PARALLELIZATION OF MODULAR SIMULATION PROGRAMS IN AN OBJECT ORIENTED ENVIRONMENT : THE TRNSYS CASE

P.Y. GLORENNEC
INSA Rennes
20 Av. des Buttes de Coësmes
35043 Rennes Cedex
France

R. EL BOUSSARGHINI
E.S.E Antenne de Rennes
Av. de la Boulaie B.P. 28
35511 Cesson Sévigné
France

Abstract
From the observation that existing simulation programs exploit neither the subjacent parallelism in building energy management problems nor parallel computer possibilities, we develop certain principles and apply them to a well-known program, TRNSYS. To this end, we can call on the possibilities offered by object oriented programming, which permits in particular :
- the easy re-use of the existing programs
- the transformation of a module into an independent process, communicating with other processes in a flexible data processing architecture (mono or multiprocessor machines)
- the display of an execution scheme, common to all the processes of the system;

We have developed a compiler which analyses the TRNSYS deck and automatically generates the necessary code for a given simulation.
The different levels permitting the communications are writen C++, in an UNIX environment. The old procedures, writen in Fortran, are re-used.

Keywords :

Object oriented programming, C++, parallelism, communications, class, models.

1. PRESENTATION

1.1. MONOLITHIC AND MODULAR PROGRAMS

Building simulation programs belong, roughly, to two categories : monolithic or modular programs.

Monolithic programs, such as DOE-2, are generally well-adapted to their task and use efficient algorithms. On the other hand, their rigidity is a drawback; the user will find difficulties to use them in a non-conventional situation, not appearing in the menu.

On top of that, their monolithic aspect makes them difficult to use, in multiprocessors systems, unless we enter in the code of the program itself.

On the other hand, the use of modular programs is more flexible, because, with only a small amount of programming, the user can modelize his own system, by drawing on the provided components (modules) or by creating his own modules.

Current programs are of quite an old conception : most of them date from the seventies (TRNSYS, DOE-2, HVAC SIM + ...) and even if they have evolved a lot (TRNSYS is now in release in its 13.1 version) they stay marked by the initial data processing concepts :

- sequentiality
- mono-processor computers
- low level procedural languages (Fortran 77, Fortran IV)
- user interface weakness.

The current situation is, however, caracterized by numerous new events of a different nature.

1.2. THE IMPORTANCE OF SIMULATION

The importance of simulation in building has permitted the development of very numerous models, from the most basic component (pipe, wall ...) to the sophisticated component (cooling tower, microprocessor controller ...) : all these models are written in different data processing languages with different input/output agreements, so that, the user who wants to integrate a new model into a basic simulation program has to devote a lot of time to interface problems. A need of clarification and normalization is becoming more and more necessary, thus, object oriented languages can provide an efficient answer with encapsulation, data abstraction and inheritance, as we will see later.

1.3. PARALLEL COMPUTER

Parallel machine development (iPSC, T-NODE ...) should permit an important saving of time, and the possibility of conceiving more complex simulations. This imposes the rethinking of simulations in term of tasks and systematizing the notion of modularity.

1.4. RE-USE OF EXISTING PROGRAMS

Lastly, one of the aim which inspired this work, was the re-use of the existing programs, insofar as a lot of them have been tried, tested. On top of that, rewriting them would necessitate considerable expense in effort, time and money.

2. AN EXAMPLE : TRNSYS

TRNSYS [TRNSYS 89] is a good example of a candidate for parallelization :
- its modular structure naturally prepodisposes it to cutting out and placing on different processors.
- its old conception needs a modernising according to the conceptions expressed above.
- simulations are time driven, which facilitates parallelization.

2.1. TRNSYS structure

A certain number of utilities are disposed around the principal program :

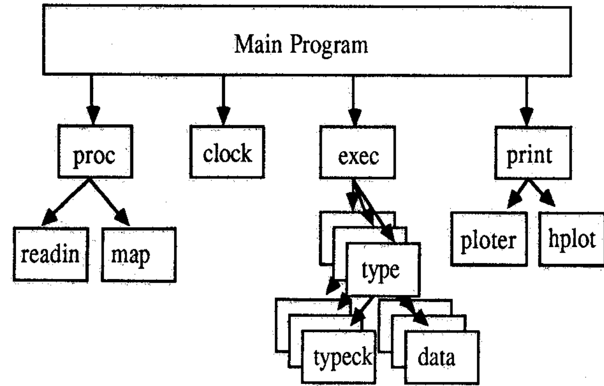

fig 1- TRNSYS Structure

As the diagram shows, there is no direct relation between the different components (TRNSYS types), which compose a system. These components are called sequentially by the utility "*exec*" itself under the main program control.

At each time step all the components (types) know their input values, and so, could start running. But the fact they are called sequentially makes each wait its turn, so the total run period is the sum of the run-periods for each component.

2.2. REMODELLING OF TRNSYS

2.2.1. PRUNING AND DIVISION OF LABOUR

The main program of TRNSYS must process too many tasks, which could be subcontracted ;

- free format reading of the input deck,
- standard check of the deck ("typeck"),
- printing tasks,
- checking the convergence of algebraic and differential equations,
- differential equation solving.

Solving these tasks implies a certain heaviness and frequent returns between different programs. Two examples of immediate simplification :

. elaboration and checking the input deck can be done independently of the actual simulation, with more elaborate type checking and realisation of more structured file. This is not a handicap : an analysis by Lex and Yacc will be easier. [Riaux 91].

. the printing tasks can be treated separately, using developed graphical functions and window possibilities.

2.2.2. COMPONENT "EMANCIPATION"

The different components become objects, communicating each other by messages. Each object is seen from outside under an unified form, which totally respects the TRNSYS module conception.

From that point of view :
- the only data in circulation is the data exchanged between the objects, the rest is cancelled.
- the necessary COMMON data is duplicated in each object memory zone.

As we explain in the next paragraph, each object becomes an autonomous, unified entity. A simulation lies in managing the exchanges between those different objects, these can be placed on different processors of a parallel computer or on different computers in an Ethernet network.

2.2.3. CONVERGENCE

The TRNSYS method of solving equations by successive substitutions is not very efficient. However, since the 12.2 release, a special component, the "*convergence promoter*" speeds up the convergence for algebraic equations. This component, type 44, becomes an object like the others.

2.2.4. CLOCK

As the main program is suppressed, a special object, the clock, has been created for synchronization and stoppingthe simulation.

3. BASIS CONCEPTS OF OBJECT APPROACH

3.1. CLASSES, OBJECTS AND INHERITANCE

Object programming [Meyer 88] is caracterized by an organization of information in autonomous cooperating entities, objects. Those entities only cooperate by messages.

An object is defined by two components :

. an internal state constituted by its variable set,
. a behaviour defined by the set of procedures which constitute the object know-how. These procedures are the methods. Moreover, we must note that the object state is modified or consulted only by calling on its methods.

The notion of class is introduced for specifying a behaviour, common to a set of objects. An object is an instance of a class.

It is interesting to complete the expression of common behaviour between objects by the expression of sharing. The inheritance rule allows this possibility, building a class by inheritance, overriding and development of an existing class.

Inheritance allows, moreover, the hierarchization of class declarations. An inherited subclass has, from its parents, variables and methods. Those methods can be overridden, i.e. redefined in a subclass, masking those declared in the parent class. The set of non overridden methods are shared between the set of instances of subclasses and parent classes.

The object internal state consists of private and shared data from the class and the superclasses. In the same way, the object interface is represented by the set of class methods and superclass methods.

3.2. ABSTRACT CLASSES AND VIRTUAL METHODS

The "*any-type*" class has been created in order to factorize the common behaviour of a set of classes. It is intended for inheritance and cannot directly be used as a model for an object . Such classes are called "*abstract classes*".

Otherwise, we notice that "*compute*" is a method common to all subclasses issued from "*any-type*". However, we can't, in this class, associate an explicit body with it ; "*compute*" is called a virtual method, it is stated in the "*any-type*" class and fully specified in all subclasses of "*any-type*".

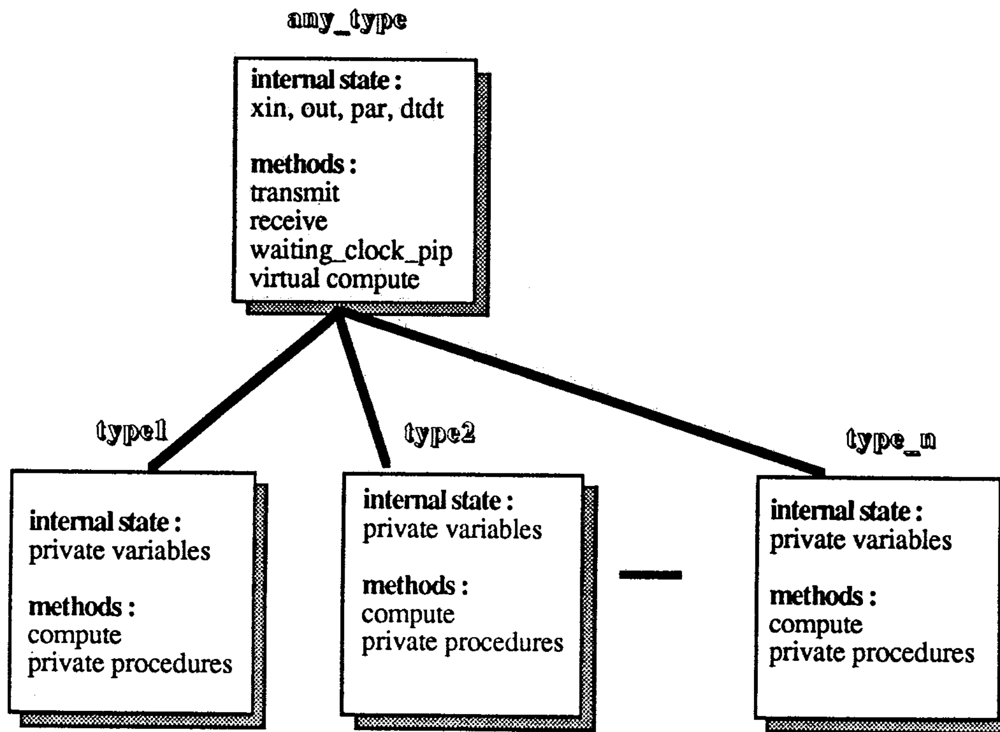

fig 2- The "any type" class and its subclasses.

4. CONNECTION BETWEEN OBJECTS AND PROCESSES

Encapsulation of a set of methods and data into a one structure allows to consider objects both as executive units and stocking units, [Krakowiak 88]. There are two ways to describe the connection between objects and processes :

1 - to associate the executive structures to the objects,
2 - to separate objects and executive stuctures. The objects are passive and their methods are performed by processes defined independently.

We choose the second case : we consider that the process has an independent execution structure. A process is associated to each object. The object methods are called successively by the process execution structure.

A process is in connection with other processes through messages sent to the associated object which has all information needed for that task at its disposal. A method is also provided for interacting with the clock process. Moreover, each object contains the procedure for its internal computing. The object also has other private procedures and variables, not visible from outside.

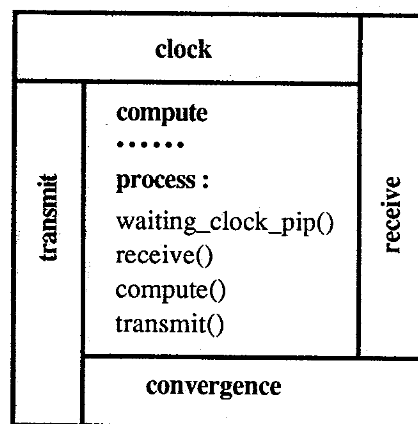

fig 3- Object/Process.

4.1. OBJECT AND PROCESS GENERATION FOR TRNSYS

The TRNSYS program is characterized by its input deck, which contains the detailed description of all the modules.

We have written a compiler for this file, with the Unix tools LEX and YACC. We now describe the successive steps allowing the creation of the processes and objects necessary for any simulation.

First step : generating an intermediate code from the input file

This step allows us to translate the simulation configuration under the form of a detailed description for each unit. This description contains for each unit :

- its type,
- its parameters and initial values,
- its inputs and initial values,
- the initial values for possible differential equations,
- for each input, the communication canal where it must be read,
- its outputs, and for each output, the communication canal where this latter must be written,
- the communication canal for clock synchronization.

Second step : creation of processes/objects

After the first step, we have all the necessary information to create all the processes/objects. Hence, we have, for each unit, the needed class and all the communication canals which will allow us to create an autonomous process.

After this second step, all the processes are created, each linked to an object.

Third step : general scheme of processes

The use of the object approach allowed us, at first, to organize appropriately and to clarify the units of our system and their communications. This approach gives us also, by the use of virtual methods, a generalization of all the processes. Therefore, the original idea which was introduced was exhibit a common execution sheme for all the processes. The final simulation is reduced to an execution of the components which are processes. Those processes have the same structure but each contains a different object.

Fourth step : clock module

A "clock" process is created to transmit the clock strokes (pips) and control the stopping of the system.

We detail only the third step

```
// third step : general scheme of a process

obj->waiting_clock_pip (pip) ; // the object waits for the
                               // clock pip

obj->init ( ) ;  // initializes the parameters and the input
                 // values while the clock pip is not a
                 // stopping order of execution

while (pip ! = END_EXEC)

DO

 // asks the object to compute, this message constitute the
 // interface with the existing Fortran  modules

 obj -> compute ( ) ;

 // for each used output, transmit the computation results

 obj -> emettre ( ) ; // this message is synchronous

 obj -> receive ( ) ; // waits all its inputs

 obj -> waiting_clock_pip (pip) ;

 done
```

Remark : for reasons of simplification, we have omitted here to mention the convergence problem. The retained execution scheme of the process is lightly modified in consequence.

4.2. PARALLELISM AND INTER-PROCESS COMMUNICATIONS

To illustrate these points, we are going to examine an example, from the TRNSYS reference manual, which shows the manner in which the different processes interact and the roles the objects play.

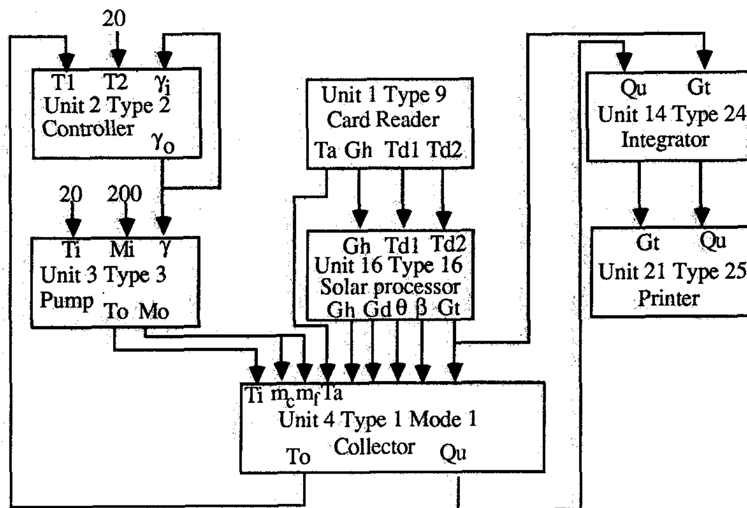

fig 4- A Simple Solar Water Heater.

From the input file corresponding to this scheme, we generate seven processes.

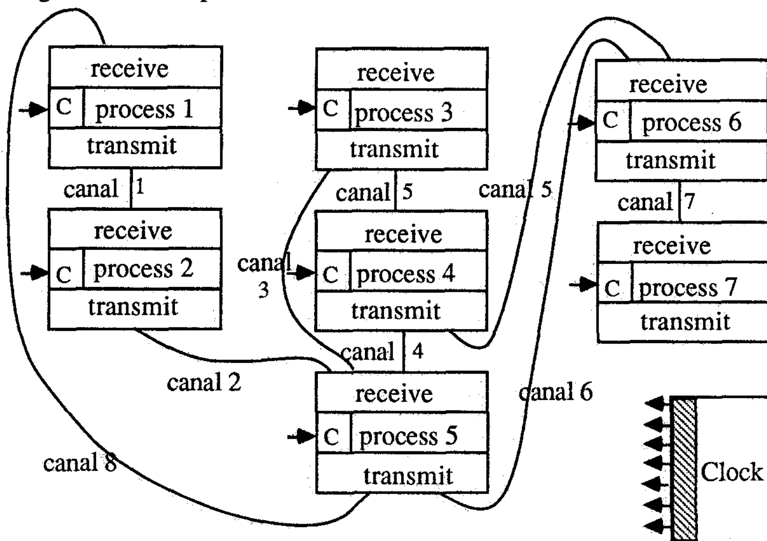

fig 5- Objects/Processes for the example.

All the processes are executed following the same scheme exhibited in the third step of the preceeding paragraph. Meanwhile, each "receive" method has different parameters which signify from which canal the data can be received and to which inputs they correspond. Equally, the "transmit" methods depend on parameters of the objects associated with the process.

To summarize, each process executes the same code, sending the messages "*transmit*", "*receive*", "*compute*", "*wait_clock_pip*" to the associated object. This latter contains its own parameters which will be used by these methods. All these processes are in direct communication with the clock process to synchronize the simulation running.

5. CONCLUSION

Our first implementation is done on an Unix system, with pipes and sockets as a mean of communication. But our aim is to implement our system on a parallel computer. In a first step, we will implement it on a hypercube computer (Intel iPSC2) [El Boussarghini 89]. All the necessary information for this implementation is, therefore, ready in the intermediate code ; we have only to specify, for each object, the identity of the processes with which it will communicate. Besides, in such an environment, we must solve the placement problem of the processes.

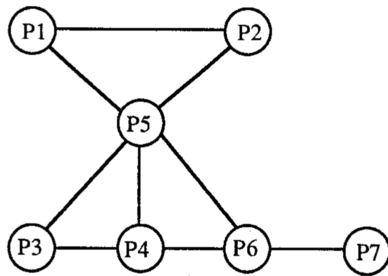

fig 6- the seven processes and their communications.

Secondly, we think of implementing it on a transputer based computer. Those two implementations will allow us to have significant measures showing the advantages of our system.

6. REFERENCES

[El Boussarghini] R. El Boussarghini. Mise en oeuvre de DATALOG sur une architecture parallèle PHD, Rennes I, September 1989

[Krakowiak 88] S. Krakowiak, M. Meysembourg, M. Riveill and C. Roisin. Modèles d'objets et langage pour la programmation d'applications réparties. *Génie logiciel & système expert*, n° 11, March 1988

[Meyer 88] B. Meyer. *Object Oriented Software Construction* Prentice -Hall, Hemel Hemptead, 1988

[Riaux 91] H. Riaux, M. Molnar, P. Boinet, J. Miriel. A graphical man machine interface for modular HVAC System simulation program, *Proc. of IBPSA*, BS'91, Nice, August 91

[TRNSYS 89] Reference manual. Solar Energy Laboratory, University of Wisconsin - Madisson