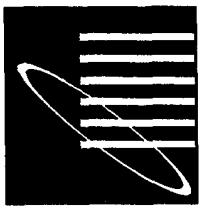

Application of the Object Oriented Programming Paradigm to Building Plant System Modelling

Dr D Tang
Department of Civil Engineering
Surveying & Building
Dundee Institute of Technology

Professor J A Clarke
Energy Simulation Research Unit
Department of Mechanical Engineering
University of Strathclyde

Abstract

The object oriented (00) approach to software engineering offers new possibilities for the modelling of plant systems within a building context. This paper describes the development of the EKS plant modelling classes which facilitates the automatic creation and maintenance of plant system simulation models. The paper focuses on the following aspects:

- *A generic software structure which allows the creation of a wide range of state-of-the-art plant/ control system models;*
- *A dynamic model building structure which allows a simulation program to be built with minimum number of component modules required and therefore eliminates the burden of redundant components.*

Introduction

One of the principal tasks in environmental studies concerns the assessment of the environmental performance of buildings. Development of computer applications in this field has undergone considerable change over the last two decades, resulting in software systems for research and design practice of increasing accuracy and efficiency.

The prediction of the energy performance of the building involves two major aspects: the performance of the enclosure and the performance of the plant systems.

Generally speaking, the building enclosure is a system of large thermal inertial which acts to filter the external and internal excitations. Mathematically such system is usually represented by a large, sparse matrix equation with predefined topological structure. This is mainly due to its governing domain theory of combined problems of thermal diffusion and aero-dynamic systems. It is clear that the fundamental elements comprising the mathematical model of the building, i.e. walls and spaces, are permanently in existence. The presence of other elements participating in the system, e.g. an occupant gain or the variation of solar gain at an external wall, changes only the magnitude of certain parameters contributed to the boundary conditions; the topological structure of the system remaining the same. As a result, the topological model structures of the building enclosures are usually prestructured as part of the simulation program and the solution of the system mathematical model can be optimised.

For plant systems, the components used will vary from system to system. The connections which couple the components within the system may also be different even when the same components are used. This requires an arbitrary selection of components and their connections to allow the definition of run-time system topology. To this end, most plant simulation models adopt a modular approach wherein each component

comprising the system can be considered as an internally pre-defined model while the complete system topology is only known when the components and connections are selected and defined at run-time. To satisfy various requirements, a plant system program has to accommodate a large number of component modules. As the result, many components modules of the program will not be used for a particular simulation. However, these redundant components (usually in the form of subroutines) remain resident in memory but idle throughout the simulation.

At the present time plant system modelling has evolved along two distinctive paths in terms of system architecture: sequential, eg TRNSYS (1978) and simultaneous, eg ESP-r (1993). In a sequential approach each component provides a pair of input/ output vectors for mutual communication while the system coordinates component execution and network data flow. Iterative solution techniques and other devices are usually employed to achieve solution speed and improve accuracy. In a simultaneous approach each component model provides independently a set of governing equations. The system at higher level coordinates the creation of system equation set by providing network dependent couplings for solution by mathematical techniques such as matrix partition and ordering. Figure 1 shows the model architectures for the simulation of a simple air-conditioning problem.

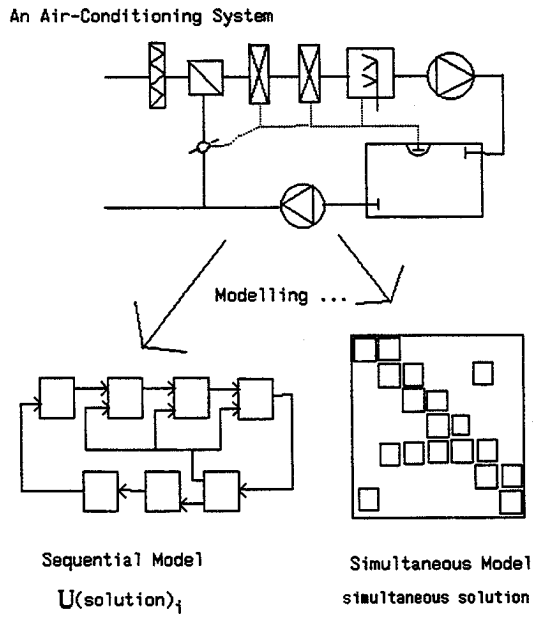

Figure 1. System model architectures

The Object-Oriented implementation

The advances in OO techniques is offering new opportunities for system modelling within a building context. A number of pioneering systems have emerged in recent years, including the US system SPANK (Sowell, 1991); the Swedish system MODSIM (Sahlin, 1988); and the UK Energy Kernel System (EKS) (Charlesworth et al 1991).

Each system has tended to concentrate on a particular issue. SPANK, for example, was developed primarily for plant and control simulation and concentrates on a novel solver which optimises the solution sequences to yield accurate and efficient solutions; MODSIM offers a generalised solver for plant and control problems, removing input-output connotations from the problem;(Wright et al, 1992) While the EKS has concentrated on the development of an OO platform for the creation, maintenance and validation of simulation models in general. (Clarke et al, 1992)

Within the EKS project an attempt was made to provide and OO-based, modular program building environment. With respect to plant and control systems, the EKS offers:

- A generic software structure which allows the creation of plant simulation programs of either simultaneous or sequential type;
- The elimination of the burden of redundant components to allow a simulation program to be built from only the components required by the problem.

These plant modelling facilities are built upon the various utility classes provided within the EKS environment. These include computational support classes which encapsulate a spectrum of analytical, numerical or statistical solution methods; intrinsic classes for data transport; generic list and network classes for dynamic data structures; and meta-classes and template classes for automatic class creation and manipulation. (These class types are explained in the previous reference). The development of the EKS plant classes involved three principal steps:

- Knowledge identification and representation
- Data encapsulation
- Automatic model construction

Knowledge Representation

The essence of the OO paradigm is the view that a program can be composed of independent objects communicating via messages. To achieve this the physical system has to be decomposed into objects which represent the knowledge of the physical world in the OO design space. It should be appreciated that in the OO field there is no commonly accepted approach to this task. For this reason, the EKS has adopted the methodology of functional/data decomposition and the result of this is the EKS taxonomy which represents the encapsulated of physical entities in the three dimensional OO design space. Figure 2 shows conceptually the 3-D distribution of classes of the EKS.

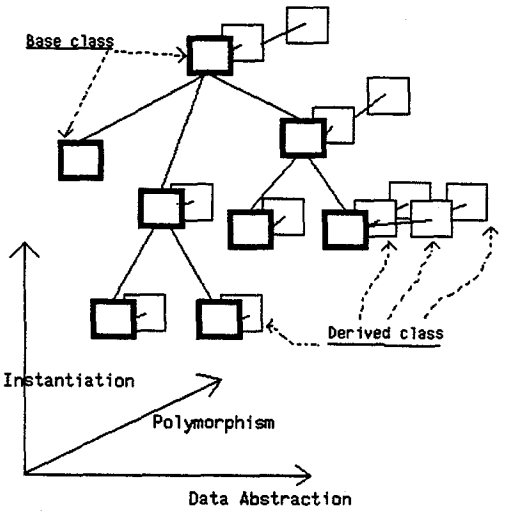

Figure 2. The OOP design space of the EKS

Building Representation

The initial development of the EKS started from an exploration of the domain of building thermodynamics by which the physical integrity of the building is sub-divided into fundamental entities - classes. Each class, either alone or in conjunction with a few support classes, handles one particular aspect of the building performance prediction process. The classes thus identified represent not only the physical entities comprising the building but also the alternative theories (mathematical methods for solving a particular problem) to which the simulation models are categorised. Therefore a wide range of modelling programs may be built. As dictated by the topological structure of the system, the instantiation of different classes are pre-defined by using the 'used by' controls.

Plant Representation

For the development of functional/data decomposition of the plant system within the EKS, two approaches were envisaged:

The Atomic Approach

In this approach a number of fundamental entities (e.g. air, liquid, solid, etc.) are identified in much the same way as can be applied to the building-side. These entities can then be used to construct any component in general. By doing so, it is not possible to apply instantiation control over the atomic entities as the structure of the components are radically different. This implies that the system should be able to handle two levels of arbitrary system topologies, i.e. the intra-component topology which is created within the component and the inter-component topology which is created dynamically at run-time. Figure 3 shows the concept of the atomic approach.

The Component Based Approach

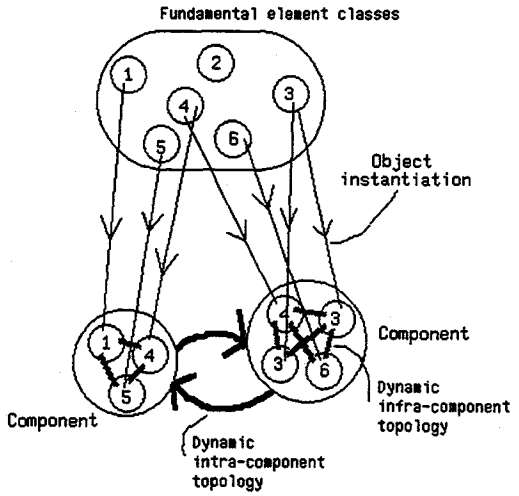

Figure 3. The atomic approach to plant modelling

In contrast to the atomic approach, the component based approach treats each component as a topologically pre-defined equation structure and therefore the mathematical complexity is hidden away within each individual component. The system is responsible for the creation and maintenance of the dynamic inter-component topology. Figure 4 shows the methods represent by this approach.

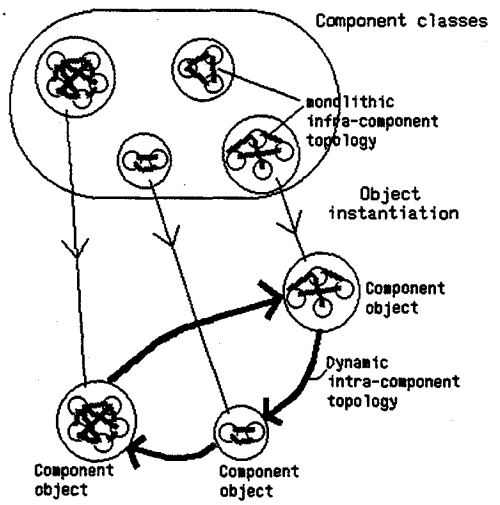

Figure 4. The component based approach to plant modelling

The component based approach was adopted initially for the plant modelling based on the consideration that the EKS aims to support the creation of state-of-the-art simulation models e.g. the sequential and simultaneous architectures. Program codes available from existing systems, e.g. TRNSYS and ESP-r, can be directly imported for the implementation of component classes. The concept of an atomic approach, on the other hand, can be viewed as a further step forward towards the exploration of the OO paradigm. However, at present it still remains as an open research issue.

To further develop the representation scheme for plant system modelling, a top-down approach was adopted with the OO inheritance

mechanism used for the following reseasons:

- To enable code sharing and data security. This means that new, derived classes need only add the code for any extra functionality they provide.
- To provide common interfaces for the implementation of alternative modelling approaches.

Based on these, the following requirements for the class hierachy were identified:

Top level

Data and functions for the representation and maintenance of the network topology are needed so that the inter-component relations are handled in a way which is independent of the modelling approach.

Level 2

Parallel inheritance of the sequential and simultaneous approaches to accommodate the fundamental difference in system architectures. While both implementations inherit the same generic system topology presented in the top level, in this level, methods for the creation of the plant system infrastructures are implemented. Here, the component is treated as 'virtual' to provide generic interfaces to allow the alternative selection of components as required.

Level 3

Reimplementation of the generic component to represent the actual component modules. Each module follows the same interface provided by the generic component at Level 2.

Lower levels

Further lower levels may be derived to allow code sharing based on the same type of component at level 3 but with increasing complexity.

Figure 5 summarises the classes identified at the present time and the associated inheritance paths. Here, the instantiation controls are shown only with respect to the base classes. It should be equally applied to the class cluster at each corresponding level.

Data Encapsulation

Classes were created for the encapsulation of data and behaviour based on the class hierachy shown in Figure 5.

Two alternative implementations are provided. In the first, the creation of model architectures are facilitated , i.e. the sequential and simultaneous approaches. In the second, the arbitrary incorporation of component modules can be built into the target program.

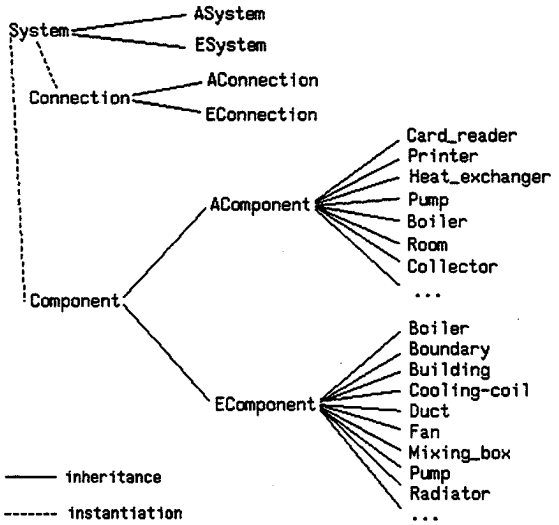

Figure 5. Inheritance of the plant classes

Implementation is achieved by applying the technique of explicit polymorphism, which in C++ is implemented using the constructs of derived class and virtual function (Stroustrup 1987). In the approach, the base class of the component holds a virtual function table built into its private data, with the entry of the function pointer resolved at run time. C++ guarantees that the virtual function entry of the base class can be substituted by any implementation of the function in the derived classes. Therefore, at program construction time, any type of component object can be coupled to the dynamic list held as part of the program. At run-time, these components can be instantiated or duplicated and connected arbitrarily to the system network as defined by the program run-time topology.

To implement the alternative model architectures, two sets of classes are derived in parallel from the generic base class as follows:

Sequential model	Simultaneous model
ASystem	ESystem
AConnection	EConnection
AComponent	EComponent

As an example, the following code shows the implementation of the generic base classes of System, Component and Connection.

```
class System : public EKSObject {
protected:
   Network(Connection,Component) system;
   Component*        findnode(char* component_name);
public:
   System(Metaclass* meta, System_def* def);
   ~System();
   List(Component)&  components();
   List(Connection)& connections();
   List(Connection)& in_connection_for(Component* cmp);
   List(Connection)& out_connection_for(Component* cmp);
   List(Component)&  up_stream_node_for(Component* cmp);
   List(Component)&  down_stream_node_for(Component* cmp);
   Matrix&           connectivity();
   Matrix&           adjacency();
   virtual void      execute();
```

```
};

class Component : public EKSObject {
public:
   Component(Metaclass* meta, Component_def* def);
   ~Component();
};

class Connection : public EKSObject {
private:
friend  class System;
   Component*   the_source;
   Component*   the_target;
   void         assign_source(Component* src);
   void         assign_target(Component* tgt);
protected:
   char*        the_source_name;
   char*        the_target_name;
public:
   Connection(Metaclass* meta, Connection_def* def);
   ~Connection();
   Connection& operator=(Connection& conns);
   Component*  source() { return the_source; };
   Component*  target() { return the_target; };
   char*       source_name() { return the_source_name; };
   char*       target_name() { return the_target_name; };
   char*       my_name() { return name(); };
};
```

Here, the generic base class System implements the system topology by creating an object of Network class which accommodates the actual components and connections to be created at program run-time.

To further implement the functionality of arbitrary component module selection, two sets of classes are created for the implementation of the actual components each of which is derived from its parent class, i.e. AComponent and EComponent respectively.

System Architecture

In a sequential system, each Component possesses a pair of heterogeneous vectors as the i/o interface. The Connection gets data from the source component and sends it to the target. The solution sequence of the system is based on an over-relaxation scheme by which the system executes the connection stack of the network while each connection object activates the source and target components. This is done for the following reasons:

- To ensure that only those components which are connected will be executed.
- To ensure that the target component always receives the latest information.

At the run-time, after the network is defined, an optimisation algorithm is used to re-order the connection list into an optimum solution sequence. The optimisation is based on two concepts: that the solution follows the system flow sequence; and that the latest data are always provided to the target component.

In the simultaneous approach, each component is represented by a set of equations with potential nodes pointing outwards to its boundary to be resolved by the Connection class at run-time. The construction of the complete system matrix equation is therefore connection based. The system executes the connection stack of the network to allocate the source and target component matrices as diagonal blocks in the system matrix and creates the coupling sub-matrices between each pair of diagonal blocks. At the same time a pointer reference component list is created to ensure that no component matrix is duplicated. This treatment also ensures that at the run-time only those components connected in the system network will be allocated into the diagonal block of the system matrix.

After completion of the system matrix, an optimisation algorithm can be used by creating the appropriate solver objects.

In both approaches, the actual components are implemented as classes derived from the generic components, i.e. class AComponent and EComponent respectively. The inheritance mechanism ensures that the actual components can be substituted into the entries created initially for the 'virtual' component objects thereby allowing an arbitrary selection of component modules to be built into the system model.

Automatic Model Construction

The configuration of the system architecture is supported by the problem's context class which is equivalent to the creation of a main program in conventional programming.

The following code is an examples of three problem context classes. Here, class Context is the EKS principal class which coordinates the whole taxonomy; Context_base_plant coordinates the creation of the system network topology; Context_a_plant and Context_e_plant coordinate the creation of the different modelling approaches.

```
class Context_base_plant : public Context {
protected:
      EKSObject*      the_system;
public:
      Context_base_plant(Metaclass* meta, Context_base_plant_def* def);
      ~Context_base_plant();
      System* get_system();
virtual void    simulate();
static  void    description(Metaclass* meta, ostream& s);
};

class Context_a_plant : public Context_base_plant {
public:
      Context_a_plant(Metaclass* meta, Context_a_plant_def* def);
      ~Context_a_plant();
      ESystem*        get_system();
virtual void    simulate();
static  void    description(Metaclass* meta, ostream& s);
};

class Context_e_plant : public Context_base_plant {
protected:
      Solver*      the_solver;
public:
      Context_e_plant(Metaclass* meta, Context_e_plant_def* def);
      ~Context_e_plant();
```

```
    ASystem*        get_system();
virtual void    simulate();
static  void    description(Metaclass* meta, ostream& s);
};
```

The automatic construction of the simulation program is supported by the utility programs provided by the EKS:

EKS_cb: the context builder which allows the creation of a user-specified context.

EKS_tb: the template builder which allows the selection of class variants.

EKS_mb: the model builder which supports the program building process.

The execution of the programs to result are similar to the conventional simulation programs by which any component class can be instantiated to create multiple objects of the same type and connected to other component objects.

To demonstrate plant model construction, consider Figure 6 which shows a heating system.

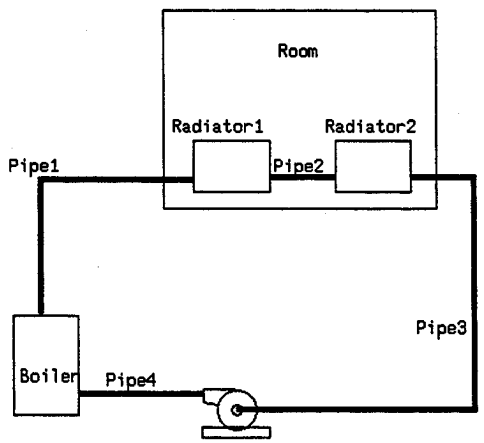

Figure 6. The heating plant system

Two system templates are created for the sequential and simultaneous system architectures with selected component classes built into the templates.

Table 1 shows the template created for the sequential plant program.

Base class	Class offered	Class selected
Context	Context	Context_a_plant
System	ASystem	ASystem
Component	AComponent	ACard_reader
Component	AComponent	ABoiler
Component	AComponent	ARadiator
Component	AComponent	APump
Component	AComponent	ARoom
Component	AComponent	APipe
Component	AComponent	AHeat_exchanger
Component	AComponent	APrinter
Connection	AConnection	AConnection

Table 1. Template for Sequential plant system

Table 2 shows the template created for the simultaneous plant program.

Base class	Class offered	Class selected
Context	Context	Context_e_plant
System	ESystem	ESystem
Component	EComponent	EBuilding
Component	EComponent	ERadiator
Component	EComponent	EPipe
Component	EComponent	EPump
Component	EComponent	EBoundary
Connection	EConnection	EConnection
Solver	Solver	Gauss_column_pivot

Table 2. Template for simultaneous plant system

Figure 7 shows the run-time model created by the instantiation of the component and connection classes defined by the template of Table 1. The execution of the program is similar to the sequential/ iterative solution scheme found in the TRNSYS program.

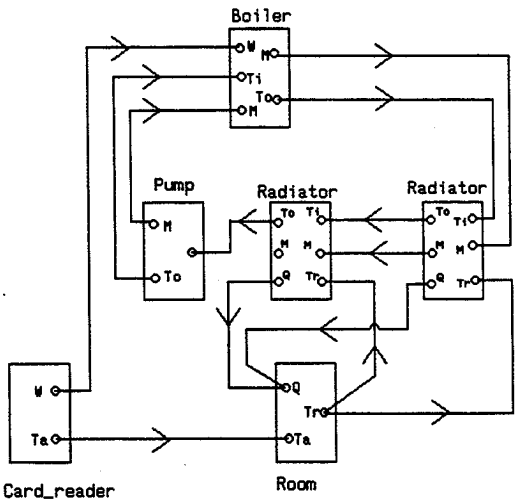

Figure 7. The sequential plant system

Figure 8 shows the run-time model, in terms of the underlying system matrix, created by the instantiation of component and connection classes as defined in the template of Table 2. The result is similar to the mathematical model structure of the ESP-r program.

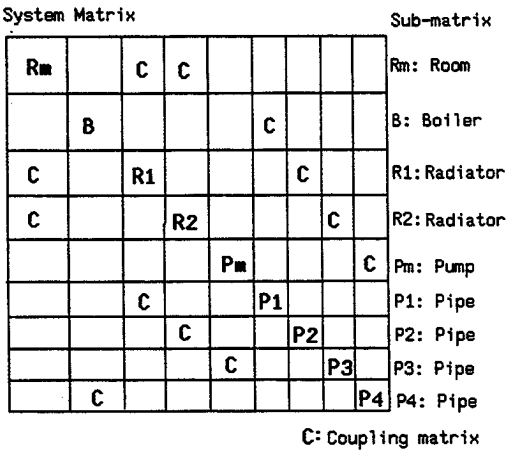

Figure 8. The simultaneous plant system

Conclusions

The development of the plant modelling facility within the EKS is an attempt to demonstrate the capability of the OO approach to support the creation of simulation models based on different system architectures. The classes as offered by the EKS provide an infrastructure for the construction of plant simulation models while, for efficiency reasons, minimising the number of component modules which need to be built into the end program.

References

Charlesworth, P.; et al, 1991. "The Energy Kernel System." In *Proceeding of IBPSA'91*, (Nice, France, August), 313-322.

Clarke, J. 1985. *Energy Simulation in Building Design.* Adam Hilger Ltd., Bristol.

Clarke, J.; et al, 1992. "The Energy Kernel System - Final Report to the SERC." ESRU, Dept of Mechanical Engineering, University of Strathclyde, November.

ESP-r 1993. *ESP-r: Version 8 Series User Manual.* ESRU, Dept of Mechanical Engineering, University of Strathclyde

Sahlin, P. 1988. "MODSIM - A Program for Dynamic Modelling and Simulation of Continuous Systems." The Swedish Institute of Mathematics.

Sowell E. F. 1991. "Next Generation Building Services Engineering Software: Opportunities for the Practitioner" In *Proceedings of BEP'91,* (Canterbury, UK, April), 1-9.

TRNSYS 1983. *TRNSYS - A Transient System Simulation Program,* Solar Energy Laboratory, University of Wisconsin-Madison.

Wright A J, et al, 1992. "EKS Project Final Report" Science and Engineering Research, Council, Swindon.