Simulation of Thermal Building Behaviour Based on an Object Oriented ADA Implementation

R. Ebert B. Peuportier G. Lefebvre

GISE (Groupe Informatique et Systèmes Energétiques)
ENPC/EMP La Courtine F-93167 Noisy le Grand Cedex
member of GER ALMETH France

Abstract

The simulation complexity of the thermal behaviour of buildings can be reduced by splitting it up as a hierarchical system of linked components. The behaviour of each component and its relations to the other components are modelled by an object oriented approach. We describe an Ada implementation of these concepts and a simple example of a multilayer wall at the end of this article.

1 Introduction

The energy crisis in the seventies and the increasing importance given to the comfort have led to investigations related to the analysis of the thermal behaviour of buildings. The most usual way to predict thermal performance is to make a *numerical simulation* of its behaviour on a computer. A lot of software of this kind has been developed during the last ten years. These programs are based on various numerical methods and physical modelling assumptions and provide results which can be different.

Until now, the general philosophy of these programs has been to gather physical data describing the building in a global model on which numerical methods are applied in order to obtain a *correct* solution; the simulation consists in calculating an approximation of the thermal state of a building at each moment of the studied period. The thermal state is generally defined as a set of temperatures at different points of the building structure, and the methods included in these programs calculate the whole temperature field at the same time. This *global* description of buildings enables the use of some very powerful numerical methods to obtain *good* solutions; but this method is rigid because it is necessary to make again all calculations when modifying a parameter value.

For the last few years, investigations have been made to improve the modelling of complex objects as a *system* (cf. IMACS 1988 proceedings[3]). We show here the possibility to build a dynamic model of a system of components which has a behaviour close to the *real* one, and which offers more possibilities to the scientist/modeller and the designer.

The building structure is described recursively as a set of components or subsystems linked by coupling conditions. This technique has many advantages:

- A system description is more *natural* than the global one. The reality is always seen as a system (a building is a set of walls, floors, ... or a set of zones, ...).
- At a macroscopic scale, it seems that not all components of a building are linked together.
- The system modelling approach is flexible because the modification of a component or a subsystem is easy; the effort made by the user to describe a complex building is saved only when variants have to be improved (what will be the behaviour of my building if I change the material of the south wall?). Libraries in which basic and/or complex models are stored can be made; modelling knowledge included in these models can be exchanged between users (usual walls, passive components, heating systems,... models).
- Not the whole thermal state of a building always has to be observed. A selective observation of some particular components or subsystems can easily be made with a system representation (What is the evolution of the mean internal air temperature? What is the flux coming from this Trombe wall?).

Some constraints must be applied to get a coherent and exact modelling process.

- The coupling conditions must be modelled in order to provide the same results as for the global calculation techniques, which are considered as our numerical reference.

- All the components of a building evolve simultaneously. They may have different time scales (a wall temperature evolves "slowly", a solar flux can vary "rapidly"), but their evolutions are parallel, and must be synchronised.

This modelling technique has been formalised, translated under an "object-oriented" form, and implemented in ADA. This modelling process is explained below, and an example is shown at the end of the paper.

In the next paragraphs we explain some of the basic concepts of our approach.

2 Concepts

2.1 Three Levels of Abstraction

For a better understanding and description of a system, we structured the data concerning the modelling of thermal systems into three levels of abstraction. They are called *real world*, *physical world*, and *logical world*.

- In the *real world* a complex object is seen as a set of elementary objects and other complex objects. An object is an entity which is easily distinguishable from its environment. For example a building contains beside other things a heating circuit as a complex object, which in turn contains a pump as an elementary object.

- In the *physical world* a physical phenomenon is associated to each elementary object of the *real world*. For instance, a wall could be modelled as "one dimensional conduction with capacity".

- Finally, the *logical world* describes representations of objects. Mostly it will be the constitutional mathematical equation, but non numerical algorithms are also possible.

The real world is the highest abstraction level where objects are handled without knowing all the details. The logical world is the most detailed one, close to its software implementation. (fig. 1)

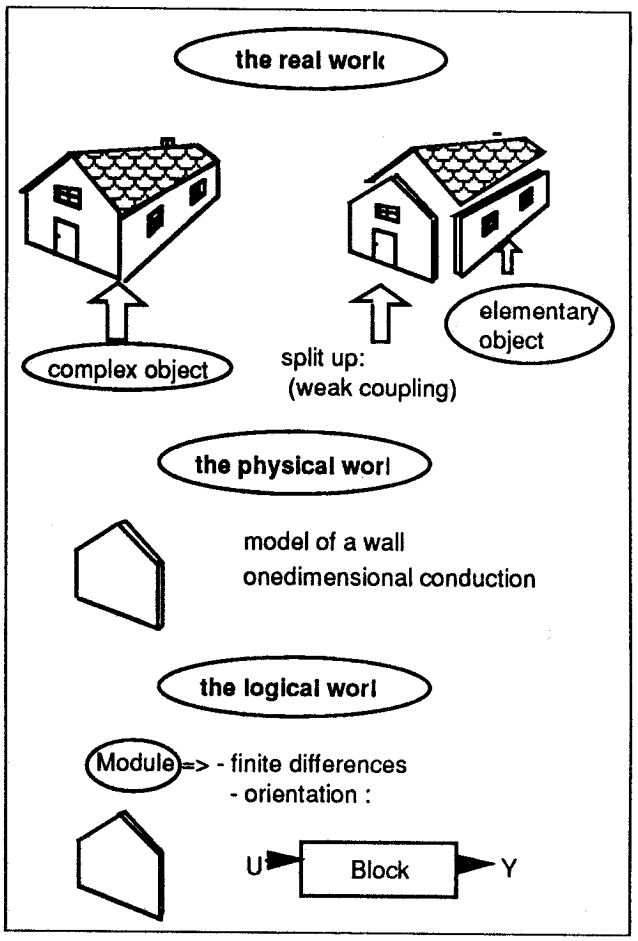

Figure 1: three levels of abstraction

2.2 Knowledge Capitalisation

The major advantage of the object-oriented modelling of buildings is *modularity*. The complexity of a model seen as a global set of data is reduced by structuring these data in a decomposition graph. The data are gathered in coherent objects; each of it can be managed without knowing the details of its contents. The objects can be cut, changed, moved, ... without having to modify the rest of a system model.

The same reasoning leads to the constitution of subsystems by recursively linking objects. This idea makes it possible to store models (objects or subsystems) in libraries. Elements of these libraries can be used to build complex models by simply coupling them. Specialists are enabled to create sophisticated models and to put them in those libraries. The user is not obliged to know the internal representation of the existing model. He only needs to know *how* to

use it. The user can access to pertinent and efficient models published in model libraries. The modularity of the object-oriented modelling, and the possibility to develop model libraries is a practical way to diffuse scientific knowledge.

2.3 Objects and Actors

The Object Oriented Programming consists of data structures organised as a set of hierarchical linked objects. In a logical object are gathered all data calculation methods, tools, corresponding to a real world object. The object can be globally submitted to various processes as storage in libraries, duplication, connection, etc. There are two kinds of objects, static and dynamic ones. They are called dynamic if they can evolve in time. Such dynamic objects are often called "actors", and are of large interest in simulation.

These actors are "living" objects. During the execution of a program they can be created, destroyed, they can receive and send messages, they can modify themselves; and all actors in a program are active at the same time. Complex systems can thus be described by a set of actors in parallel evolution.

2.4 Coupling of Objects

The division of a system into smaller subsystems raises further problems which are due to the management of the relations between the objects. (We cannot say a heating system is a set of pumps, tanks, heaters, radiators and pipes. We must describe their order and their interactions.) Cutting gives us a structure of dependencies between the components. These dependencies must be reconstituted in the simulation system in order to get it valid. When assembling the structure, we describe the neighbourhood of a module and the relations to its neighbours at the same three different levels as mentioned above.

- At the most abstract level, the *real world*, we describe the fact of neighbourhood between modules. It simply says that a component has a relation with another one. Without the knowledge about a neighbour a module stays alone and has no interaction with any other.

- The *physical world* fixes the type of relation between two linked modules. Each module has frontiers to communicate with the outside. A frontier is a set of some input/output-variables, which belong together in a logical way (e.g. temperature and flux at one side of a wall). The relation between two connected modules is described in a so called *interface*. (fig. 2)

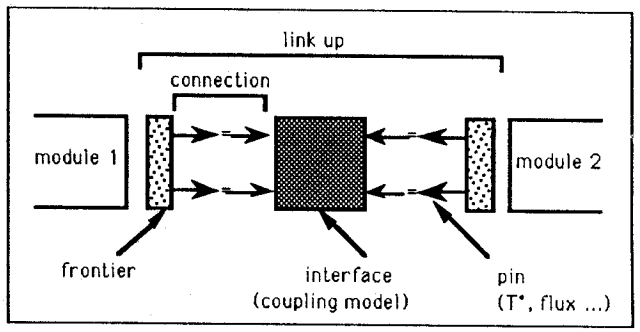

Figure 2: two modules linked up by an interface

- In the *logical world* a method to fulfill the physical constraints is described. It could be, for example, the description of an iteration algorithm to solve the coupling.

The complexity of a model can be split up and structured by a lot of different ways. Our choice has consequences on the data structure and the numerical solving method.

The main idea is that the expression of the interface model where frontiers of subsystems are coupled has to be as simple as possible. Interfaces may not be capacitive elements, because they have no material support; they then do not have *memory* and their states are the instant results of balances between the objects which are linked up by this interface.

The second idea is that objects are seen only through their frontiers. Interfaces are expressed as relations between frontiers, and do not touch the internal state of the coupled models.

These two structural hypothesis help in the modelling process by giving a framework in which the split up of a system must take place. But not all solving techniques are possible. These hypothesis are constraints that the resolution method must get over.

2.5 Resolution Methods

We suppose that a global resolution of a problem may be achieved by a set of local resolutions and relations traducing the constraints expressed on the coupling interfaces. Each interface sends constraints to the coupled objects, which intend to reach a state satisfying these constraints.

It is a natural approach because it seems close to the physical "reality". Physical phenomena are local,

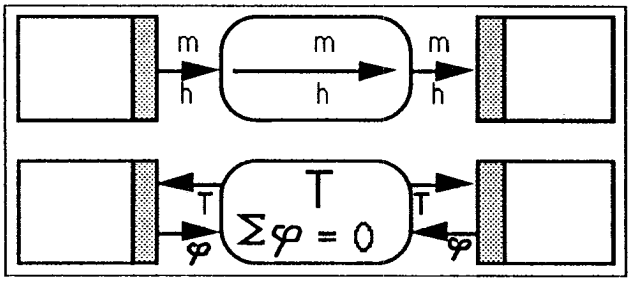

Figure 3: two implemented interfaces

they only have interactions with the elements in a (logic) neighbourhood. (For instance a wall between two rooms only sees the temperature values of the two separated rooms. It is not directly influenced by the roof temperature.) A model would have to represent only direct links (the wall with its contiguous rooms), the indirect links would be expressed during the simulation by transitive effects between neighbouring elements.

Different kinds of simulation interfaces have been implemented.

- The first is the simplest one. It only transmits the value from a sending module to one (or more) receiving modules. (The mass flow and its temperature leaving a pump can be sent to the input of a solar collector.) It is the standard interface to describe a cycle of components as it is typical for heating/air conditioning systems. This interface realises connections of outputs of some models with inputs of some others.

- The second one is used to describe a connection of modules limited by Diriclet boundary conditions on the frontiers, where temperature is fixed. A Newton iteration algorithm may be used to find the interface temperature which brings the heat flux sum of the connected modules to zero. (fig. 3)

The resolution of a system at *one* global level is a known method that works generally well. Gathering a set of local resolutions to resolve the same system raises some new problems that still have to be studied.

3 Implementation

The next paragraph describes the evolution of programming languages and clarifies our choice of using ADA for the implementation of the explained ideas. The following ones explain the practical implementation of the previous abstract concepts.

3.1 Evolution of Programming Languages

From simple "FORmulae TRANslation" the role of the computer languages evolves in order to give more possibilities of abstraction to the user. Thus the models don't have to restrict themselves anymore to numerical equations, they can be written in the form of complex structures.

Moving from FORTRAN to PASCAL we get the possibility to structure the data. Objects can be gathered into classes (**types**) and the corresponding variables may be dynamic by the use of pointers. The term "dynamic" only means that the variables can be created and destroyed during the execution. Their evolution can be simulated by the means of a procedure acting on state variables included in the structure (**record**).

A new kind of structure appeared with ADA: the **task**. Variables of a task type are called "living" or "active" objects, because they include dynamic possible actions. All tasks of a program are executed "at the same time". They can exchange parameters between each other. Like any other variable a task can be dynamic: it can be created at any time and destroyed if needed. A building can thus be described as a network of tasks, where the number of components and the general structure are not fixed. A task can create its own subtasks and can communicate with other tasks by "rendez-vous", a mechanism which synchronises the evolution of the various actors.

Another advantage of the proposed method is the possibility to describe varying components (e.g. movable insulation, electrochromic windows, ...). Also many components could be controlled (heating/cooling device, ventilation, shading device, etc. ...) and the coupling envelope/equipment can be treated precisely. At last, the interactions between the building and its environment can have their own tempo, which means different time steps corresponding to the needed degree of precision. The greater flexibility in description allows to model easily the "intelligent buildings" that are emerging, and it is in phase with the evolution in technologies concerning especially the controlled components of the envelope.

3.2 Modularity

A fundamental tool to manage the complexity of a system is modularity. It helps in two ways to man-

age a complex structure. One could be seen as "top-down". We inspect our system at different levels of abstraction. At each level we can split it up into different functional modules. On the other hand is the "bottom-up" method. Using predefined modules out of a box of bricks we can build our system beginning with elementary objects to higher and higher levels of complexity and abstraction. To accomplish this kind of work the existing modules must fulfill some constraints of similarity in order to match each other. We have to notice that a complex system is never constituted one or the other way. It is rather a mixture of both approaches.

Booch (1986)[1] gives a good definition of a module in the sense of computer languages as an object: *An object is an entity that has state, is characterised by the actions that it suffers and that it requires of other objects. Objects have two views associated with them: the outside view, which defines its interface, and the inside view, which provides its implementation.* Whereas the outside view of an object serves to express the abstract behaviour of the object, the inside view indicates how that behaviour is implemented. (Masini 1989[4]) One object can interact with another by seeing only the outside view, without knowing how the other is represented or implemented.

The computer object that we want to define, the actor, is a data structure that can evolve and that can communicate with other components. As explained above the ADA feature **task** responds well to our ideas of object oriented simulation. It has an interface part, which is called **specification** and an implementation part which is called **body**. In defining the interface we determine the possible actions of a module.

At this moment the following form of a simulation module exists; it declares two entries for initialisation and three entries for the simulation itself, termination is a buildt-in feature by the language.

```
task type Any_Module_Type is
  entry Initialise(Myself : in  Module);
  entry Initial_Frontier(T: out Message);
  entry Frontier(T : in  Message);
  entry Result(Phi : out Message);
  entry Fix_The_State;
  --entry Show;
end Any_Module_Type;
```

After the construction of a new task we have to initialise it by the call to `Init_Model`, because we cannot pass parameters during the construction itself. The argument **Module** contains the type of the module and the eventual branches of the system tree.

The first actor, which represents the simulation system in a whole, is created by the main program. Corresponding to the description tree in **Module** it recursively creates and initialises the needed branches (submodules) and the needed interface objects in its own body to connect the submodules. (see below the use of dynamic tasks.)

The entry `Initial_Frontier` responds with the values that the module supposes after the initialization at its surrounding frontiers.

The simulation itself only uses the three entries **Frontier**, **Result**, and **Fix_The_State**. **Frontier** accepts new frontier values. **Result** is the answer to the last change at a frontier, and **Fix** tells the module to calculate its new state using the last frontier values.

The optional entry **Show** can be called to write out the momentanous values of the module.

Only a small number of points of access to a module are available. All linking of modules passes through this type of interface. The internal representation of the phenomena cannot be used for any outside calculation. Thus one representation can easily be replaced by another and we get a very flexible simulation system. This is true at two levels. One is that we can change a representation for another that treats the same phenomenon in a perhaps better way. The other level is to change the system to get a better performance. (For example one could replace onedimensional conduction by twodimensional conduction in order to get more detailed results.)

3.3 Actors Implemented As Tasks

After the construction and initialisation of all needed module-tasks the simulation begins. Since a call to any of the entries can occur at any time, the three simulation entries are gathered in a common loop. Wherever a call arives, this entry will be "accepted" and worked out. Only one entry can be accepted in a time, thus we use the select statement.

When a new information arrived in the task via `Frontier`, a new state and new boundary responses are calculated.

An inside view of our simulation task that simulates a wall modelled as onedimensional conduction by finite differences.

```
accept Initialise(Myself : in Module) do
  ...
end Initialise;
  ...
```

```
loop
  select
    accept Frontier(T : in Message) do
      Current_T := T(1..In_Length);
    end Frontier;
    State := (M_Inv * Last_State) +
             ((M_Inv * Q) * Current_T);
    Current_Phi := (J * Last_State)
                 + (G * Current_T);
  or
    accept Result(Phi : out Message) do
      Phi(1..Out_Length) := Current_Phi;
    end Result;
  or
    accept Fix_The_State do
      Last_State := State;
    end Fix_The_State;
  or
    accept Show do
      ...
    end Show;
  or
     terminate;
  end select;
end loop;
```

After a call to **Frontier** the new state (**State**, the temperatures in the nodes) and the resulting heat flux at the edges (**Phi**) are calculated. These matrix calculations are performed parallelly with all other active tasks. During the rendez-vous (the synchronization of two tasks) only a result copying occurs without further calculation. Whenever The calling unit can talk to the module whenever it wants to.

Since we must distinguish between calls in the iteration loop to find the right frontier values, and the evolution in time, we use the entry **Fix_The_State** to tell the module that the next timestep has arrived. At the moment we use this simple means that forces us unfortunately to use the same timestep all over the whole system.

Another important implementation aspect is the use of dynamic variables, which are created (and possibly destroyed) at run-time. Pointer variables keep the address of such a dynamic variable. In ADA also tasks can be variables of this kind. This enables us to create and destroy active objects (actors) corresponding to the needs in the system description. A task can therefore create its own sub-tasks. This constitutes a network of tasks, of which the number and the structure are not fixed from the start.

```
type Ptr_Finite_Differences is
  new Finite_Differences_Module;
```

Tasks of type **Finite_Differences_Module** can than be created in the body of a **room** module.

```
task body Room is
  My_Walls : array (1..Max_Z)
             of Ptr_Finite_Differences;
begin
  for I in 1 .. Number_Of_Walls loop
    My_Walls(I) :=
      new Finite_Differences_Module;
  end loop;
  ...
end Room;
```

The room creates its own local walls depending on the information it had received during the initialisation.

4 Examples

4.1 A Simple Wall

In the following example we observ the evolution of the temperature field in a simple wall from the start of stresses to a stationary state with onedimensional heat transfer.

The wall is made of two layers, one is 8cm insulation and the other is 20cm concrete. The initial temperature in the whole wall is 19°C and it is stressed at both sides with Fourier boundary conditions, this means with a constant heat exchange coefficient. The boundary temperatures are 0°C at the left and 19°C at the right side (fig. 4)

Besides others we created a finite differences model that calculates onedimensional heat conduction. A model to calculate the behaviour of a constant exchange coefficient was called COEF. Thus, our wall can be split to the four terminal modules LEFT and RIGHT of the type COEF and INSUL and CONC of the former mentioned type FINITE_DIFFERENCES.

This basic structure of the terminal models can be linked in different ways. Following are three that we examined. They are called Min_L, Min_W, and Min_M (fig. 5)

In Min_L the four terminal modules are directly linked to the main module MIN_L. Therefore MIN_L contains three internal interfaces, the first between LEFT and INSUL, the second between INSUL and CONC, and the third between CONC and RIGHT.

A more "physical" way to link is used in Min_M. First INSUL and CONC are linked in the module WALL (one interface). Finally LEFT, WALL, and RIGHT are

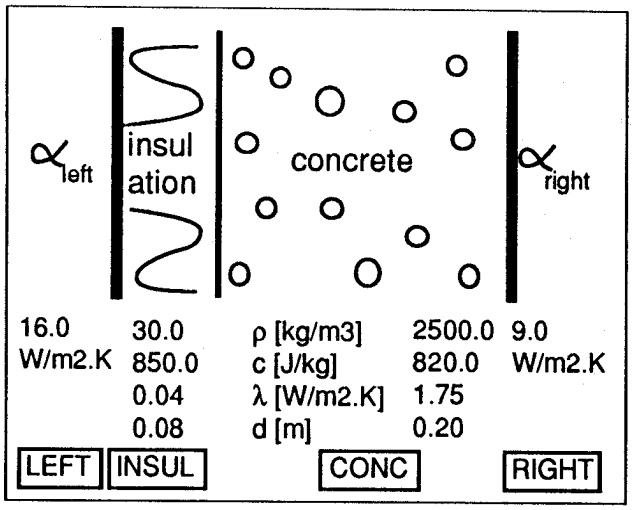

Figure 4: a wall of concrete and insulation

connected together in the module MIN_M (two interfaces).

Since the most rapid evolution is to be expected at the left side, we tried Min_W that should avoid unnecessary calls of the slower terminal models at the right side. Thus, LEFT and INSUL are linked to L_INSUL, and CONC and RIGHT to CONC_R. MIN_W is made of L_INSUL and CONC_R.

The timestep for the simulation was 60s and the three models reached the stationary state within 20.000 s.

This type of simulation seems appropriate to study the coupling between the envelope and the heating device. Actors can be variable components like a movable insulation, a glazing of variable transparency, an electrochromic coating, ... The number and the structure of the components are not fixed, thanks to the use of dynamic variables.

5 Towards Object Oriented System Simulation

Abstract data types (even implemented as tasks) are somewhat static. Once defined it is not possible to adapt to new uses except by modifying its definition. We consider it useful that one can keep an once defined type (and its set of corresponding operations) and declare a new type as function of the first one. The new type inherits all the properties of his "parent", but it can be extended or constrained to reach a specific objective.

This important concept of object oriented languages, the inheritance, does not exist in ADA. But it would be possible to develop an extension of the language, as C++ was derived from C. (Forestier 1989[2]) That is why we will try to develop an abstract description of models and their relations using the ideas of object oriented analysis and object oriented programming. This abstract description of a system can then be translated into a corresponding computer code.

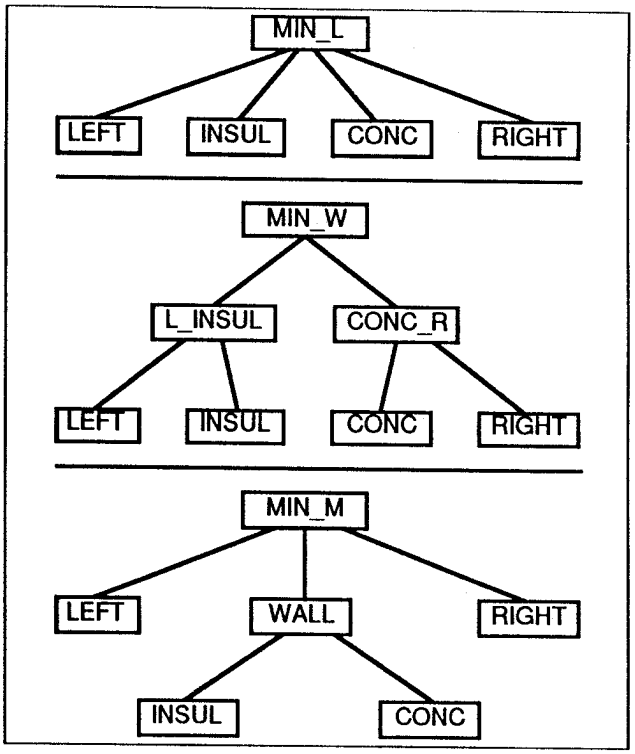

Figure 5: three possible ways to split up the wall

References

[1] Booch G. *Software Components with Ada.* Benjamin/Cummings, Menlo Park, CA, 2nd edition, 1986.

[2] Forestier J.P. et al. *ADA++, A Class and Inheritance Extension for Ada.* Proceedings of Ada-Europe, Cambridge University Press, 1989.

[3] IMACS 1988. *IMACS 1988, 12th World Congress On Scientific Computation.* Paris, 1988.

[4] Masini G. et alt. *Les Langages à Objets, Langages de classes, langages de frames, langages d'acteurs.* InterEditions, 1989.