.e., modifiability-extensibility vs. calculation time). The equilibrium of this balance is determined by the size of the problem to solve. A utility function is proposed which can be used to determine the optimum choice.

Key words: Simulation, modeling, dynamic system, object oriented programming, computer applications

INTRODUCTION

Object Oriented Programming (OOP) supported by proper framework architectures provide features that might significantly improve the design, implementation and maintenance of large codes (Meyer, 2000; Yang, 2008). This methodology has reached many branches of applied computation, such as data bases, software agents, expert systems, computer graphics, communications support, among others (Adetunde, 2009).

Every computer code has two fundamental elements: data and functions. Data are the carriers of the digital information by means of a proper codification, whereas functions (which actually also should be represented through a codification) are procedures responsible for data modifications. In the procedural programming methodology the programs are divided in modules defined by the functions (subroutines, procedures, functions, etc.). In turn, OOP changes radically the principle of program structure: the data become the fundamental elements, whereas functions (called methods) are defined and implemented according to the modularization naturally suggested by the data. In this way, each program module (called object) is responsible for their own data which can only be modified by their own methods and can only be "observed" by the other objects through communication protocols. In addition OOP introduces other modularity concepts (e.g., abstraction, inheritance and polymorphism) that help the programmer in structuring the systems ensuring the future extensibility and modifiability.

Most of the software tools currently available for scientific computation are still structured based in function-driven modularizations, requiring substantial additional work and often being practically impossible to maintain and extend the systems, even for minor changes. The main reason of this is that the scientific software is still implemented using procedural languages (e.g., in FORTRAN) or OOP languages but following essentially procedural designs. Typical symptoms of this kind of troubles is that often large scale numerical problems are performed employing several tools generating partial calculations which afterward are processed using other applications in order to obtain the final results. There are some interesting coding efforts of complex numerical methods applying OOP (Langtangen and Munthe, 2001; Machiels and Deville, 1997; Liu *et al*., 1996; Kees and Miller, 1999). Most of them conclude that the roll played by the mathematical approach to each particular problem is crucial to achieve good software designs.

Continuous simulation calls for interesting requirements from the programming language and the modeling environment (Aqel, 2006; Mohamed *et al*., 2008). The diversity of the simulated phenomena, often requiring interdisciplinary approaches, leads to increasingly complex designs covering progressively wider scopes.

Corresponding Author: Gustavo Boroni, CONICET and Universidad Nacional del Centro,7000 Tandil, Argentina
Tel/Fax: +542293439690

The latter clashes with the demands of simplifications, modifications and multidimensional structure, not only from the point of view of the construction of models with structured methodologies (e.g., blocks) but also in the numerical solution of the underlying equations. Formal programming languages have the advantage of univocacy which is a problem of some block diagrams or textual descriptions (Hakman and Groth, 1999). For example, the mathematical expression z = y/3 + 2x, will always generate an executable code yielding the same results independently of the code generating tool. However, classical formal languages usually lead to structural, conceptual and topological information loss in the equation mass. In turn, block diagrams preserve the structural information but often darken the mathematical representation, hiding the local behavior and the conceptual information.

Nevertheless simulation software definitely goes toward OOP languages and Object Diagram Editors (Otter and Elmqvist, 1995), for they have demonstrated wider scopes and represents more naturally the physical reality. Therefore the application of OOP to continuous simulation problems by means of numerical solvers of differential equations combines the structural and mathematical information, without hiding one from the other.

In this article, a study of the problems arising in the application of OOP in continuous modeling and simulation by means of systems of combined Differential and Algebraic Equations (DAE) is presented. The purpose of the study is to assess an appropriate OO design for solving DAE initial-value problems in a generalized approach, in order to facilitate the task of the modeler that often involves a process of progressive model sophistication, modifying, extending and eliminating equations and system variables. It will be shown that in this type of problems involving a strong bias toward functionality relative to data structures it is important to start from mathematical approaches compatible with OO methodology. In the case of continuous simulation this leads naturally to implicit numerical schemes. Within this type of schemes, the family of Backward Differentiation Formulas (BDF) appears as a solid candidate compatible with software flexibility and modifiability requirements.

CONTINUOUS SIMULATION

Generally speaking, a continuous simulation is a numerical representation of temporal-evolving entities based in synchronized processes; therefore time is a single global variable serving as reference of all other variables. This concept which can look obvious to those familiarized to numerical simulation through differential equations is not the only possibility to simulate dynamical systems (Bhatti *et al*., 2006). There exists also the concept of local time, used in the so called discrete simulation, where each thread of the general process is associated to a different time line, all of which should be synchronized by means of certain defined criteria.

Physical systems involving time dependent variables are generally represented within continuous simulation frameworks by means of ordinary differential equations. Problems involving additionally the space as independent variable are more complex and should be represented with partial differential equations. The target of the present work is the first class of problems i.e., a single independent variable although some of the conclusions can support the treatment of partial differential equations.

Let us consider for example a simple system of two tanks with two pumps transferring water between them (Fig. 1).

The dynamics of this system can be described as a first approximation with two differential equations for the water volumes and two algebraic equations for the flow rates:

$$\begin{aligned} \dot{v}_1 &= q_1 - q_2 \\ \dot{v}_2 &= q_2 - q_1 \\ q_1 &= p_1(v_{ref} - v_1) \\ q_2 &= p_2(v_{ref} - v_2) \end{aligned} \quad (1)$$

where a simple proportional control of the water levels is assumed, aiming to keep a reference volume, in both tanks.

The usual procedure to numerically solve such problem is to apply a classical integration scheme (i.e., Runge-Kutta), for the sake of which the set of equations is written as:

$$\dot{\vec{x}} = \vec{f}(\vec{x}) \quad (2)$$

where, is the so called state vector. In this case:

$$\begin{aligned} \vec{x} &= (v_1, v_2) \\ f_1 &= (p_1 - p_2)v_{ref} - p_1 v_1 + p_2 v_2 \\ f_2 &= (p_2 - p_1)v_{ref} + p_1 v_1 - p_2 v_2 \end{aligned} \quad (3)$$

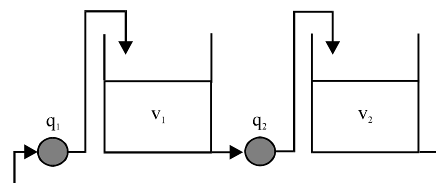

Fig. 1: Tanks of regulation by pumps

During regular modeling processes, the equations become more and more complicated as additional terms and variables are introduced in order to represent more details of the reality. The specialist task is to experiment with the successive sophistications following a series of hypotheses and comparisons against reality. The following are a list of typical modifications usually encountered during the development of dynamical models which complicate the modeler job:

Change a constant by a function of the state variables: During the development of differential dynamical models, it is usual to extend a system introducing dependencies of the control parameters on the state variables. This modification in turn can increase programming complications when the data structure is too rigid. For example, in the base case the model can be extended by postulating a volume dependence of the control parameters, p_i (i.e., non-linear control). If change generality is intended, instead of changing the flow rate equations one should change the declaration of from constant to auxiliary variable and then include an additional line with the volume function. Moreover, depending on the way each data type (constant, auxiliary variable, state variable) is communicated, consistency should also be ensured during the calculations in the data transfer between program modules.

Change a constant by a state variable: This modification is similar to the precedent but here a differential equation is added to account for the variation of the parameter. The inconveniences are analogous than before.

Replace a differential equation by an algebraic equation: This change is a very common transformation when working with balance equations of the type:

$$\dot{E} = \phi_{in} - \phi_{out} \tag{4}$$

where the increase and decrease rates of the variable, E ϕ_{out}, y ϕ_{in}, are functions of the state variables.

If the time constant of this equation is much smaller than the other system equations, a quasi-static approximation can be written by zeroing the temporal derivative, that is:

$$\phi_{in} = \phi_{out} \tag{5}$$

which is an algebraic equation. Modifications of this type change the order of the system of equations, affecting among other things the time step control. However, the main inconveniences is not this but the fact that the entire set of equations should be rewritten all over again to produce the appropriate form required by explicit schemes, that is $\dot{\vec{x}} = \vec{f}(\vec{x})$. In effect, after Eq. 4 is eliminated the variable E is no longer a state variable, although it will continue to be called to calculate $\vec{f}(\vec{x})$, an therefore it should be calculated from the auxiliary algebraic equations including the new one. These annoyances imply reprogramming of the model, with the additional charge of associated implementation errors.

Replace an algebraic equation by a differential equation: This change is the transformation opposite to the precedent. In this case also inconveniences in the time step control and the reimplementation can appear.

Add a term depending on a temporal derivative: This change typically appears when a derivative control term is included. For example in the level control case:

$$\begin{aligned} q_1 &= p_1\left(v_{ref} - v_1\right) - d_1\dot{v}_1 \\ q_2 &= p_2\left(v_{ref} - v_2\right) - d_2\dot{v}_2 \end{aligned} \tag{6}$$

This kind of modification is complicated for it requires rewriting the system as $\dot{\vec{x}} = \vec{f}(\vec{x})$ which imply as in the case 3 the reprogramming of the model.

Generate a system of N instances: This type of change is usual in the modeling of processes with a large number of similar components. Typical examples are the chemical processes, piping transport and population models. Actually the spatial discretization of partial differential equations generally leads to sets of many similar ordinary differential equations coupled (cells). In the level control case, the extension to N instances is direct, yielding:

$$\begin{aligned} &\dot{v}_1 = q_1 - q_2 && q_1 = p_1\left(v_{ref} - v_1\right) \\ &\dots && \dots \\ &\dot{v}_i = q_i - q_{i+1} && q_i = p_i\left(v_{ref} - v_i\right) \\ &\dots && \dots \\ &\dot{v}_N = q_N - q_1 && q_N = p_N\left(v_{ref} - v_N\right) \end{aligned} \tag{7}$$

In such cases it is very useful for the modeler to make use of the inheritance property which enables encapsulate the common parts of the subsystems, saving implementation work. Clearly the inconveniences associated with all the preceding modifications increase by a factor N with this extension.

SYSTEMS OF DIFFERENTIAL AND ALGEBRAIC EQUATIONS (DAE)

The numerical solution of initial-valued problems modeled by DAE attracted the interest of the numerical community form the last 30 years (Brenan *et al*., 1996).

Many engineering and scientific problems can be naturally modeled with DAE, yielding mostly to balance equations of the general form:

$$\vec{F}(\dot{\vec{x}}, \vec{x}, \vec{y}) = 0 \tag{8}$$

together with a set of auxiliary algebraic equations representing restrictions or constitutive relations:

$$\vec{G}(\vec{x}, \vec{y}) = 0 \tag{9}$$

In order to ensure that real initial-valued problems can be appropriately represented, the specific form of Eq. 8 and 9 should ensure that the temporal derivatives are uniquely determined for every valid set of values of and. In such cases one can always transform a DAE in a system of Ordinary Differential Equations (ODE) of the type:

$$\dot{\vec{x}} = \vec{f}(\vec{x}) \tag{10}$$

which can be solved with some of the classical numerical solver schemes (Runge-Kutta, etc.).

The actual transformation of a DAE to the form given by Eq. 10 can be troublesome for various reasons. For instance, DAE often increase their stiffness when they are rewritten as ODE. However, even if this were not the case, the interest in keeping the DAE structure during the numerical calculation arises from the search of efficient software designs, particularly from the point of view of the modeler.

The general objective of the study can be therefore written as a totally implicit non-linear function of the form:

$$\vec{F}(\dot{\vec{x}}, \vec{x}) = 0 \tag{11}$$

which are often called residual functions and are taken as a combination of differential and algebraic equations. An Initial-valued Problem (IVP) consists of determining the temporal functions $\vec{x}(t)$ that satisfy Eq. 11 given the values $\vec{x}_0$ in t = 0.

There are a variety of numerical methods for solving IVP of DAE which can be classified in three classes: single-step, multistep and extrapolation methods. The efficiency of each type of approximation depends on the specific problem. In what follows, the software-design aspects of a subclass of multistep method, BDF (Backward Difference Formulas) (Gear, 1971), having interesting flexibility properties, will be studied.

Generally speaking, BDF is based in transforming the DAE in a purely algebraic system by means of numerical approximations. In order to do that let us define a transformation replacing $\vec{x}$ and $\dot{\vec{x}}$ in the residual Eq. 11 by functions $\vec{X}(\vec{x}_n, \vec{x}_{n-1}, \vec{x}_{n-2}, \ldots)$ and $D\vec{X}(\vec{x}_n, \vec{x}_{n-1}, \vec{x}_{n-2}, \ldots)$, representing multistep estimators of the state vector and its derivatives (by convention $\vec{x}_k = \vec{x}(t_k)$). For example, a semi-implicit 2-step scheme would be:

$$\vec{X} = \frac{\vec{x}_n + \vec{x}_{n-1}}{2} \qquad D\vec{X} = \frac{\vec{x}_n - \vec{x}_{n-1}}{\Delta t} \tag{12}$$

This operation defines a set of estimated residuals:

$$\vec{G}(\vec{x}_n, \vec{x}_{n-1}, \vec{x}_{n-2}, \ldots) = 0 \tag{13}$$

Equation 13 are a set of algebraic equations whose unknown variables are the state vector in the new time, $\vec{x}_n$ which is the calculated knowing its past history.

Within the described general procedure, different alternatives exist according to the specification of:

- The formulas $\vec{X}$ and $D\vec{X}$
- The control strategy of the time step
- The numerical method for finding the roots of Eq. 13
- The data structure and classes to handle the different steps of the general algorithm

The abstract BDF method presented previously yields to an advantageous calculation scheme for the model developments, mainly because it solves naturally most of the problems introduced by the typical modifications appearing during the modeling of system dynamics. Actually, the basic functional form $\vec{F}(\dot{\vec{x}}, \vec{x}) = 0$ is invariant under any of the mentioned model changes mentioned in the previous section. From this perspective, one is tempted to say that implicit schemes are a "natural mathematics" for OO, the "natural" objects being the functions $\vec{F}(\dot{\vec{x}}, \vec{x})$. The implementation of the numerical solution can be easily encapsulated, leading the modeler to "see" a solver "black box" in which he or she introduces, adds, changes, eliminates, functions $\vec{F}(\dot{\vec{x}}, \vec{x})$, without worrying about rewriting the equations or data communication problems.

ANALYSIS AND OO DESIGN OF DAE SOLVERS

From the point of view of programming, traditional subroutine libraries available to solve differential equations typically present two main weaknesses: rigid interfaces and complexity exponential with the variety of equations to solve. The first weakness affect the users for

it is relatively difficult to elaborate models using these libraries, the modeler assuming a double roll: to represent its model mathematically and to adapt the resulting representation to the required interface. The second problem affects the library developers, since the number of programming and maintenance tasks explode.

The mentioned troubles are consequence of deficient abstractions in the design phases. A first problem is that in order to optimize the numerical performance the access to subroutine is often forced to rigid data structures which require the implementation of new subroutines when any of the data structures is changed during the modeling process. An addition inconvenience is that these subroutines often include input variables that describe the data structure.

Although better implementations of numerical methods can be produced using OO programming, one should remind that there always will be a penalty on the performance. Therefore, since numerical simulation often requires complex and highly efficient codes, the performance of high level programming languages will compete with the design improvements achieved from their application. In this section, an analysis of these issues and the results obtained during the design and implementation of a general DAE solver tool for continuous simulation are presented.

Analysis: Essentially a DAE is a set of scalar functions $F = \{F_1 (A, B), F_2 (A, B), ..., F_N (A, B)\}$, where A is a set of N functions of the temporal coordinate (real and independent) and B are the corresponding temporal derivatives of A. The methodology BDF consists of transforming the set F in a system of N algebraic equations $G \{G_1 ([X]), G_2 ([X]), ..., G_N ([X]) = 0$, where [X] is a string of N elements representing the values of the functions A evaluated at current time. The transformation $F \rightarrow G$ is performed assigning to each A and B element an multistep estimator function f ([[X]]) , where [[X]] represents a list of strings [X] increasing its length as time advance (i.e., the elements of the list [X] are the values of A calculated at each time step).

Design: The purpose of the system to implement is the numerical solution of DAE sets, trying to maintain flexibility to support appropriately the development of mathematical models for continuous simulation. Two methods will be implemented and compared: explicit fourth-order Runge-Kutta and generalized BDF. Runge-Kutta methods and BDF method were implemented to solve the explicit and implicit schemes respectively. In what follows, the following design aspects will be omitted for the sake of clarity: constructors, destructors and methods of consultation of attributes. The syntax for attributes declaration, methods and pseudocodes is a relaxed syntax C++ but the schemes can be implemented in any OOP technology (e.g., JAVA, Small talk). If using another OOP technology to implement the proposed schemes, the comparative results will be similar.

Classes for the implementation of implicit schemes

DAE system: This class defines generically DAE sets of the type given by Eq. 11. Dynamic matrices are used to represent the dependent variables $\vec{x}$ and $\dot{\vec{x}}$, the columns being associated to the system variables and the rows representing the discretized time coordinate (Fig. 2). The functions $\vec{F}$ are declared on the method "Calculat-F()" which receives through parameter the estimation of $\vec{x}$ and $\dot{\vec{x}}$, returning the corresponding values of $\vec{F}$. Since a single problem can imply several systems of equations linked through the variables, an association relation 0..* was included between objects of the same class. Moreover, the following methods were defined to complete the functionality of this class:

- "Initial-guess()": used to obtain the seed for the solver of $\vec{G}$ roots, $\vec{x}_{guess}(t)$
- "Posible-solution()": stores the roots of $\vec{G}$, taken as a possible solution of the current state vector which should be tested for consistency before being accepted
- "Update-history()": accept a possible solution that pass the consistency test
- "Retrieve-history()": retrieve the past values of the state vector

Controller: This class represents the master control responsible for synchronizing the entire calculation process. *Controller* manages the current time and the time step. The method "Advance()" is used to advance the system a time step beginning from the present time. During the execution of "Advance()" the last calculated values of the state vector is stored in the history list and the current time step is calculated calling the method "Calculate-New-DT()" (Fig. 3).

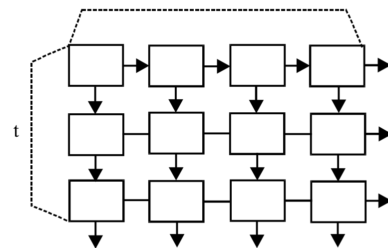

Fig. 2: Temporary dynamic representation of the variables

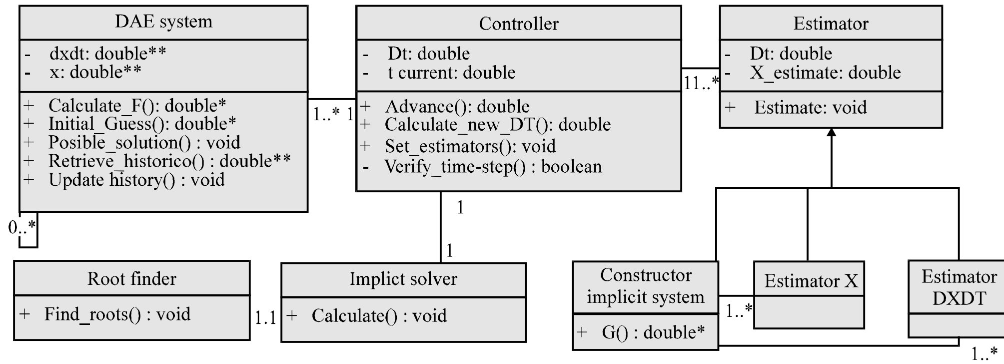

Fig. 3: Class diagram for implicit schemes

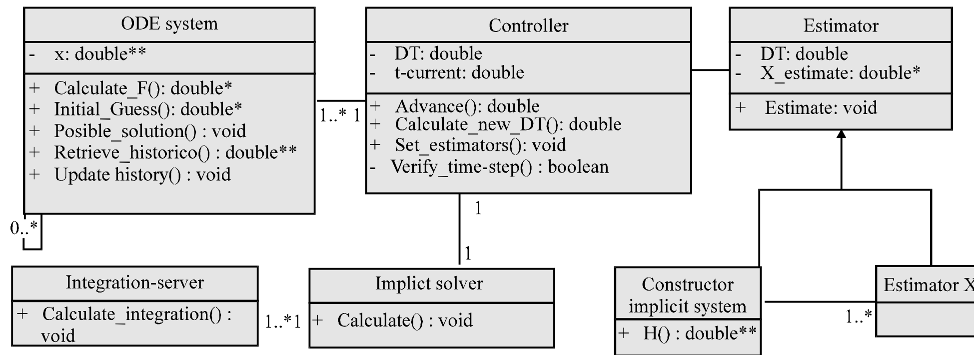

Fig. 4: Class diagram for explicit scheme

Estimator: This is an abstract class defining generically relations of the variables of $\vec{F}$ with the state vector and the current unknown state. The abstraction guarantees the possibility of user implementation as well as the use of libraries of estimators. The current time step is included as class property. The subclasses "Estimator DXDT" and "Estimator X" define the relation of the temporal derivative and the state vector with the past values of the state vector and the corresponding unknown current value.

Constructor implicit system: This is another subclass of "Estimator" defining the method "G()" which represent the algebraic equations associated to the residual functions $\vec{F}$.

Root finder: This is an abstract class defining generically the numerical methods that solve the roots of the algebraic equation $\vec{G}=0$. All class that inherits from "Root Finder" should implement the method "Find-Roots()" having the algebraic function "G()" and the guess values $\vec{x}_{guess}$ as input parameters and returning the root vector $\vec{x}_{root}$.

Implicit solver: This class constructs the sequence required to transform the user model equations to an implicit algebraic problem. The method "Calculate()" receive as a parameter an instance of the class "DAE System".

Classes for the implementation of explicit schemes: Most of the classes designed for explicit schemes are similar to those corresponding to the implicit schemes (Fig. 4). The main differences are the following:

ODE system: This class specifies generically systems of ordinary differential equations defined by Eq. 10. The functions of the derivatives, $\vec{f}$ and the auxiliary algebraic equations are declared in the method "Calculate_DXDT()" which receives as input the values of $\vec{x}$ and delivers the values of $\dot{\vec{x}}$.

Constructor explicit system: This is a subclass of "Estimator" defining the method "H()" that represents the algebraic equations associated to the functions $\vec{f}$.

Explicit solver: This class builds the calculation sequence of the explicit scheme which is performed through the method "Calculate()" that receives as input an instance of the class "Explicit System".

RESULTS

Numerical study of the model tanks: Consider a water level control of a series of tanks (Fig. 1). This case is appropriate to analyze the advantages of implementing BDF schemes in a modeling process, starting from a simple base case (proportional control) and progressively rising the model sophistication following a circular process of trial and error. Two schemes were implemented following the OO paradigm, BDF and explicit Runge-Kutta. Using both implementations, different modifications of the model were applied, assessing in each case the software flexibility.

The importance of each modification was characterized by the following parameters:

- Number of equations affected by a modification (r1)
- Number of equations that should be added to the model after applying the required algebraic steps (r2)

The software adaptability to each modification was characterized by the following indicators:

- Number of additional code lines required (r3)
- Intermediate algebraic steps required to implement a modification to the model (r4)

A series of two tanks with classical proportional level control was taken as the base case. The dynamics of this system can be described by two differential equations for each water volume and two algebraic equations for the flow rates:

$$\begin{aligned} \dot{v}_1 &= q_1 - q_2 \\ \dot{v}_2 &= q_2 - q_1 \\ q_1 &= p_1\left(v_{ref} - v_1\right) \\ q_2 &= p_2\left(v_{ref} - v_2\right) \end{aligned} \tag{14}$$

where, v_{ref} is the reference volume.

The described base model was extended to series of 3 and 4 tanks and the following modifications were imposed to each extension:

- Include derivative control terms
- Introduce a dependence of the control coefficients on the state variables
- Transform the algebraic equations in differential equations

The initial conditions and the values of the constant parameters in each particular case are detailed in Table 1. The indicators r_i were calculated in each case, are detailed in Table 2. For example, include one derivative control term $\delta_1\dot{v}_1$ (case a in Table 1) for tank 1 in the explicit scheme, the number of equations affected by a modification is r1 = 2 ($\dot{v}_1 = q_1 - q_2$ and $q_1 = p_1\ (v_{ref}\text{-}v_1)$, the number of equations that should be added to the model after applying the required algebraic steps is r2 = 0 (replace the original equations), the number of additional code lines required is r3 = 2 (declaration and initialization of δ_1) and the intermediate algebraic steps required to implement a modification to the model r4 = 3 (steps to obtain the expression of $\dot{v}_1$).

The general observation is that the BDF implementation reduces substantially the number of code lines and the number of intermediate algebraic steps. However, similarly to the previous cases, this advantage increase the computer costs which calls for pondering the importance of each software characteristic (i.e., modifiability-extensibility vs., calculation time). The equilibrium of this balance is determined by the size of the problem to solve. If the number of equations of the model is high modifiability and extensibility are the preponderant factors, whereas in models with few equations the gain in software implementation does not match the decrease in numerical performance.

To analyze the mentioned balance two metrics are defined that represent the competition between both elements of quality, software metric, S and numerical metric, C:

S = Number of code lines plus number of algebraic steps (r3 + r4)
C = Real time/calculation time

Table 1: Study of the model tanks - Initial conditions and values of parameters

Modifications	Equations	Initial conditions
a) Include derivative control terms	$q_i = p_i(v_{ref} - v_i) - \delta_i\dot{v}_i \dot{v}_i = q_i - q_{out}$	$v_{ref} = 10, p_1 = 0.2, p_2 = 0.1,\ \delta_1 = 0.5, \delta_2 = 0.5,\ v_1(0) = 10.01, v_2(0) = 10$
b) Introduce a dependence of the control coefficients on the state variables	$q_i = p_i(v_{ref} - v_i) - \delta_i\dot{v}_i\ \ \dot{v}_i = q_i - q_{out}$ $p_i(v_1,..,v_n) = k_i(v_1 + .. + v_n)$ $\delta_i(v_1,..,v_n) = c_i(v_1 + .. + v_n)$	$v_{ref} = 10, k_1 = 50, k_2 = 100$ $c_1 = 20, c_2 = 20, v_1 = 10.01\ v_2 = 10$
c) Transform the algebraic equations in differential equations	$I_i\dot{q}_i + K_i q_i \lvert q_i \rvert = \Delta p_{A,i}$ $\Delta p_{A,i} = f(v_i) - g(\dot{v}_i) + \Delta p_{B,i}$ $f(v_i) = p_i(v_{ref} - v_i)\ g(\dot{v}_i) = \delta_i\dot{v}_i$ $\Delta p_{B,i} = \Delta p_B(\dot{v}_i)\ \dot{v}_i = q_i - q_{out}$	$v_{ref} = 10, p_1 = 0.2, p_2 = 0.1$ $\delta_1 = 0.5, \delta_2 = 0.5, K_1 = 1$ $K_2 = 1, I_1 = 2, I_2 = 2$ $\Delta p_B = 0.05, v_1 = 10.01, v_2 = 10$

Table 2: Characterization parameters of the proposed modifications, software metric S and numerical metric C

		BDFG[1]						RK4[2]					
#Tanks	Modifications	r1	r2	r3	r4	S	C	r1	r2	r3	r4	S	C
1	a)	2	0	2	0	2	0.196	2	0	2	3	5	0.005
	b)	4	2	4	0	4	0.198	4	2	4	3	7	0.006
	c)	6	3	6	0	6	0.21	6	3	6	4	10	0.007
2	a)	4	0	4	0	4	0.198	4	0	4	6	10	0.006
	b)	8	4	8	0	8	0.212	8	4	8	6	14	0.007
	c)	12	6	12	0	12	0.213	12	6	12	8	20	0.007
3	a)	6	0	6	0	6	0.196	6	0	6	9	15	0.006
	b)	12	6	12	0	12	0.208	12	6	12	9	21	0.008
	c)	18	9	18	0	18	0.21	18	9	18	12	30	0.008
4	a)	8	0	8	0	8	0.212	8	0	8	12	20	0.007
	b)	16	8	16	0	16	0.211	16	8	16	12	28	0.008
	c)	24	12	24	0	24	0.214	24	12	24	16	40	0.010
20	a)	20	0	20	0	20	0.198	20	0	20	30	50	0.011
	b)	40	20	40	0	40	0.23	40	20	40	30	70	0.016
	c)	60	30	60	0	60	0.421	60	30	60	40	100	0.021

[1]BDF: Backsword diffential formulas, [2]RK4: Runge-kutta 4 order under score

Table 2 shows the calculated values of S and C for each modification of the tank-series problem. Figure 5 shows the dependence of S and C with the number of equations which can be taken as the model size. It can be seen that C is lower in the BDF implementation, whereas the opposite occurs with S.

The conclusion is that the optimum scheme will ultimately depend on the relative importance that the modeler assigns to the metrics C and S. This type of decision is usually solved by means of a utility function representing the subjective usefulness perceived by the users for a given pair (C, S). In the present analysis, a utility function similar to those used in microeconomy is proposed, that is (Henderson and Quandt, 1980):

$$U = \exp(-S-\alpha C) \tag{15}$$

where, α represents the relative importance of the numerical performance respect to the software flexibility. The best alternative is determined by the scheme having the higher value of U.

Figure 6 shows the optimum decision map calculated for the modeling of the series of tanks, in the plane defined by the parameter α and the number of equations. It can be seen that for α values higher than 10 explicit Runge-Kutta implementations are more convenient for systems of few equations, whereas BDF schemes are better when the number of equations exceed a certain threshold. The critical number of equations separating both alternatives is lower for lower values of α. This means that as the importance software implementation quality increase, the minimum system size for convenience of implicit BDF implementations decrease. For α values lower than 10, BDF implementations is always recommended.

Similarly than in economy, the value of α in particular cases can be estimated by means of the relative impact of

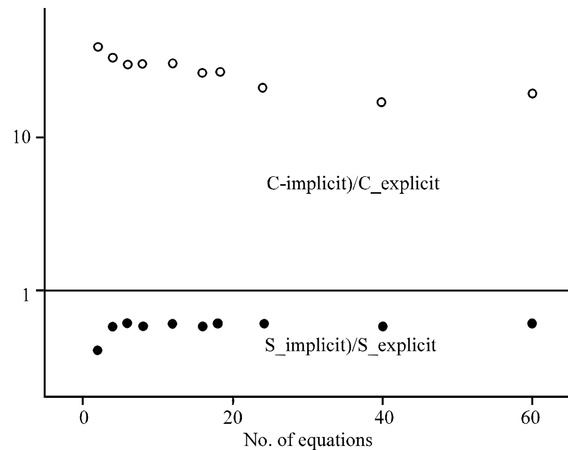

Fig. 5: Relations between software metric, S and numerical metric, C

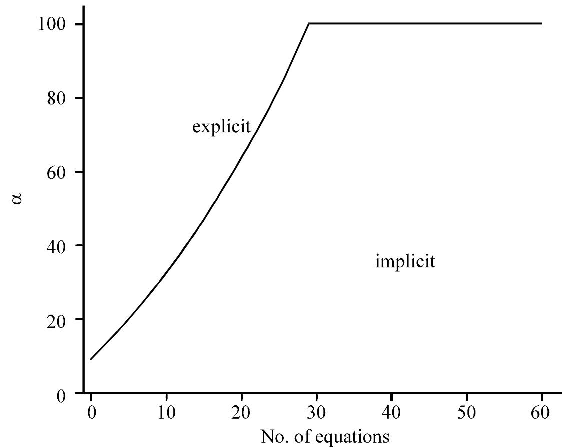

Fig. 6: Optimum decision map calculated for the modeling of the series of tanks

the factors C y S on the utility. In the software context, α can be interpreted as the number of S units producing the

same impact than a C unit on the development work which depends on the developer skills and what type of work requires more efforts and resources.

Implementation of the core model of CAREM NPP: The CAREM Nuclear Power Plant (NPP) has an integrated reactor, the whole high-energy primary system, core, steam generators, primary coolant and steam dome is contained inside a single pressure vessel (Fig. 7). The flow rate in the reactor primary system is achieved by natural circulation. The driving forces obtained by the differences in the density along the circuit are balanced by friction and produce losses, resulting a flow rate in the core that allows for sufficient thermal margin to critical phenomena.

Mathematically, the core model is represented by differential algebraic equations, corresponding to the mass and energy. All equations are expressed with implicit DAE's.

$$F_1 = A\rho_1 l_1^{(1)}(h_f + h_1)/2 + W_1(h_1 - h_f)/2 + qAl_1 - E_1^{(1)}$$

$$F_2 = A\rho_1\ l_1\ (h_f + 3h_1)/4\text{-}E$$

$$F_3 = A\rho_1 h_f\left(l_1^{(1)} + l_2^{(1)}\right) + W_1(h_1 - h_f)/2 + qAl_2 - A\rho_1 l_1^{(1)}(h_1 + h_f)/2 - E_2^{(1)}$$

$$F_4 = A\rho_1\ l_2\ (3h_f + h_1)/4\text{-}E_2$$

$$F_5 = \text{-}D_{pg}\ \text{-}D_{pa}\ \text{-}D_{pf}\ \text{-}D_{pe}\ \text{-}D_{pfi}\text{-}I^{(1)}$$

$$F_6 = gm/A\text{-}D_{pg}$$

$$F_7 = \rho_e u_e^2 - \rho_1 u_1^2 - D_{pa}$$

$$F_8 = k_{2c}\rho_e u_e^2 + k_{1c}\rho_1 u_1^2 - D_{pf}$$

$$F_9 = c_f\Big(M/Au_1^2 + 2\Omega u_1\big(L - (l_1 + l_2)\big)(\rho_1 L - M/A)/(\rho_1/\rho_e - 1) + \sigma_1\Big(\Omega\big(L - (l_1 + l_2)\big)/(-1)\Big)^2(\rho_1/\rho_e - 3)\big(L - (l_1 + l_2)\big)/2 + 1/\rho_1\big(M/A - \rho_1(l_1 + l_2)\big)\Big) - D_{pfi}$$

$$F_{10} = u_1 M/A + \Omega\ (L\text{-}(l_1 + l_2)\ (\rho_1\ L\text{-}M/A)/(\rho_1/\rho_e\text{-}1)\text{-}I \quad (16)$$

$$F_{11} = u_1 + \Omega\ (L\text{-}\ (l_1 + l_2))\text{-}u_2$$

$$F_{12} = |q| + v_{fg}/h_f\text{-}\Omega$$

$$F_{13} = W_1\text{-}W_e\text{-}M^{(1)}$$

$$F_{14} = A\ (L\text{-}\ (l_1 + l_2))\ (\rho_1\text{-}\log\ (\rho_1/\rho_e)/(1/\rho_e\text{-}1/\rho_1)) + A(l_1 + l_2)\text{-}M$$

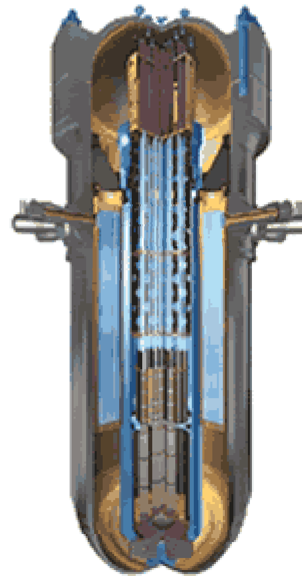

Fig. 7: CAREM nuclear power plant-single pressure vessel

$$F_{15} = \rho_1\text{-}\rho_f$$

$$F_{16} = \rho_e\ Au_e\text{-}W_e$$

$$F_{17} = \rho_1\ Au_1\text{-}W_1$$

$$F_{18} = qAL\ v_{fg}/W_1\ v_f\ h_{fg}\text{-}N_{pch}$$

$$F_{19} = v_f\text{-}(0.0010941477378 + P\ (3.6380424746E\text{-}11)$$

$$F_{20} = v_g\text{-}(0.042764801477\ P\text{-}2.31776361958E\text{-}09))$$

$$F_{21} = v_{fg}\text{-}v_f + v_f$$

$$F_{22} = h_f\text{-}(96777.46999 + P\ (0.043951985226))$$

$$F_{23} = h_{fg}\text{-}(1939710.988\text{-}P\ (0.06241920591))$$

$$F_{24} = \rho_f\text{-}1/v_f$$

In Fig. 8 it can see the classes implemented for this example. The core model has 6 differential equations and 18 algebraic equations, where several of them are nonlinear. The update process through the implicit scheme is very simple because the changes are local. On the other hand, these equations implemented with the explicit scheme can be difficult, especially in some

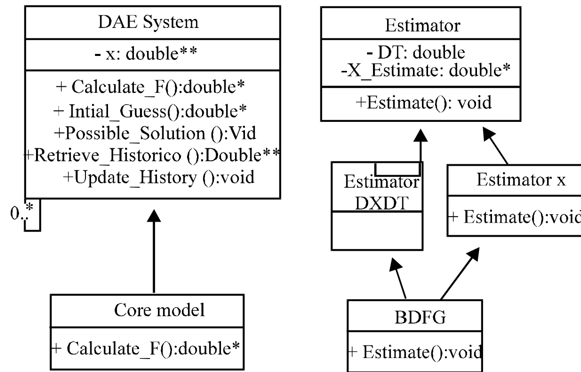

Fig. 8: Class diagram of core model

equations that may have more derivative terms. A minimum change to a component linked to another may cause changes for all components of the reactor. Also, there are algebraic equations which the conversion to ODE's can be causing a double error of approximation.

CONCLUSIONS

The application of OO designs to improve the development of continuous simulation systems was studied. The analysis has tackled the difficulties and complications that affect the work of modelers and library developers, either about the numerical performance or the software implementation.

A class structure providing rapid implementations of continuous dynamic simulation problems was presented, emphasizing the natural representation by means of DAE systems. This approach leads naturally to implicit numerical schemes among which BDF was found compatible with OOP. However, improvements in software flexibility and reusability introduce numerical penalties that should be taken into account in the general balance between flexibility and performance.

It was demonstrated than a metric established in some cases to determine the most convenient scheme, explicit or implicit, regarding the user requirements. The concepts of inherited and encapsulation play an important roll, producing programs much more modular than the procedural versions, thus promoting code reusability, extensibility and maintainability.

A recent research direction on modeling and simulation is to use Model Based Designs (MBD) (Bhatta and Goel, 1996) which helps to accomplish the type of design modifications mentioned with relatively little overhead compared to procedural methodologies. MBD is a mathematical and visual method of addressing the problems associated with designing complex control systems and is being used successfully in many motion control, industrial equipment, aerospace and automotive applications. However highly coupled non-linear systems (e.g., nuclear reactor neutronics, thermohydraulics and control dynamics) still can involve a strong overhead of design and testing, or otherwise increase the computational cost associated to the precision requirements of some engineering problems. Indeed, the use of DAE based models is complementary to MBD and in fact they can be implemented in MBD tools.

REFERENCES

Adetunde, I.A., 2009. The mathematical models of the dynamical behaviour of tuberculosis disease in the upper East region of the Northern Part of Ghana: A case study of bawku. Curr. Res. Tuberculosis, 1: 15-20.

Aqel, M.M., 2006. A simulation technique for engineering control systems. J. Applied Sci., 6: 157-162.

Bhatta, S.R. and A.K. Goel, 1996. Model-based design indexing and index learning in engineering design. Engin. Applic. Artificial Intelli., 9: 601-609.

Bhatti, M.A., L.C. Xi and Y. Lin, 2006. Modeling and simulation of dynamic systems. J. Applied Sci., 6: 950-954.

Brenan, K.E, S.L. Campbell and L.R. Petzold, 1996. Numerical Solution of Initial Value Problems in Differential-Algebraic Equations. 2nd Edn., SIAM., Philadelphia, PA., USA., ISBN-13: 9780898713534, pp: 266.

Gear, C., 1971. The simultaneous numerical solution of differential-algebraic equations. IEEE Trans. Circuit Theory, 18: 89-95.

Hakman, M. and T. Groth, 1999. Object oriented biomedical system modeling: The rationale. Comput. Meth. Programs Biomed., 59: 1-17.

Henderson, J.M. and R.E. Quandt, 1980. Microeconomic Theory. McGraw-Hill, New York, USA.

Kees, C.E. and C.T. Miller, 1999. C++ implementations of numerical methods for solving differential-algebraic equations: Design and optimization considerations. ACM Trans. Math. Software, 25: 377-403.

Langtangen, H.P. and O. Munthe, 2001. Solving systems of partial differential equations using object-oriented programming techniques with coupled heat and fluid flow as example. ACM Trans. Math. Software, 27: 1-26.

Liu, J.L., I.J. Lin, M.Z. Shih, R.C. Chen and M.C. Hsieh, 1996. Object-oriented programming of adaptive finite element and finite volume methods. Applied Num. Math., 21: 439-467.

Machiels, L. and M. Deville, 1997. Fortran 90: An entry to object-oriented programming for the solution of partial differential equations. ACM Trans. Math. Software, 23: 32-49.

Meyer, B., 2000. Object-Oriented Software Construction. 2nd Edn., Prentice Hall, UK.

Mohamed, I., A.G. Hussin and A.H. Abdul Wahab, 2008. On simulation and approximation in the circular gression model. Asian J. Math. Statist., 1: 100-108.

Otter, M. and H. Elmqvist, 1995. The DSBlock model interface for exchanging model components. Proceedings of thte EUROSIM, (EUROSIM'95) Viena, USA. pp: 505-510.

Yang, M.F., 2008. Using simulation to object-oriented order picking system. Inform. Technol. J., 7: 224-227.