Energy Simulation Research Unit

ESRU

The Energy Kernel System
Final Report

for Grant GR/F/07880

Submitted to SERC

Professor J A Clarke
Dr D Tang
Dr K James

Energy Simulation Research Unit
Department of Mechanical Engineering
University of Strathclyde

D F Mac Randal

Informatics Department
Rutherford Appleton Laboratory

November 1992

Energy Simulation Research Unit
Division of Thermo-Fluid
and Environmental Engineering
Department of Mechanical Engineering
75 Montrose Street
Glasgow G1 1XJ

Email esru@uk.ac.strathclyde
Phone +44 41 552 4400 X3024
Fax +44 41 552 8513

Table of Contents

Project History and Acknowledgements

Between 1979 and 1985, the UK Science and Engineering Research Council (SERC) sponsored work on environmental prediction modelling which highlighted the need for greater flexibility in model structures. In 1985, a number of European and North American research organisations considered the future of building energy modelling and its relationship to computer-aided building design. A degree of consensus emerged on the limitations of contemporary models and the features which would be desirable in the next generation.

The UK response was the notion of an advanced machine environment (Clarke et al 1988; Clarke 1988) which would foster the collaborative development of the next generation of performance assessment programs. Such a system would form the nucleus, or kernel, of future model building activities, permitting the rapid prototyping of any program architecture and facilitating a coherent approach to development, validation and maintenance. This system subsequently became known as the Energy Kernel System or EKS.

The EKS project is based on the 1987 report to the SERC (then) Building Subcommittee (Clarke 1987). This envisaged a 2 phase project: a 3 year concept demonstration phase to be followed by a 2 year system refinement/ delivery phase. The first phase was further partitioned into 5 research topics (Figure 1) four of which were funded:

Topic	Grant Holder
Method Production/ Encapsulation	University of Strathclyde
Software Engineering Infrastructure	Rutherford Appleton Laboratory
Automated Program Construction	University of Newcastle
Class/ Program Validation	University of Bath & Rutherford Appleton Laboratory

The project was undertaken as a collaboration between four institutions: the Universities of Bath, Newcastle and Strathclyde, and the Rutherford Appleton Laboratory - a grouping which operated most successful in terms of its complementary talents and viewpoints. Because the methods production/ encapsulation and software engineering aspects of the project were inextricable throughout the project, this final report has been co-authored by the researchers at the University of Strathclyde and the Rutherford Appleton Laboratory who worked on these aspects. It should be read in conjunction with the other project reports previously submitted (Wright el al 1992 and Hammond et al 1992).

Throughout the project, a SERC-appointed steering group was active to oversee developments and review progress. There was also a technical exchange with institutions in the UK and elsewhere and, in particular, with the US Lawrence Berkeley Laboratory through Ed Sowell's participation in the project as a SERC Visiting Fellow. To our colleagues in the EKS team and all others involved in the project, we extend our thanks for their technical inputs and encouragement.

Executive Summary

The project set out to explore the feasibility of developing an advanced program building environment termed the Energy Kernel System (EKS). The objectives were to:

- Identify the computational methods underlying building energy/ environmental prediction models.
- Develop a procedure for establishing these methods in an organised form within a demonstration system.
- Research the feasibility of adopting the object-orientated (OO) programming paradigm in the representation of the methods and their underlying data structures.

- Identify suitable theories and computation methods for inclusion within the EKS demonstrator.

Having become familiar with OO technology - in terms of an OO language (C++) and OO database (ONTOS) - it proved possible to construct a demonstration taxonomy of classes from which models of different functionality can be built. The role of these classes is to represent the physical entities which comprise a building (rooms, walls, etc) and the abstract entities which dictate its thermodynamic state (heat transfer theories, numerical methods, etc). These classes are organised into 'used by' and 'derived from' hierarchies and placed under the control of an instantiation mechanism. This means that programs possessing different modelling capabilities can be constructed automatically by merely selecting the required class variants - that is no user coding is required. And because each physical entity within a modelled building has a matched object at run-time, an EKS-produced program can be matched to the physical system it is being used to model. In particular the EKS class taxonomy has been progressed to a stage where it can support the construction of demonstration programs which exhibit near state-of-the-art characteristics. These classes have been placed under the control of an OO database to facilitate added security of use and support object persistence. To demonstrate the methodology of the EKS Demonstrator a number of example programs - of increasing complexity - have been constructed. These range from simple sun tracking, through inclined surface solar irradiance to a dynamic building and plant program.

The principal achievement of the project is that a complex engineering domain has, for the first time, been decomposed and re-expressed using the OO programming paradigm. This has lead to a better understanding of the role of the new OO technologies (languages and databases) in the development of more powerful design tools. In particular the EKS points to a future where:

- Design tool evolution is undertaken on a task sharing basis because different researchers can contribute new classes or modify existing ones.

- The program construction and maintenance process is more efficient because of the high level of code reuse.

- The validation process is enabled because individual classes can be tested in isolation, the meaningful connection of objects can be guaranteed and data encapsulation prevents illegal data interference.

- Design tools will possess greater realism vis-a-vis the reality while being easier to maintain and evolve because of the encapsulation and hiding on the underlying complexity.

In the short term it is anticipated that the EKS will be explored by researchers who are concerned with advancing the state-of-the-art in modelling buildings and the environmental control systems they contain. Already several research organisations have expressed an interest in acquiring the EKS during its proving phase. These include the UK Building Research Establishment, the University of Eindhoven, the Technical University in Delft, the University of Wellington, the Korea Institute for Energy Research, the University of Erlangen and Electricite de France. In the medium term it is possible that software vendors might use the EKS to construct and maintain the theoretical 'engine' of future design support systems (the Committee will recall that Intergraph were one of the original backers of the EKS project). In the longer term it is entirely feasible that end users such as Architects, Engineers and Energy Managers could use the EKS to achieve bespoke software solutions for particular problems (the Committee will recall that British Gas and Ove Arup also endorsed the project).

Should a resource be found to continue the development of the EKS Demonstrator into a robust product, then it should provide a program building platform which ensures that new techniques and theories in heat transfer, validation and numerical methods will become more immediately available to the community of potential users. Model developers - CAD vendors, research organisations and ultimately, perhaps, design practice and legislative bodies - can then select and combine these methods to produce an application program of particular architecture. Because the methods are established as independent, fully documented and tested entities, the program validation and accreditation process, at component and whole-model levels, is

greatly assisted.

Background and Objectives

Building designers have traditionally employed a wide range of methods to ensure that the performance characteristics of a building will be acceptable. Simple calculations and rules of thumb are applied throughout the design stage in an attempt to minimise heat loss, maximise the utilisation of solar energy, prevent inefficient plant operation, ensure high comfort satisfaction, control problematic air movement and so on.

In recent years researchers have stressed the futility of attempting to optimise, in such a piecemeal manner, a system which is inherently dynamic in that many parameters change over time and at different rates; inherently non-linear in that some parameters depend on the system state which, in turn, can only be assessed if these parameters are known; and inherently systemic in that the different heat transfer mechanisms interact in a complex manner. In an attempt to address this complexity and provide effective design decision support, the subject of building simulation has received growing attention in recent years - particularly the integrity of the underlying mathematical models and their validation. One issue that remained to be tackled was the means by which simulation systems can efficiently be built and adapted in response to new theoretical advances, changing user needs, deficiencies exposed through use and changes in the underlying IT.

The EKS project is an attempt to place on a rational basis the construction and accreditation[#] of advanced design tools for the building industry. It does this by providing a program construction and maintenance platform which dispenses with the need to work with source code. The expectation is that this platform will dramatically improve researcher productivity and serve to enable task sharing evolution of future, simulation-based design tools. In the longer term practitioners themselves will be able to use the EKS to create bespoke tools to handle particular design problems. **It is stressed that the EKS is not a new energy program but a program building environment.**

Essentially the EKS is an enabling technology for the disparate range of activities now underway in the field of advanced building modelling - the many theoretical developments; algorithmic and whole model validation; product modelling; and design integration.

The objectives and premises of the EKS are:

Objective	Premise
To separate out the calculation methods, data structure and model architecture elements of future design tool construction.	The developments in each of these areas will be more easily integrated and future design tools will be better structured.
To simplify the program building process.	This will ensure that future design tools are more robust and easier to maintain.
To establish validation within the model construction process.	The validation component of design tool accreditation will be better served.

The issue of program construction techniques is the subject of several large scale projects in Europe (MODSIM) and North America (SPANK), though these are more limited in scope than the EKS. The program accreditation issue on the other hand can be recognised only implicitly in projects with other, more pressing goals (for example, the development of simulation based standards within the European standards organisation CEN).

Objective	Premise
To promote state-of-the-art developments through ease of integration of new methods as they emerge and to encourage the mixing of simulation and other engineering applications software.	This will ensure that design tools evolve in tandem with theoretical and interface advances and not, as at present, with a considerable lag.
To enable and encourage interdisciplinary collaboration between model developers, and between developers and end-users.	The quality of future design tools will improve markedly if participatory development is enabled.
And to remove the burden of machine portability and other hardware/ software problems from the model builder.	Application experts and practitioners will become more productive if the machine aspect is removed from the design tool construction equation.

EKS Demonstrator Overview

The Energy Kernel System is an objected-oriented platform for simulation model creation, maintenance and validation. Essentially it contains a set of class definitions corresponding to the building and thermodynamic domains. These classes can be considered as the basic building blocks from which a wide range of modelling programs may be built - from simplified performance assessors to state-of-the-art simulators. Each class, either alone or in conjunction with a few support classes, handles one particular aspect of the building performance prediction process. (See a later section for a description of the class types available within the EKS Demonstrator.)

Classes are organised into a ‘taxonomy’ as shown in Figure 2. This taxonomy specifies how the classes interrelate and defines the information flow between them. The ever present dilemma between extensibility - that is the ability of any existing class to use or be used even by a newly created class - and security - that is the guarantee that the given classes are compatible - is solved by using special "Metaclass"[#] classes. A "Metaclass" defines the behaviour of its associated class, including which other classes are required for its correct operation. Thus, by insisting that programs can only be built by using "Metaclass" classes, it is possible to have an extensible but secure system without paying the performance penalty of runtime type checking.

Actual program building is carried out by the "Template" class. Given the chosen program architecture, as defined by a particular "Context" class, the "Template" builds up a collection of "Metaclass" instances which define the program structure. These "Metaclass" objects can then check that the specified program is internally consistent. Furthermore, since the "Template" is in effect a specification for the program, there is no need to generate a executable program at this stage. Instead, once the problem specific data is available, the "Template" can use its "Metaclass" classes to create a customised program tailored to the problem being addressed. This is all achieved internally in the "Template" class; the program builder or user need know nothing about the "Metaclass" scheme.

In its present form (see Figure 3) the core of the system is an Object Oriented Database (OODB). This plays 3 separate roles. Firstly, it can hold entities such as climatic data and material properties encapsulated as persistent objects[*]. Secondly, it can hold the problem description and results as objects, enabling the interfaces to be separated from the body of the performance prediction engine. Thirdly, it holds the "Template" and "Metaclass" objects used internally by the EKS environment. The chosen language for the EKS demonstrator was C++, primarily because of its widespread acceptance and partly for efficiency reasons. However, one serious drawback is the lack of an in-built mechanism for examining or manipulating classes

Throughout this report names quoted thus "" signify classes.

at runtime. This feature was provided by the OODB as part of its schema definition facilities.

Recognising that reliance on an OODB platform could restrict EKS exploitation, the system was configured so that it is possible to run the system with or without the OODB. In the latter mode some of the program composition checking features are disabled and the absence of the OODB necessitates a system rebuild when a new class is added (although it is still significantly easier than the equivalent work using conventional program development techniques).

The EKS OODB (ONTOS 1989) has knowledge of the interface specification of the various EKS classes, which can be used to construct a wide range of program architectures. These classes are an encapsulation of an entity's representation (in the form of data) and behaviour (in the form of functions). As shown in Figure 3, surrounding the OODB are a number of utility modules which are used to specify the architecture of a particular program. Assuming that the user has a well formed modelling hypothesis, EKS operation will entails the use of the following programs:

EKS_cb: a program which allows the definition of a program's context (this is equivalent to the creation of a main program in conventional programming). For example it may be that one program might offer site and building modelling capabilities while another might offer a site, multiple buildings and plant capabilities. Within the EKS two different "Context" classes would be required although the latter could be derived from the former. The output from this module is a "Context" class. (The EKS Demonstrator comes complete with several example "Context" classes to demonstrate the program specification process - see ~eks/demo.)

EKS_tb: Given a "Context", this module allows the user to specify the precise capabilities of a program by selecting EKS class variants as required. As each class is selected the "Metaclass" mechanism determines the dependent classes so that the process is automated. This allows program building to be carried out incrementally and with no need for specialist knowledge in terms of the underlying algorithms. The output from this stage is a program "Template" which defines the classes and class connections from which the program will be constructed. A "Template" can either be stored in the OODB for later recall or held as an ASCII file.

EKS_dd: This module takes a "Template" as input and outputs the corresponding OO product model. If the data of this product model are unacceptable, in that they cannot be obtained from the intended end user type, then the previous applications can be revisited and the program architecture modified. At the end of this iterative process, the data requirements would typically be made known to some third party problem definition package such as a separately tailored interface application or a CAD system.

EKS_dm: This module allows the definition of a given problem in terms understandable to the EKS generated program and holds the information in the form of "X_def" objects. (The term "X_def" stands for 'something_definition' where 'something' might be a room, a material, a plant component, a site, etc. Typically a program will require several "X_defs" to define the site, building geometry & construction, plant layout and so on.) These "X_defs" can then be stored within the OODB.

EKS_mb: Program construction can be placed under the control of the OODB in which case the OODB-installed "Template" is consulted and correct class use can be guaranteed. Alternatively, and for use in cases where an OODB is unavailable, module EKS_mb can be used to build the required model from the "Template" as held in an ASCII file. This procedure is equivalent to the conventional link/ load operation.

* An object is a specific instance of a general class.

EKS_rm: Finally, this module is used to associate the program with its data model as held in "X_def" form. Again this operation can be placed under OODB control or invoked conventionally.

These EKS interface tools exist in two forms to handle the case where the OODB is present and the case where it is not. In either case they will appear identical to the user, the only difference being that the program "Template" and corresponding "X_defs" will either be stored within the OODB on within an ASCII file on disk.

The intention is that the EKS will improve researcher efficiency by placing model development on a task sharing basis. In addition, by allowing programs of different architecture to be built from a common set of classes, program integrity should improve and the validation process should be better served. In the longer term environments such as the EKS open up the prospect of radical changes to the design support process. For example, consider a design support system which allowed the definition of a design hypotheses by the graphical selection of component parts representing walls, windows, radiators, shading devices, sensors, a sun type, a site type and so on. If these components were related to EKS classes then the designer is effectively constructing, in real time, a model which is matched to the problem. Given the functionality of the EKS classes it would a relatively simple matter to then arrange that the instanced object start to operate immediately on selection. By bringing together hypothesis manipulation and performance appraisal a real-time computer-supported design environment is enabled. This in turn would enable the application of simulation at the earlier stages of the design process where the potential benefits are greatest.

To summarise the project's achievements:

- A system has been developed which demonstrates the construction of a range of models at different levels of abstraction - from simplified performance assessors to state-of-the-art simulators.
- The project has proved the technical feasibility of applying the OO programming approach to a complex engineering domain.
- The EKS demonstrator is designed to be extensible and portable, and can operate in either stand-alone or OODB modes.
- The EKS concept makes state-of-the-art developments accessible to users without the need to handle source code, providing the means to allow designers to participate in the design tool creation process.
- And the project has demonstrated the benefits to be gained from effective collaboration between the IT and domain communities.

The form and content of the EKS and the underlying rationale is also described elsewhere (Clarke et al 1991; Clarke et al 1988b; Charlesworth et al 1991).

EKS User Types

In examining the EKS it is important to appreciate the different possible user types corresponding to three separate levels at which users might interact with the system.

EKS Developer/ Extender

This user type will be concerned with developing/ extending the EKS class taxonomy and its associated software tools. While the EKS provides an extensive set of classes, for the foreseeable future it will be necessary to enhance the functionality of existing classes, provide alternative implementations of existing functionality and extend the taxonomy by adding completely new classes. These tasks breaks down into two levels: minor modifications/ extensions to existing classes (for example deriving from an existing class and

replacing a function); and major development work such as adding a new principal class. While the former could be carried out by someone with little C++ experience, the latter will require a sound grasp of the organisational principles behind the EKS, a reasonable knowledge of C++ and an acquaintance with the wider computational environment in terms of the ONTOS database and Unix™.

Program Builder

Users within this category will, typically, use the "Context" and "Template" building programs to select the required classes from the overall taxonomy as known to the OODB at any time. In-built within the taxonomy is the knowledge of class dependencies. For example, a "Room" 'knows' that it can use (among others) an "Air_volume" to handle its contained air mass. On selecting a particular "Room", the model builder is given the opportunity to select one of the possible variants of "Air_volume". The selected "Air_volume" is then checked to ensure that it provides at least the functionality that the selected "Room" expects. After all the required classes have been identified, the program specification is stored in the OODB. Users in this category will not require to know anything about programming the EKS, other than to be able to use the EKS tools, but should obviously have a sound grasp of both modelling and the domain.

End User

This user type uses the program constructed by the Program Builder to appraise the energy/ environmental performance of a particular design. Typically, this is a three stage process. Firstly, the system to be modelled has to be described in a manner acceptable to the EKS. It is anticipated that EKS-built programs will be interfaced to a separately constructed application to handle user inputs. For this reason only a rudimentary problem description interface facility is provided with the EKS itself - more sophisticated variants, for example based on emerging Intelligent Front-End systems (Clarke and Mac Randal 1991), could be developed in future. Secondly, the program is invoked via the program initiation mechanism. This results in the creation of the minimum necessary instances of the EKS classes. Control is then passed to the 'simulate' method of a top level "Context" object. This actually starts a simulation and interacts, directly or indirectly, with the user to establish the simulation requirements. The run-time interface is dictated by the "Context" class, so a Model Builder, by deriving a new "Context", can provide whatever interaction is considered appropriate. This could even extend to dynamic object substitution where, using the OODB, one algorithm could be substituted for another at run time in a manner which is transparent to the other program parts. Finally, as a simulation proceeds, the results obtained can be directed to a "Results" object for storage in the OODB or they can be transferred to disc files or displayed directly. Since they interact only with the run-time interface of the "Context" class, these users will not be able to distinguish EKS built programs from the current hand built versions.

EKS Classes

An EKS class is the encapsulation of an entity's:

- *Description* in the form of (usually private) data.
- *Behaviour* in the form of (usually public) functions.

For example a "Construction" might possess thickness and material as private data while offering functions such as resistance & diffusivity, time & frequency domain response functions, thermal discretisation and state variable manipulation as its behaviour. (The entire functionality of the EKS Demonstrator classes can be found in the class header files as held in ~eks/classes.)

Within the EKS classes break down into three basic types - base, principal and intrinsic:

base classes define the interface of the EKS principal classes and therefore their basic capabilities. These 'generic' classes cannot be used because their functions are virtual, being implemented only in the derived principal classes. Essentially base classes exist to allow a future extension of the EKS class hierarchy into other domains without the need to carry the thermodynamic functionality of the existing EKS classes. They also exits so that the entity they represent can be implemented in a variety of ways while guaranteeing that these different implementations will operate with the other EKS classes.

principal classes are selected by a program builder to define the capabilities of some target program. Each principal class is derived from a corresponding base class and offers a particular implementation of the virtual functions of its parent - that is they represent the physical or thermodynamic states of the entities they represent. In some cases, and with domain theory type classes in particular, several alternative principal classes will exist to represent the alternative theoretical approaches.

intrinsic classes are the work horses of the EKS serving to transport data between the principal classes, to control the class selection process at template specification time, to contain essential support data such as climatic time series and to dimension all properties of state. Intrinsic classes, because they are not selected by the program builder, are not shown in the EKS class taxonomy.

The separation of the underlying functions in this way gives maximum flexibility and code reuse when creating new programs. For example base classes provide a common point of entry for the principal classes and permit the classes derived from them to be used in different contexts. Several principal classes may be derived from a common base class while some of these derived classes may act, in turn, as the parent class of other derived classes. For example steady state and dynamic conduction classes may share the same base class, while finite difference and response function conduction classes may be derived from the dynamic conduction class. Any class can be replaced by a derived class (even at run-time) without necessitating any changes in the rest of the program. (In the conduction example, steady state or dynamic conduction classes can be interchanged without changing any other program class or, if conduction is of type dynamic, either finite difference or response function approaches can be selected.) Since the replacement class is derived from the replaced class, it has the same interface and so other classes can continue to behave as though the change had not occurred. It is this facility that gives the EKS its flexibility: in terms of model building (support of theoretical variants), in terms of run-time features such as dynamic model substitution, in terms of support for program validation and in terms of program maintenance. Where different program architectures handle the same task in different ways, derived classes provide a means to incorporate that functionality while ensuring minimal impact on the rest of the system.

Behavioural inheritance also reduces coding and improves reliability since new classes gain access to their parent's code and so need only add the code for the extra functionality they provide. Extensibility is also assisted because new classes and variants of existing classes can be added without requiring any changes to existing classes.

Appendix 1 lists the principal and intrinsic classes as contained within the EKS Demonstrator (note that base classes are matched to principal classes and so are excluded).

Principal Classes

The essence of the OO paradigm is the view that a program can be composed of independent objects communicating via messages. It was therefore well appreciated from the project's outset that the main challenge was the decomposition of building modelling into classes and subclasses and the definition of the properties of these classes in terms of their data members, behaviour and inter-relationship. It was also appreciated that in the OO research and application field there was still no commonly accepted definition of the OO approach (Editor et al 1990). The project therefore struggled in its early stages to develop a rational

basis for system decomposition and class identification.

It has been stated that "the most important concept in the object-oriented approach is data abstraction" (ibid.). To achieve this (and the subsequent data encapsulation), one approach is to undertake a data analysis of the target domains (here building physics and thermodynamic simulation). As the EKS was intended to be a model building environment and not a building model, such an approach was considered to be impracticable in terms of the resources available and the total number of possible modelling approaches. Instead a functional decomposition approach was adopted (Clarke et al 1989) which adhered to the following strategy.

Step 1 The domain (building energy modelling) was decomposed into its primitive functionality such as sun position tracking, conduction, convection, radiation, equation solving, polygon operations and the like. These are the functions, albeit at different levels of abstraction, found within all contemporary modelling systems. The initial research task was therefore to undertake a comprehensive functional analysis of existing models for building, plant and control system simulation (Tang 1989). The functions so identified are described elsewhere (Wright et al 1990 and Clarke et al 1990).

Step 2 A minimum level data requirement for each function was identified. For example, a one dimensional layer conduction function will require a set of thermophysical properties irrespective of the underlying mathematical model and so its minimum data requirement (in EKS terms) is a "Material" object (or strictly speaking a pointer to such an object) and a "Dimension" object. The matrix inversion function requires the topology and coefficient values (or the means to determine such values) irrespective of the inversion technique to be used. It is also worth noting that only EKS relevant functionality was considered. For example when considering sun-related processes the sun position is relevant while angular rotation is not.

Step 3 At this stage the functions were associated with a physical class which logically would know about the context of the function. For example "Construction" logically knows about thermal resistance, while "Room" logically knows about shortwave response. More contentious perhaps, "Construction" may know about overall, reference U-values (which have prescribed surface overall resistance values) while only "Building" may know about overall, actual U-values because to construct this parameter requires knowledge of the properties and thermodynamic state of different entities (air volumes, surfaces, constructions, exposures and so on). It follows that class functions must relate only to the intrinsic data and properties of a class and not require the existence of, or assume data or properties of, another class. This ensures that a class will encapsulate only data which is pertinent to that class and that the data members of each class, as required to support its functions, can be guaranteed to be available at run-time to the object made from the class.

Step 4 Abstract classes are now identified by gathering together related functionality as implied by the functions of the physical classes. For example, the convection function of class "Room" requires surface area, hydraulic diameter and heat flow direction all of which are geometrical entities and so are gathered together into an abstract class "Polygon". This ensures that classes will not possess functionality where that functionality could be made generally available by encapsulation within another class.

Step 5 Where a class has one or more domain theory functions (for example convection, shortwave response and occupant behaviour in the case of a class "Room") these functions are implemented as links to abstract classes containing the required functionality of the domain theory. The different formulations of any given domain theory can then be handled by derived classes: this serves to facilitate the handling of the multiplicity of domain theories, without incurring combinatorial explosion in the parent class.

The foregoing class identification approach ensures that class functions relate to the intrinsic data and properties of the class and do not require the existence of, or assume data/ or properties of, another class. For example, the class "Room" would have no functions which would require any knowledge of another room. Such a function would be located within a class which intrinsically had the right to know about both "Room" instances - for example a "Building" class.

It is recognised that other grouping mechanisms could have been employed - for example grouping on the basis of computational intensity - but, because the EKS classes should be semantically acceptable to the modelling community, these were not considered relevant to the EKS. Nevertheless it is thought likely that the foregoing process has given rise to the same classes as would have resulted from a conventional data decomposition of real world entities[#]. The important aspect about this approach is that it immediately eliminates all aspects of the problem that are not related to the target domain. It also provides a mechanism to deal with the physical/ abstract mix so dominant in thermodynamic modelling.

Appendix 1 lists the EKS principal classes, most of which relate to physical entities such as rooms, walls, surfaces, etc. while others relate to the underlying computational support provided by the EKS.

Computational Support Classes

As a simulation model development platform, the EKS will eventually encounter a wide range of systems represented by different mathematical categories, e.g. hyperbolic partial differential model for aerodynamic systems, parabolic partial differential model for diffusion systems, elliptic partial differential model for wave and vibration systems, ordinary differential models for lumped parameter systems and so on. This calls for an efficient and generic computational support facility which provides the range of mathematical tools and solvers required.

To achieve this, the EKS employs the following techniques:

- An internal mathematical representation format based on vector symbolism for the general system representation.
- A common mathematical interface for communication between the EKS classes via special transport classes.
- Dynamic data structures and sparse matrix techniques for system maintenance.

By these mechanisms access can be obtained to the EKS generalised solver classes which embrace a spectrum of analytical, numerical and statistical solution methods. The solver class mechanism has been designed to accommodate most of the currently available equation solving methods. This is achieved by using the C++ class inheritance mechanism in which a base solver class defines the solver interface by means of virtual functions which are then implemented in a set of derived classes. In this way a given solver can be selected at run time without affecting program structure. As an example, the following codes shows the implementation of the solver class.

```
class Solver : public EKSObject {
protected:
    Solver(Metaclass* meta, Solver_def* def);
    Solver(Solver& c);
    ~Solver();
```

The CEC Combine project which is attempting the development of an integrated data model has used the technique of data decomposition of existing design tools with a subsequent OO encapsulation (COMBINE 1992).

```
virtual  void time_discretise(Matrix& m, Vector& q, Vector& s, int
n, float dt);
public:
virtual Vector& execute(Equation_iterator eqn_iter) = 0;
};
class Gauss_column_pivot : public Solver {
public:
     Gauss_column_pivot(Metaclass* meta, Solver_def* def);
     ~Gauss_column_pivot();

virtual Vector& execute(Equation_iterator eqn_iter);
};
```

In the above, class "Solver" is the parent class to class "Gauss_column_pivot". The actual code implementing the solution algorithm is in function execute() within class "Gauss_column_pivot". In class Solver, the execute() function is virtual.

At the present time many of the available solver packages are based on algebraic methods for linear systems and iterative methods for non-linear (and linear) systems. The equation representations required for these packages conform directly to the EKS internal representation as given by equations (1) and (2) later. This means that these solvers can be encapsulated as solver classes within the EKS with only minor modification. For those proprietary solvers which are embedded in some coded algorithm, they must firstly be separated before they can be implanted within the EKS.

Within the EKS Demonstrator two generic types of solver are provided each with several implementations. The first type includes conventional vector/ matrix operators and linear, nonlinear system solvers. These are mathematical tools usually available in software libraries and the like. The EKS makes no effort to improve the efficiency of such tools, only to provide the means by which the data they require can be encapsulated and supplied. For this type of solver the EKS provides a number of well known methods such as Runge-Kutta, Gear, etc. (see Appendix 1). The second type comprises a direct solver which is based on an ‘implicit row-wise sparse array’ technique. This solver type receives, stores and processes only the non-zero entries of the vectors and matrices which define the problem and so is able to provide high efficiency by ensuring that minimal non-zero entries are created during the elimination process.

In addition to these solver types, a range of support tools are available to determine vector inner products, vector norms, matrix products, matrix determinants, matrix inverses, pseudo-inverses, singular value decompositions, matrix eigen-values, matrix eigen-vectors and so on. These features of the EKS allow direct system study and could be used to compose customised solvers.

Finally some effort was expended on the two-way mapping between system graphs as employed in the graph theory method and the state-space representation employed by the EKS. This means that for any given system graph, the EKS is able to provide an identical set of equations which are then encapsulated in the theory classes. This is the fundamental tool for system automation. For example, after a system has been drawn on the screen via a graphic interface, the topology of the system network can then be depicted by the characteristic matrices of the system graph. The identical set of system state space equations can then be obtained via the methods of graph theory (Tang 1990). It should be noted however that while some experimental classes were developed within the timescale of the project, the integration of the techniques of graph theory within the EKS was outwith the project’s remit and resources.

Intrinsic Classes

In addition to the principal classes, the EKS also has a range of special classes termed intrinsic. These include:

- Data classes to encapsulation climatic data, material properties, program results and the like. In some cases instances of these classes may be held within the OODB as persistent objects. These objects, which are equivalent to the data-sets available for use with conventional energy programs, are available to all EKS constructed programs.

- Dimension classes - temperature, pressure, mass flow rate, weight, etc. - used to ‘type’ all EKS object returns and so ensure data security (Wright el al 1992).

- Computational support classes - vector, equation, matrix, vertex, variable - to encapsulation theory and support data such as time, location and state variable.

- Infrastructure classes which assist in the program building process and facilitate object control at run-time. Such classes include "Template" to hold the program definition, "Metaclass" to hold the class relationships, "List" to enable the processing of lists of related entities (such as the "Construction" objects in a "Room") and "Network" to hold topological information.

The "Template" Class

The "Template" class is a framework for defining a particular program which can then be stored in the OODB or on disk for later recall when it can be associated with specific building description data. Such a program is then run by the user much like a conventional program, interaction with the OODB being largely hidden.

The "Template" has three main tasks:

- To ensure that only legal combinations of the EKS classes can be used together.

- To hold a permanent record of the program composition, and provide a focal point for model builder and end user access to the program.

- And to provide the mechanism by which the end user applies the simulation program to their specific problem.

As described above the EKS consists of a collection of classes, which can be used by the program builder in various configurations to form specific programs. Since not all simulation programs require the full set of available principal classes, and there can be several alternative implementations of these classes, a program composition facility is required to ensure that only legal combinations of the supplied classes are used. Furthermore, once a legal program has been defined by a program builder, it is necessary to capture its configuration/ composition as a template for future use.

The template for a particular simulation program is implemented as an instance of a "Template" class. Thus, the objective of a program builder who is developing a new simulation program is to produce a "Template" instance. Clearly, during the creation of this instance, there is an opportunity to prevent the attempted use of illegal or incomplete combinations of EKS classes. In order to do this, the "Template" class requires information about the connectivity and dependencies of all the EKS classes. As this cannot be built into the "Template" class without completely destroying the extensibility of the EKS, the "Metaclass" mechanism (see below) has been provided to hold this and related information.

Of course, end users are not interested in merely having a specification for a simulation program, they will want to simulate a specific problem. Rather than having pre-allocated objects into which the problem data is read, as in conventional Fortran simulation programs for example, the "Template" instance can produce, at run-time, a program tailored specifically for the particular problem being addressed. This is achieved by giving the "Template" a ‘simulate’ function that obtains the data defining the user’s problem, instantiates

the necessary EKS classes and transfers control to the top level "Context" instance. Furthermore, the "Template", with its in-built knowledge of the composition of the program is an obvious place to hold those user accessible functions for examining and interacting with the program.

Several examples of EKS "Templates" are given later and can be found in the sub-directories of ~eks/demo.

The "Metaclass" Class

A "Metaclass" (there is one for each principal class) is responsible for the following tasks:

- To hold information about its related class, its capabilities and requirements.
- To assist the "Template" in ensuring the integrity of the resultant program.
- And to enable its class to interact with new variants of subordinate classes.

The power and flexibility of the mix and match approach to program building offered by the EKS carries within itself two potential dangers. Firstly, there is the problem of ensuring that the objects selected by the user are actually compatible, both in terms of software interoperability and in terms of conformance to the same underlying mathematical/ physical model. Secondly, there is the difficulty of ensuring that the system is adequately extensible, in that new classes and variants of classes can be added later without requiring any changes to existing classes. This is particularly important from the validation perspective, as even apparently simple modifications to a piece of code can invalidate any tests which have been performed on the class.

Clearly, extensibility is not a problem when a new class or class variant wishes to use an existing class. However, it is anticipated that in the EKS the normal pattern will be for users to modify the more fundamental classes, such as the domain theories, as and when the state-of-the-art progresses. This raises the problem of ensuring that the existing classes further up the class taxonomy can correctly use classes which do not yet exist.

There are two aspects to this compatibility requirement. Firstly, the class software viewpoint, equivalent to ensuring there are no compilation errors in a conventional program. In OO programming, the use of derived classes and late binding offers a mechanism by which this can be done. In a program, any class can be replaced by a derived class without necessitating a change in the rest of the program. Since the new class is derived from the old one, it has the same interface (or a superset) and thus the other classes continue to access the derived class as if it had not been changed. However, where the programmer has provided an alternative implementation for some of the base class' functionality, the new version of the functionality is used. Any extra functionality added to the original (base) class then becomes available to any other new classes being added to the program. Thus, providing the existing classes can be made to talk to the new classes, extensibility is ensured. Actually achieving this is not straightforward, principally because it means that no class can itself create an instance of a class it wishes to use, but has to be given a pointer to an existing instance of the required class. Then, since it could equally, and transparently, be given a pointer to an instance of a derived class, extensibility is ensured. All that is required, therefore, is for an external entity to take responsibility for creating the instance and passing in the pointer.

The second compatibility aspect is from the domain viewpoint. To ensure that the new classes are compatible with all the other classes in the program requires knowledge of what functionality is required/ provided by both the new class and (potentially) all the other classes. Given this, the Template can ensure that only valid combinations of classes can be put together into a program. Provided a new class has the appropriate functionality, that is it is derived from a suitable class, it will be acceptable. So all that is necessary is to know, for each class, what variants of other classes it requires. Clearly, building this knowledge into the "Template" constructor[#] would result in a monolithic, non-extensible "Template", so the information has to

be held elsewhere.

The way the EKS handles both these compatibility issues is to associate with each class a "Metaclass". The class then holds just the code necessary to implement the functionality used to carry out the simulation. All the higher level information/ functionality necessary to ensure program integrity and system extensibility is placed in the "Metaclass". When the "Template" constructor is creating a new program, it checks with the "Metaclass" classes (of those classes selected by the program builder) that together they form a "valid" program - for example that all domain theories generate the same type of equations and that the solver can handle this sort of equation. Also, at run-time, when an object wishes to use another class, for example the "Layer" class will require a "Conduction" instance, the "Metaclass" of "Layer" (actually termed "Meta_Layer") will create instances of the particular "Conduction" variant specified by the "Template", but return them to the "Layer" class as if they were of the variety that "Layer" expected. This ensures that to use a new "Conduction" class, the code of "Layer" does not require modification. Currently, the "Metaclass" achieves this dynamic class manipulation by using the schema modification facilities provided by the ONTOS database system.

To assist model builders to quickly create and test new classes, a default "Metaclass" will be automatically provided if the user does not explicitly supply one. This default will not carry out any consistency checking, since this is the responsibility of the person using the class. Clearly, if a class is to be given to anyone else, the appropriate "Metaclass" should also be supplied.

The List Class

Since C++ does not support the definition of a vector class with the type of elements as argument, the "List" class was required. This is a generic carrier class which uses the macro processor to mechanise the creation of a class "Vector" at run time. When a list is created at run time, the C macro pre-processor concatenates its argument and calls the macro with that name, the macro expansion mechanism is then activated to create dynamically a new class which contains homogeneous types of class "Vector". Within the generic "List" class, a ‘doubly-linked list’ (see later) technique is used to manipulate the vectors (Tang 1991). This approach has the following advantages:

- Dynamic memory allocation so that only the actual vector size is required.
- Faster array accessing than with vector indexing (although reducible to normal vector indexing if required).

The following table lists the functions which are provided to support the maintenance of lists:

Function	Description
size()	size of the list
append(type* a)	append element "a" at the end of the list
append(type* a, type* b)	append element "a" after element "b" in the list
insert(type* a)	insert element "a" at the head of the list
insert(type* a, type* b)	insert element "a" before element "b"
remove(type* a)	remove element "a" from the list
first()	get first element from the list
last()	get last element from the list
clear()	delete the entire list
=	operator, assign a list to another
+=	operator, append a list to another
+	operator, concatenate two list

A ‘constructor’ is the special class function which creates the object instance.

A complementary class, "List_iterator", is provided to support sequential and random list scanning and to allow list element retrieval via conventional subscript indexing.

The Network Class

Similar to the generic "List" class, the generic "Network" class uses the macro expansion mechanism of the C pre-processor to allow the run time creation of a network of homogeneous arc and node types as defined in its argument list. The "Network" class then uses the "List" class for the maintenance of this arc/ node list. Again a class "Network_iterator" is provided to support arc/ node list manipulation within networks.

Within the EKS Demonstrator the "Network" class is used extensively:

- Within the "Building" class to represent "Room" contiguities.
- And within the "Plant" and "Fluid_flow" classes to represent system connection topologies.

The following table lists the functions provided to support the maintenance of networks:

Function	Description
add_node(ntype* a_node)	add a node into the network
add_arc(atype* an_arc, ntype* src_node, ntype* dst_node)	add an arc between two nodes in the network
no_nodes()	total number of nodes in the network
no_arcs()	total number of arcs in the network
connected(ntype* n1, ntype* n2)	check whether nodes "n1" and "n2" are connected
connected(atype* a, ntype* n)	check whether arc "a" is connected to node "n"
arc_source(atype* a, ntype* n)	check whether node "n" is the source of arc "a"
arc_target (atype* a, ntype* n)	check whether node "n" is the target of arc "a"
connection(ntype* n1, ntype* n2)	get the arc between node "n1" and "n2"
connection(atype* a, ntype* n)	get the other node connecting node "n" and arc "a"
arc_source(atype* a)	get the source node of arc "a"
arc_target (atype* a)	get the target node of arc "a"
nodes()	get the nodes in the network
arcs()	get the arcs in the network
connections(ntype* n)	get all the arcs connected to node "n"
neighbours (ntype* n)	get all the nodes neighbouring node "n"
node_iterator()	get the node iterator of the network
arc_iterator()	get the arc iterator of the network
arc_iterator(ntype* n)	get the arc iterator of node "n"

EKS Class Taxonomy

The EKS classes are organised into a class taxonomy which is designed to offer the functionality required to build alternative program architectures while guaranteeing security of use. These classes represent the properties and behaviour of the various entities found in buildings (rooms, constructions, surfaces, etc.), in plant systems (components, connections, controllers, etc.) and in programs (numerical solvers, time control, etc.). The EKS inheritance mechanism gives the EKS user access to alternative implementations of these classes where appropriate and supports the addition of new implementations.

Once identified the EKS classes were organised into a three dimensional ‘used by’ hierarchy. Figure 2 shows one plane of this hierarchy which defines the relationship between the principal classes from which programs are actually built - rooms, constructions, sites, surfaces, layers, plant components, solvers and so

on.

Orthogonal to this plane is the EKS inheritance hierarchy used to represent the alternative domain theories. For example Figure 4 shows alternative conduction theories as related to the principal class "Layer".

Finally the third plane of the taxonomy represents conventional OO inheritance to enable code sharing even among classes of fundamentally different types. This inheritance mechanism reduces coding and improve reliability. Inheritance means new, derived classes need only add code for the extra functionality they are providing since they automatically use their parent's code, presumably already 'validated' to some extent. As an example, consider the classes derived from "Physical_entity".

Physical_entity	(abstract class, not used directly)
Contents	(general entity, not producing heat)
Heat_source	(abstract class for internal gains)
Occupant	(describing occupants)
Equipment	(describing heat generating contents)
Lights	(special kind of equipment item).

Clearly, "Heat_source" is a specialisation for thermal modelling, but other classes could be derived from "Physical_entity" or "Contents" for other applications. The importance of inheritance relates to the extensibility of the system. For example, a "Cavity_layer" could be seen as a specialisation of "Layer", having many of the same instance variables (for example orientation and length), and functions (for example area calculation), but with the new instance variables added (for example cavity resistance), and some functions replaced/ enhanced (for example heat flux calculation).

In this way the principal classes are related, while the orthogonal classes can be considered to be separate class hierarchy rooted on each principal class. The many intrinsic classes are thereby hidden from the EKS user (but not the EKS Developer).

Some particular points to note:

- Classes with associated state variables - for example "Room" and "Construction" will have an instance for each physical occurrence. Other classes, for example "Finish", present the possibility of shared instances. This has strongly influenced the instantiation protocols: no class which contains state variables may instantiate a class which does not.

- Because, for example, the "Building" class must handle inter-room processes then it needs to be the instantiator of "Room" objects (or at least be informed of their existence). Equally, and more surprisingly, the "Construction" class would not be instantiated by the "Room" class since the latter can have no *a priori* knowledge of another room. The "Construction" class is therefore placed under the control of the "Building" class which has full knowledge of overall topology.

- The significance of the EKS approach is that, at run-time, object control is ordered with the control flow acting down the taxonomy hierarchy. A developer subsequently creating a different implementation of a class can therefore assume that its data will be available. This approach ensures that the EKS cannot be used to build anarchic programs, with all their synchronisation problems, and should be contrasted with the normal OO programming approach where it is the responsibility of the function to get its data. To do this the function would need to know about its context.

- Each principal class in the taxonomy is effectively a generic class, capable of being implemented in different ways. For example, the "Conduction" class, whose function is to represent the conduction process, can be implemented using a steady state, a response function or numerical approach as

shown in Figure 4.

Clearly there is no single best way to structure the classes, and any structure for a real-world system will be biased towards a particular view (such as energy modelling as here). However, an efficient structure should be adaptable for other areas (by using the more general base classes), and be easy to use and extend.

Theory Representation

At the present time many alternative algorithms exist for application in a building modelling context. One objective of the EKS is to ensure that future model users are not limited to only those out of date algorithms imposed by a particular system. To achieve this, the EKS offers a spectrum of theory classes with each theory covered by more that one algorithm. At program construction time, the EKS principal classes can then be coupled to any theory class as dictated by the intended application and the required level of complexity. In this way, the principal classes provide the interface to the theory variants. This is achieved by arranging that the alternative mathematical models of any given theory (such as conduction, air flow, room shortwave response, etc) are derived from a single base class and so possess the same interface.

As an example of the process, consider the "Conduction" class for which Figure 4 shows the derived class variants. The "Layer" class possesses a function which returns the theory for conductive heat transfer. The actual code which represents this theory is not held in the "Layer" but instead is located in a class derived from the "Conduction" base class. In effect the "Layer" object's conduction function asks the "Conduction" object to return the appropriate theory.

In order to achieve mathematical consistency and minimise numerical error propagation, most building simulation programs adopt a coherent model discipline throughout the system, e.g. finite difference, finite element or response factor methods. Such an approach will give rise to a set of equations - algebraic, differential, linear or non-linear equations - which can be simultaneously solved by a variety of techniques. For those systems which use different mathematical approaches to represent different components (rooms, coils, etc.), each component is treated in isolation and linked to the other components by parameter passing. The solution of such systems is effectively by iteration.

Within the EKS a technique was required to represent the many different possible mathematical theories from which programs could be constructed. The requirement was that this be done in a manner which preserved the integrity of the original mathematical model in a mathematically consistency way.

One way to achieve this would be to provide representation schemes for all the different possible approaches - a finite difference layer and a finite difference conduction; a steady state layer and a steady state conduction and so on. Such an approach would require:

- Matched pairs for each theory type.
- And a switching device to allow pair selection at run-time.

Clearly this would be an inelegant solution in that the former requirement gives rise to the problems of interface complexity and combinatorial explosion (Clarke et al 1990b) while the latter requirement results in code redundancy and the need to propagate the switching code even to the most simple programs.

To investigate the system representation format which would best satisfy the needs of the generalised model building environment, a study into the system representation tools of graph and matrix methods was carried out (Tang 1990b). The outcome was the decision to base the EKS internal theory representation on a vectorised state-space equation method. In the event the method was implemented using sparse matrix technique incorporating a bi-directional linked-list management system as explained below and elsewhere (Tang 1990). Using this technique it is possible to represent most of the equation-based theories in the domain of building energy modelling in a consistent manner as vectorised state-equations.

Equation Representation

Consider, as an example, a spatially discretised wall being represented in terms of its temperature function θ by the following equation.

$$\begin{cases} c_1 \dfrac{d\theta_1}{dt} = h_o(\theta_o - \theta_1) + R_1(\theta_2 - \theta_1) + \sigma\varepsilon_1({\theta_{so}}^4 - {\theta_1}^4) \\ c_2 \dfrac{d\theta_2}{dt} = R_1(\theta_1 - \theta_2) + R_2(\theta_3 - \theta_2) + q_2 \\ \cdots \quad \cdots \quad \cdots \\ c_n \dfrac{d\theta_n}{dt} = h_I(\theta_I - \theta_n) + R_{n-1}(\theta_{n-1} - \theta_n) + \sigma\varepsilon_n({\theta_{si}}^4 - {\theta_n}^4) \end{cases} \tag{1}$$

in which the heat transfer coefficients and thermal physical properties (h_o, h_I, R_i, ρ, K_i etc) may be represented by non-linear functions.

In the EKS, this equation-set is represented using vector symbolism:

$$\mathbf{C}\ \dot{\theta}(t) = \mathbf{A}(\theta, \mathbf{U})\ \theta(t) + \mathbf{B}(\theta, \mathbf{U})\ \mathbf{U}(t) \tag{2}$$

in which $\mathbf{C} = diag[C_i]$; $(i = 1, 2, \ldots, n)$ and

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{2,1} & 0 & \cdots & 0 \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & \cdots & a_{n,n} \end{bmatrix}$$

in which $a_{11} = -(h_o + R_1 + h_{ro})$, $a_{12} = -R_1$, $a_{21} = -R_1$, ..., $a_{nn} = -(h_I + R_{n-1} + h_{ri})$;
$h_{ro} = \sigma\varepsilon_1({\theta_{so}}^2 + {\theta_1}^2)\ (\theta_{so} + \theta_1)$, $h_{ri} = \sigma\varepsilon_n({\theta_{si}}^2 + {\theta_n}^2)\ (\theta_{si} + \theta_n)$, $\theta(t) = [\theta_1(t), \theta_2(t), \ldots, \theta_n(t)]^T$,
$\mathbf{B} = [b_{i,j}]$; ($i = 1, 2, \ldots, n$; $j = 1, 2, 3, 4$) and $\mathbf{U}(t) = [\theta_o, \theta_{so}, q^o, \theta_I, \theta_{sI}]^T$.

Note that entries in the coefficient matrices $A(theta, U)$ and $B(theta, U)$ are generally nonlinear functions and that returns from these functions are not restricted to real numbers. For example the following table gives the possibilities in the case of the different equation systems.

Equation type	Entries in matrix
Partial differential equation.	Partial differential operators.
Spatially discretised nonlinear partial differential equation.	Pointers to functions.
Linear time variant differential equation.	Time series.
Linear algebraic equation.	Constants.

It can be shown that for any nonlinear system which is represented by its original simultaneous nonlinear equations, there always exists an identical transformation in the form of equation (2). Furthermore, for any system which is transformed into equation (2) from its original nonlinear representation, it can always be identically transformed back to the original representation (Tang 1990). Therefore the representation of equation (2) is an identical transformation. This type of representation has been implemented in the EKS Demonstrator to handle the thermal representation of both building and plant components.

OO Implementation

Theory in the form of equation (2) is encapsulated within the EKS via four special transport classes: "Equation_set", "Equation", "Coefficient" and "State_variable". These classes are serviced in turn by parameterised carrier classes which are based on the bi-directional linked list methodology. Figure 5 shows the dependency structure of these transport classes.

The following codes show the data definitions of the "Equation_set", "Equation", "Coefficient" and "State_variable" classes.

```
class Equation_set : private List(Equation) {
    friend Equation_iterator;
    protected:
        List_iterator(Equation) eqn_iter;
    public:
        Equation_set();
        Equation_set(Equation_set& e);
        ~Equation_set();
        Equation_set& operator=(Equation_set& e);
        Equation_set& operator+=(Equation_set& e);
        Equation_set& operator=(Equation_iterator e_itr);
        Equation_set& operator+=(Equation_iterator e_itr);
        int size();
        Equation* first();
        Equation* last();
        void append(Equation* eqn);
        void insert(Equation* eqn);
        Equation* operator[](int i);
        Equation_iterator iterator(EquationOrder order = AS_EQNS);
        friend ostream& operator<<(ostream& s, Equation_set& es);
};

class Equation {
    protected:
        Equation_enum the_equation_enum;
        State_variable* the_state_variable;
        List(Coefficient) the_coefficients;
        Energy the_gain;
        ThermalResistance the_resistance;
        SpecificHeatCapacity the_capacity;
    public:
        Equation(State_variable* a_state_variable, Equation_enum a_enum,
        float self_coupling_coeff, Energy initial_gain,
        ThermalResistance resist, SpecificHeatCapacity capac);
        Equation(Equation& e);
        ~Equation();
        Equation& operator=(Equation& e);
        State_variable* state_variable();
        Equation_enum equation_enum();
        Energy gain();
        ThermalResistance resistance();
        SpecificHeatCapacity capacity();
        void append_coefficient(State_variable* a_state_variable,
        float a_value,
        State_variable_enum a_state);
        void change_coefficient(State_variable* a_state_variable,
        float a_value);
        void remove_coefficient(State_variable* a_state_variable);
        float coefficient_value(State_variable* a_state_variable);
        void add_gain(Energy a_value);
    friend ostream& operator<<(ostream& s, Equation& e);
    List_iterator(Coefficient) coeff_iterator();
```

```
};

class Coefficient {
    protected:
        friend Equation;
        float the_value;
        State_variable* the_state_variable;
        State_variable_enum the_state_variable_enum;
    public:
        Coefficient(State_variable* a_state_variable, float a_value,
        State_variable_enum a_state);
        Coefficient(Coefficient& c);
        ~Coefficient();
        Coefficient& operator=(Coefficient& c);
        float value();
        State_variable* state_variable();
        State_variable_enum state_variable_enum();
    friend ostream& operator<<(ostream& s, Coefficient& c);
};

class State_variable {
    private:
        static int next_id;
    protected:
        int the_id;
        float the_value;
        EKSObject* the_owner;
    public:
        State_variable(EKSObject* an_owner, float a_value);
        State_variable(State_variable& n);
        ~State_variable();
        State_variable& operator=(State_variable& n);
        EKSObject* owner();
        int id();
        float state_variable();
        void state_variable(float value);
    friend ostream& operator<<(ostream& s, State_variable& sv);
};
```

To summarise the equation handling process:

- Mathematical theories are constrained to conform to a state-space matrix equation form to facilitate a representation style which separates equation structure (topology) from coefficient values (actual values or function identifiers).

- State-space matrix equations are implementation as compressed matrices comprising "Coefficient", "Equation" and "Equation_set" objects.

- "Coefficient" objects encapsulate real values if the problem is linear, function pointers if the problem in non-linear, a node identifier, the state variable type and coefficient dimensionality.

- "Equation" objects encapsulate a "List" of coefficients, a node identifier and the equation dimensionality.

- "Equation_set" objects encapsulate a "List" "Equation" objects.

The run time arbitrary selection of alternative methods is achieved by applying the methodology of explicit polymorphism which in C++ it is implemented using the constructs of derived class and virtual function (Stroustrup 1987).

The base class of a given type of domain encapsulates data common to all implementations and function entries implemented as virtual functions: this means that a base class can never be directly used. The specification of the base class ensures that the derived classes have the same interface. The alternative mathematical models of any given theory (such as conduction, air flow, room shortwave response, etc.) are derived from a single base class and so possess the same interface. The derived classes - that is the different implementations - reimplement the virtual functions.

In doing so, the base class of the domain 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 classes derived from the base class. Therefore, at program construction time, the EKS physical classes can be coupled to any theory class as dictated by the intended application and the required level of complexity.

Consider the example of layer conduction in which individual conduction equations - a steady state layer will possess one, a one-dimensional finite difference formulation one per node in a normal direction, and so on - are encapsulated within "Equation" objects referenced to a unique region. These "Equation" objects contain "Coefficient" objects as private data. Finally, all the equations of a "Layer" are encapsulated within an "Equation_set". "Construction" and "Room" objects then possess a "List" of "Equation_sets" to represent their energy balance equations. It follows then that the "Equation_set" class is a fundamental entity which is passed up the class taxonomy from "Layer" to "Context" and hence to "Solver" where the equations can be then be transformed into the required format.

At present the EKS offers a range of alternative theories for each topic - conduction, convection, air flow, shortwave & longwave response, casual gains, control behaviour and so on. Furthermore these theories have been selected to represent the range of possible models from a low order approach - time invariant, user specified air changes for example - through intermediate order formulations, to (near) state-of-the-art methods. The purpose of this treatment is to demonstrate the capability of the EKS to accommodate a spectrum of modelling techniques and to support class interchangeability. The following table lists the domain theories as implemented in the EKS Demonstrator.

Domain	Level		
	Basic	Intermediate	Advanced
Convection	Standard design data for convective heat transfer coefficients when represented as scheduled time series.	Empirical correlations for the convective heat transfer coefficients.	
Conduction	U-value method for steady-state systems.	Response function & transfer function methods for linear, time invariant dynamic system.	Spatially discrete numerical method with state variable dependent thermophysical properties to handle non-linear dynamic systems.
Longwave radiation	Standard design data for the radiative heat transfer coefficient when represented as scheduled time series.	Linearised formula for the black body radiative heat transfer coefficient with respect to area weighted view factors.	4th power radiative exchange formula for the grey body radiative heat transfer coefficient with respect to view factors based on spatial coordinates.
Shortwave distribution	Shortwave radiation distributed to room surfaces and objects according to predefined proportions.	Shortwave radiation distributed to room surfaces and objects based on view factors.	Numerical representation of the shortwave radiation distribution in a room including multiple reflection and absorption by a ray tracking algorithm.

Domain	Level		
	Basic	Intermediate	Advanced
Occupant gain	Standard design data for occupant heat gains when represented as scheduled time series.	Non-linear correlation equation to determine heat gains as a function of influencing parameters.	
Equipment gain	Standard design data for equipment heat gains when represented as scheduled time series.	Non-linear correlation equation to determine heat gains as a function of influencing parameters.	
Lighting gain	Standard design data for lighting heat gains when represented as scheduled time series.		Import luminance distribution data from lighting simulation and convert to thermal energy distribution.
Lighting luminance distribution	Luminance distribution due to point sources weighted according to predefined factors and represented by scheduled time series.	Luminance distribution from point sources calculated as function of normal distance between surfaces and light sources.	Numerical representation of luminance distribution from point sources considering surface multiple reflection and absorption by ray tracking algorithm.
Sky irradiance	Isotropic sky irradiance model.	Anisotropic sky irradiance model.	

In this way the EKS provides structured alternative theories in support of the alternative modelling approaches while guaranteeing security of use.

Implementation of Theory Classes

To explain the implementation of the theory classes within the EKS consider the classes developed to represent the thermal energy diffusion within solids. The "Conduction" class provides three alternative treatments for thermal conduction theory: steady-state, response function and finite difference. Consider the "Conduction" base class:

```
class Conduction : public EKSObject {
protected:
    Conduction(Metaclass* meta, Conduction_def* def);
    Conduction(Conduction& c);
    ~Conduction();
public:
virtual Equation_set generate_equations(State_vector* sv);
    EKSObject* lyr;
        Equation_iterator yr_iter) = 0;
virtual Equation_set combine_equations(Equation_iterator lyr1_iter);
    State_variable* lyr1_surf_node;
    Equation_iterator lyr2_iter;
    State_variable* lyr2_surf_node;
    EKSObject* data_supplier) = 0;
virtual void inject_energy(Equation_iterator lyr_iter);
    State_variable* node_in_lyr;
    Energy gain) = 0;
};
```

Three derived classes - "Conduction_steady_state", "Conduction_response" and "Conduction_finite_volume" are established to represent the different modelling approaches. When called by class "Layer", the "Conduction" function ‘generate_equations()’ generates N state-space equation(s) for a homogeneous layer where N is the number of layer subdivisions. The equations are return to "Layer" in the form of a pointer to an object of type "Equation_set". The actual contents of the "Equation_set" object will depend on the

derived class of "Conduction" being used and thereby on the implementation of the ‘generate_equations()’ function. For a "Conduction_steady_state" class, the returned "Equation_set" is simply $UA\delta\theta$; for a "Conduction_response" class the "Equation_set" is the convolution equation which incorporates the response factor series; while for a "Conduction_finite_volume" the "Equation_set" will contain the nodal Fourier Numbers coefficients of the energy balance state matrix equation.. In similar manner the function ‘inject_energy()’ is called by "Layer" to handle cases where there is heat generation. For a "Conduction_finite_volume" this might entail modification to the state-space equations while for a "Conduction_steady_state" the function is null.

The ‘combine_equations()’ function of "Conduction" is called by "Construction" to concatenate the representation of different layers within a single construction. For a "Conduction_finite_volume" this will entail a number of matrix equation operations, for a "Conduction_response" the combination of layer response functions and for a "Conduction_steady_state" the combination of resistances in series.

The important point is that in each case the return from "Conduction" to "Layer" or "Construction" is an "Equation_set" and so the problem of interface combinatorial explosion is overcome. (For those cases where alternative derived classes do require different interfaces, it is recommended that a shallow inheritance structure is used.)

EKS in Use

The EKS can be used with or without an OODB. In each case the template building program, EKStb, the data definition program, EKSdd, and the program building and execution program, EKSrm, will appear similar. The only difference if that in the former case the various operations as controlled and coordinated by the OODB, whereas in the latter case the template is stored on disk and the program is built and executed in a conventional manner.

To demonstrate the EKS use, the system comes complete with several example "Context" classes. These define several possible programs progressing from trivial site analysis, through building models of intermediate complexity to state-of-the-art (in terms of theory) multi-zone building models with plant. What follows in this section is a brief description of these programs in terms of their specification and capabilities. Where appropriate the EKS facility of theoretical substitution and incremental program building is demonstrated.

Site Climate Program

With reference to Figure 2 (and ˜eks/classes) and starting from a simple "Context" class, "Context_1a" (see ˜eks/demo/site), defined as

```
#ifndef CONTEXT_1A_H
#define CONTEXT_1A_H

#include "context/Context.h"
#include "context/Context_1a_def.h"
class Site_basic;


extern Type* Context_1a_Type_pointer;

class Context_1a : public Context {
protected:
    Site_basic* the_site;   // Site_basic* or derived
public:
    Context_1a(Metaclass* meta, Context_1a_def* def);
    Context_1a(APL* theAPL);
    ˜Context_1a();
```

```
    void Destroy(Boolean aborted);
    void putObject(Boolean deallocate);
    void deleteObject(Boolean deallocate);
    Type* getDirectType();

    Site_basic* site();

    virtual   void simulate();
};

inline
Type* Context_1a::getDirectType()
{
    return TYPE_OF(Context_1a);
};

inline
Site_basic* Context_1a::site()
{
    // cast guaranteed by Metaclass
    return (Site_basic*)the_site;
};

#endif
```

a simple program template may be constructed, via EKStb, defined by

Class	Minimum Derived Class	Class Selected
Context	Context	Context_1a
Site	Site_basic	Site_basic
Climate	Climate_met	Climate_met

In each case the class selection process is assisted by the EKS browser which shows the EKS principal classes and gives information on each.

When the program defined by this template is run it is capable of producing the data as shown in Figure 6a.

By simply changing to another "Context", say "Context_1b (see ~eks/demo/site):

```
#ifndef CONTEXT_1B_H
#define CONTEXT_1B_H

#include "context/Context_1a.h"
#include "context/Context_1b_def.h"
class Site_sun;

extern Type* Context_1b_Type_pointer;


class Context_1b : public Context_1a {
protected:
public:
    Context_1b(Metaclass* meta, Context_1b_def* def);
    Context_1b(APL* theAPL);

    Type* getDirectType();

    Site_sun* site();
```

```
    virtual   void simulate();
};

inline
Type*   Context_1b::getDirectType()
{
    return TYPE_OF(Context_1b);
};

inline
Site_sun* Context_1b::site()
{
    // cast guaranteed by Metaclass
    return (Site_sun*)the_site;
};
#endif
```

the EKS "Metaclass" facility will automatically extend the class selection process so that a template along the following lines might result.

Class	Minimum Derived Class	Class Selected
Context	Context	Context_1b
Site	Site_sun	Site_sun
Climate	Climate_met	Climate_met
Sun	Sun_basic	Sun_basic
Sky	Sky_basic	Sky_basic
Sky_irradiance	Sky_irradiance	Perez
Sky_temperature	Sky_termperature	B&M

Such a program is then capable of producing the same output as in the previous case plus the additional outputs as shown in Figure 6b.

In this way alternative programs can be constructed which are conceptually similar (same "Context") or significantly diverse (different "Context").

Wall Conduction Program

Once a robust model of the site has been achieved, the program developer may wish to develop a fabric conduction model. Such a model is constructed in this section to demonstrate the incremental building and theory substitution aspects of the EKS.

Assuming that the relevant "Context" classes exist - "Context_2a" and "Context_2b" in ~eks/demo for example - the following two templates might follow: the first represents a steady state approach and the second a dynamic approach.

Class Types	Minimum Derived Class	Class Selected
For steady state program:		
Context	Context	Context_2a
Site	Site_basic	Site_basic
Climate	Climate_met	Climate_met
Construction	Construction_basic	Construction_basic
Surface	Surface_basic	Surface_basic

Class Types	Minimum Derived Class	Class Selected
Layer	Layer_basic	Layer_basic
For dynamic program:		
Context	Context	Context_2b
Site	Site_sun	Site_sun
Climate	Climate_met	Climate_met
Sun	Sun_basic	Sun_basic
Sky	Sky_basic	Sky_basic
Sky_irradiance	Sky_irradiance	Perez
Sky_temperature	Sky_temperature	B&M
Construction	Construction_es	Construction_es
Surface	Surface_es	Surface_es
Layer	Layer_es	Layer_es
Conduction	Conduction	Conduction_fd
Solver	Solver	Gauss_column_pivot

The former program will produce outputs such as U-value and steady state heat loss while the latter program is able to produce results as given in Figure 7.

Higher Order Models

In a similar manner models of higher order can be constructed by simply establishing a related "Context" class and then using it to commence the ("Metaclass" controlled) class selection process to define a template. Within the EKS this process is demonstrated for a multi-zone building model (capable of simulating a system such as shown in Figure 8a) and for a plant model (capable of simulating a system such as shown in Figure 8b). In each case the process is demonstrated in ˜eks/demo in the form of Shell Scripts which guide the user through the process from program specification to "X_def" creation and program execution.

Coping with the Technology

A difficult issue which had to be addressed throughout the project was the ability of the research team to understand and apply state-of-the-art technology in the form of OO languages (C++) and OO databases (ONTOS). This section is an attempt to briefly summarise our learning experience in an attempt to give some direction to those who may wish to follow the same course.

As a superset of the C programming language, C++ has much in common with C. This means that individuals who are proficient in C will find getting started with C++ easy. However it is essential to recognise that C++ is object oriented whereas C is not: there is little to be gained by implementing a procedural program in C++. When using an OO language it is necessary to spend a greater proportion of time on the software design stage. For example this phase of the EKS consumed approximately 70% of the projects resources - ie to design the EKS as opposed to programming it. Even then it is probably true that the resource required to design and implement the EKS classes was significantly underestimated. In particular it is the view of the Strathclyde team that in the absence of OO paradigm guidelines, the EKS team overall expended a disproportionate effort on the philosophical issues of class design. With hindsight it would have been better to have continued the rapid prototyping approach adopted for the EKS Prototype in the development of the Demonstrator.

If better OO application guidelines had existed, or if the team’s collective viewpoint had been more coherent, it is likely that the EKS classes which remain unfinished or require modification (because infrastructural aspect changed between the Prototype and Demonstrator versions) would have been completed. This, in turn, would have allowed the EKS to surpass the state-of-the-art instead of merely attaining it. The

lesson then is that:

- Real applications of OO technologies will require substantial resources in order to adequately cover the different facets of the problem.
- OO requires a substantial investment in system design as opposed to implementation.
- It is likely that much of this design resource will be expended on the philosophical aspect of OO class construction in the absence of anecdotal knowledge on ‘best practice’.

Conclusions and Future Work

The EKS project has achieved a number of goals.

- A demonstration system has been developed which supports the construction of a range of models from simplified calculators to state-of-the-art simulators.
- The project has proved the technical feasibility of applying the OO programming approach to a complex engineering domain.
- The EKS provides the means to allow designer participation in the design tool creation process.
- And the project has demonstrated the benefits to be gained from effective collaboration between the IT and domain communities.

There are three principal deliverables from the project:

Firstly, an OO class taxonomy has been progressed to a stage where it can support the construction of programs which exhibit near state-of-the-art characteristics (multi-zone, dynamic, heat and fluid flow). The role of this class taxonomy is to represent the physical entities which comprise a building (rooms, walls, etc.) and the abstract entities which dictate its thermodynamic state (heat transfer theories, numerical methods, etc.). These classes are organised into a ‘used by’ and ‘derived from’ hierarchy and placed under the control of an instantiation mechanism. This means that programs possessing different modelling capabilities can be constructed automatically by merely selecting the required class variants - that is no user coding is required. And because each physical entity within the building has a matched object at run-time, an EKS-produced program will always be matched to the system it is being used to model.

Secondly, these classes have been installed within the ONTOS OODB. The project has therefore lead to a better understanding of the role of the new object oriented technologies (languages and databases) in the development of more powerful design tools and design support environments.

Thirdly, a number of programs - of varying complexity - have been built using the EKS to demonstrate the process. These range from simple sun tracking, through inclined surface solar irradiance to complete multi-zone, heat and flow numerical models.

If the means can be found to build upon the achievements of the EKS then it is possible to envisage a future in which:

- Design tool evolution is undertaken on a task sharing basis because different theoreticians can contribute new classes or modify existing ones.
- The program construction process is more efficient because it can take place in a task sharing manner due to the high level of code reuse.

- The validation process is enabled because individual objects can be tested in isolation, the meaningful connection of objects can be guaranteed and data encapsulation prevents illegal data interference.

- Design tools possess greater realism vis-a-vis the reality while, at the same time, being easier to maintain and evolve.

In the short term it is anticipated that the EKS will be used by those members of the research community who are concerned with advancing the state-of-the-art in modelling buildings and the engineering plant they contain. Already several laboratories have expressed an interest in acquiring the EKS during the initial proving phase. In the medium term it is likely that CAD system vendors and the like will use the EKS to construct and maintain the theoretical 'engine' of future CAD systems. In the longer term it is entirely feasible that end users such as Architects, Control & Environmental Engineers and Energy & Facility Managers will use the EKS to achieve bespoke software solutions for particular problems.

In the longer term the EKS approach will fosters ease of manipulation of methods and ensure that new methods in heat transfer, numerical processing, interface design and the like, as they emerge, will become immediately available to the community of potential users. Model developers - CAD vendors, research organisations and ultimately, perhaps, design practice and legislative bodies - can then select and combine these methods to produce an application model of particular architecture, targeted for a given machine. Because the methods are established as independent, fully documented and tested entities, the program validation and accreditation process, at component and whole-model levels, is greatly assisted.

In developing the EKS, an attempt has been made to anticipate related developments in the field. For example, the emerging international STandard for the Exchange of Product data (STEP: Turner 1990, Gielingh 1990) is likely to have a major impact on building simulation in the medium to long term. Within the standard, real-world entities are described using the STEP language Express, which has many object-oriented features. The principal medium of exchange of data in the construction industry is still text and drawings, with some de facto industry standards for electronic drawings produced on CAD systems. At present, progress is being hampered by the lack of an agreed standard for describing the geometry and topology of a building, and the lack of standard 'libraries' of generic building components. When the STEP standard becomes established for building data, it is envisaged that software (currently being developed) will be used to translate C++ class definitions to and from Express entity definitions. It will then be possible to map a STEP building description in Express into an OODB data structure of C++ objects, with the potential for interface to other software packages such as CAD, lighting design and so on through the neutral format of STEP. This will go some way to removing one of the major barriers to effective interaction between simulation programs; namely the arbitrary and incompatible data structures currently in use. Clearly, an OO programming approach based on real-world entities, as in the EKS, will facilitate development in this area.

The model building tools provided with the EKS are fully functional, but do not have a particularly sophisticated interface. What is envisaged for the future would be a powerful user interface to the facilities provided. The main feature of this interface would be a browser type facility for examining and selecting the classes making up the program, together with help and guidance on the capabilities and validity of the various classes. There would also be facilities to display the resultant program graphically, and to modify the program by replacing/adding classes.

While the EKS demonstrator has provided proof of concept, several further developments can be identified in order to evolve the system to its full potential.

User Support

In the short term there is a need to subject the EKS demonstrator to field trials to ensure that the conceptual basis is sound and that potential users understand the benefits the technology might bring. Clearly such trial users groups would require help and support, particularly in the early stages. To be fully effective, this support should comprise both software engineering and domain expertise. It must also be sufficiently long-

term to give users the confidence to embark on real projects based on the EKS demonstrator.

EKS Enhancement

As a concept demonstrator, there are a number of issues that have not been fully addressed by the EKS project to date, either due to the lack of resource/time (eg a user interface) or deliberately left to Phase II (eg integration with STEP). There are also some further system issues ('intelligent' Metaclasses, use of parallelism) that arise directly from the system design. Addressing these issues is crucial to the long term viability of the EKS. This work could probably best be carried out as a series of collaborative mini-projects with carefully defined objectives.

EKS Validation

The validation capabilities provided by the EKS demonstrator should be used to verify and validate the system, classes and templates of the EKS itself. The extent of the EKS provision with respect to validation is described elsewhere (Hammond et al 1992).

EKS Uptake

The EKS concepts clearly have applicability in other domains. Other disciplines could be encouraged to explore the EKS with a view to extending its class taxonomy to other domains such as lighting and acoustics. One initial step in this direction could be to develop a robust and comprehensive class library from accredited EKS classes, possibly with international co-operation (COMBINE, STEP).

EKS Spinoff

Finally, as a major new infrastructure platform, the EKS should give rise to significant further research, such as exploring new simulation techniques, developing new types of simulation tool and exploring new software engineering paradigms.

References

Charlesworth P, Clarke J A, Hammond G, Irving A D, James K, Lockley S, Mac Randal D F, Tang D, Wiltshire T J and Wright A J (1991) 'The Energy Kernel System: The Way Ahead ?' *Proc. of of Building Environmental Prediction '91* Canterbury, England.

Clarke J A (1987) 'The Future of Building Energy Modelling in the UK', *Report to the Building Sub-Committee*, Science and Engineering Research Council, Swindon.

Clarke J A (1988) 'The Future of Building Energy Simulation: The Energy Kernel System' *Energy and Buildings* V10(3), pp.259-66.

Clarke J A, Mac Randal D F, Powell J A, Wiltshire T J (1988) 'The Energy Kernel System' An Overview Submitted to the Building Sub-Committee of SERC in Support of Five Grant Proposal, Energy Simulation Research Unit, University of Strathclyde, Glasgow.

Clarke J A, Lockely S, Mac Randal D and Wiltshire J (1988b) 'An Object-Orientated Approach to Building Performance Modelling' *Proc. USER1* Ostend, Belgium.

Clarke J A, James K and Tang D (1989) 'Simulation Methods Concerning Building Performance Prediction: A General Review' *EKS Project Paper* Energy Simulation Research Unit, University of Strathclyde, Glasgow.

Clarke J A, Charlesworth P, Hammond G, Irving A D, James K, Lockley S, Mac Randal D F, Tang D, Wiltshire T J and Wright A J (1990) 'Presentation to the Steering Committee' *EKS Project Paper* Energy Simulation Research Unit, University of Strathclyde, Glasgow.

Clarke J A, James K and Tang D (1990b) 'EKS Prototype Status Review and Issues Arising' *EKS Project Paper* Energy Simulation Research Unit, University of Strathclyde, Glasgow, October

Clarke J A and Mac Randal D (1991) 'An Intelligent Front-End for Computer-Aided Building Design', *Artificial Intelligence in Engineering*, Computational Mechanics Publications, Vol 6, No 1, pp.36-45.

Clarke J A, Charlesworth P, Hammond G, Irving A D, James K, Lockley S, Mac Randal D F, Tang D, Wiltshire T J and Wright A J (1991) 'The Energy Kernel System' *Proc. of of Building Simulation '91*, pp.313-22, Nice, France.

COMBINE (1992) 'The COMBINE Project' Various publications available from F Augenbroe, Faculteit der Civiele Techniek, Delft University of Technology, Delft, The Netherlands.

Editor F H and Lochovsky K W (1990) *Objected-Oriented Concepts, Databases, and Applications* ACM Press, New York.

Hammond G and Irving A D (1992) *EKS Project Final Report* Science and Engineering Research Council, Swindon.

ONTOS (1989) *Object Database Documentation* Ontologic Inc, Three Burlington Woods, Burlington, MA 01803, USA.

Stroustrup B (1987) *The C++ Programming Language* Addison-Wesley Publishing Company Inc.

Tang D (1989) 'Simulation Methods Concerning Building Performance Prediction: A General Review' *EKS Project Paper* Energy Simulation Research Unit, University of Strathclyde, Glasgow.

Tang D (1990) 'The EKS Theory Representation' *EKS Project Paper* Energy Simulation Research Unit, University of Strathclyde, Glasgow.

Tang D (1990b) 'An investigation into the Intrinsic Relationships between Graph and Matrix Representation of Building Physics in the Context of Energy Modelling' *EKS Project Paper* Energy Simulation Research Unit, University of Strathclyde, Glasgow.

Tang D (1991) 'The Generalised System Solution Classes in the EKS Environment' *Proc. of Building Simulation '91*, pp.323-27, Nice, France.

Wright A J, Charlesworth P, Clarke J A, Hammond G P, Irving A, James K A, Mac Randall D and Tang D (1990) 'The use of Object-Oriented Programming Techniques in the UK Energy Kernel System for Building Simulation' *Proc. European Simulation Multiconference*, pp.548-52, Nuremberg, Germany.

Wright A J, Wiltshire T J and Lockley S (1992) *EKS Project Final Report* Science and Engineering Research Council, Swindon.

Appendix One: EKS Demonstrator Class Types

The following table lists the principal and intrinsic classes of the EKS Demonstrator and gives for some classes a brief description. Complete details on each class can be found in ~eks/classes where the class header files (?.h) define the encapsulated data and functions and the ?.C files contain the function implementation.

Class	Description
Principal Classes	
building:	
"AirVolume"	Represents an air filled space.
"AirVolume_def"	Supply data for the "AirVolume" e.g. space coordinates and thermo-physical properties.
"AirVolume_es"	Energy conservation of room air: returns list of coefficients of differenced air node equation.
"AirVolume_es_def"	Supply data for the "AirVolume_es" e.g. pointers to convection theories.
Building	Representation of a building.
Building_def	Supply data for "Building" e.g. contiguity of room constructions.
"Building_es"	Generates a complete set of equations for a building based on various thermal theories.
"Building_es_def"	Supply data representing a "Building" e.g. space information.
"Construction_basic"	Representation of a multi-layered construction.
"Construction_basic_def"	Supply data for a "Construction_basic" e.g. number of layers.
"Construction_es"	Energy conservation of multi-layered construction: concatenates the conduction equations generated by layers.
"Construction_es_def"	Supply data for the "Construction_es".
"Convection_fd"	Natural convection heat transfer coefficients evaluated as a function of surface to air temperature differences.
"Convection_fd_def"	Supply data for the "Convection_fd" e.g. pointers to air volume and surface pairs.
"HeatSource"	Generates a heat source.
"HeatSource_def"	Supply data for the "HeatSource" e.g. scheduled data and convection/ radiation split.
"Layer_basic"	Representation of a constructional layer.
"Layer_basic_def"	Supply data for the "Layer_basic" e.g. thickness, material.
"Layer_es"	Energy conservation of a layer: creates the conduction equations for a homogeneous layer based on various theories.
"Layer_es_def"	Supply data for the "Layer_es" e.g. pointer to the conduction theory.
"Room"	Representation of a room.
"Room_def"	Supply data for "Room" e.g. list of surfaces and heat sources.
"Room_es"	Energy conservation of a room: creates the equation set for a room including conduction, convection, radiation and heat heat generation.
"Room_es_def"	Supply data for "Room_es" e.g. pointers to the alternative conduction, convection and radiation theories.
"Space"	Representation of a geometrical space.
"Space_def"	Supply data for the space e.g. pointer to an "Air_volume".
"Surface_basic"	Representation of a surface.
"Surface_basic_def"	Supply data for "Surface_basic" e.g. thermal resistance of surface layer.
"Surface_es"	Generates the surface energy balance equation.
"Surface_es_def"	Supply data for the "Surface_es" e.g. pointer to the related "Construction".
"Surface_es_sw"	Complements the room equation set by "adding in" surface shortwave gain to conservation equations.
"Surface_es_sw_def"	Supply data for "Surface_es_sw" e.g. pointer of the radiation object.

Class	Description
Principal cont'd	
"Context":	Establishes a problem context by instantiating one or more of the following classes: Site, Building, Plant, Control and Solver.
"Context_1a"	Problem context for simple site model generation.
"Context_1a_def"	Supply data for "Context_1a" e.g. pointer to the simple site object.

Class	Description
"Context_1b"	Problem context for detailed site model which incorporates solar radiation.
"Context_1b_def"	Supply data for "Context_1b" e.g. pointer to the detailed site object.
"Context_2a"	Problem context for simple multi-layer construction model generation.
"Context_2a_def"	Supply data for "Context_2a" e.g. pointer to the detailed site object.
"Context_2b"	Problem context for detailed multi-layer construction model which incorporates site, dynamic conduction and solver.
"Context_2b_def"	Supply data for "Context_2b" e.g. pointers to objects of site, construction, conduction and solver.
"Context_3a"	Problem context for simple building model generation.
"Context_3a_def"	Supply data for "Context_3a" e.g. pointers to objects of site and building.
"Context_3b"	Problem context for detailed building simulation model which incorporates site, theories for conduction, convection and radiation, shortwave distribution. solver, etc.
"Context_3b_def"	Supply data for "Context_3b" e.g. pointers to objects of site, building, theory objects and solver.
"Context_pb"	Problem context for the creation of algorithmic type (TRNSYS) plant model.
"Context_pb_def"	Supply data for "Context_pb" e.g. pointers to the plant components.
"Context_pc"	Problem context for the creation of state-space type (ESP-r) plant model.
"Context_pc_def"	Supply data for "Context_pb" e.g. pointer to the plant components.
"Control":	There are no control classes in the EKS Demonstrator at this time.

Class	Description
Principal cont'd	
"Plant":	Classes to handle plant components and connections. There are two types of classes: those for sequential type modelling (eg TRNSYS like) and those which support simultaneous type models (eg ESP-r like).
"Component"	Generic class for system components.
"Component_def"	Supply data for "Component" e.g. its name.
"Connection"	Generic class for system connections.
"Connection_def"	Supply data for "Connection" e.g. a name and the names of source and target components.
"Context_pa"	Problem context for the creation of the system topology.
"Context_pa_def"	Supply data for "Context_pa" e.g. pointer to the topological system.
"System"	Generic class for system description.
"System_def"	Supply data for "System" e.g. list of components, list of connections and target components.
sequential types:	
"ABoiler"	Generates an algorithmic type steady-state boiler model.
"ABoiler_def"	Supply data for "ABoiler" e.g. thermo-physical properties of water.
"ACard_reader"	Generates source data required by the system model.
"ACard_reader_def"	Supply names of the data specified.
"AComponent"	Generic class for algorithmic type system component.
"AComponent_def"	Supply data for "AComponent" e.g component name.
"AConnection"	Creates a multi-variable connection between two components.
"AConnection_def"	Supply data for "AConnection" e.g. the number of variables and the connection pairs.
"APrinter"	Generates an algorithmic type output device.
"APrinter_def"	Supply data for "APrinter_def" e.g printer type.
"APump"	Generates an algorithmic type, steady-state pump model.
"APump_def"	Supply data for "APump" e.g. nominal water mass flow rate.
"ARadiator"	Generates an algorithmic type, steady-state radiator model.
"ARadiator_def"	Supply data for "ARadiator" e.g. UA value and properties of working fluid.
"ARoom"	Generates an algorithmic type, steady-state room model.
"ARoom_def"	Supply data for "ARoom" e.g. UA value of the walls.
"ASystem"	Creates an algorithmic type system.
"ASystem_def"	Supply data for "ASystem" e.g. name of the system, lists of components and connections.
"Ideal_pipe"	Generates an algorithmic type, steady-state pipe model.

Class	Description
"Ideal_pipe_def"	Supply data for "Ideal_pipe" e.g. diameter.
"Type_5"	Generates an algorithmic type heat exchanger model based on TRNSYS Type No. 5.
"Type_5_def"	Supply data for "Type_5" e.g. mode of operation, UA value and hot and cold side water specific heat.
simultaneous types:	
"Ac_building"	Generates a state-space, 2-node building model.
"Ac_building_def"	Supply data for "Ac_building" e.g. dimensions, thermo-physical properties, heat transfer coefficients, heat generation, etc.
"Ac_coil"	Generates a state-space, 3-node cooling coil model.
"Ac_coil_def"	Supply data for "Ac_coil" e.g. dimensions, thermo-physical properties of working fluid, etc.
"Ac_duct"	Generates a state-space, 1-node air duct model.
"Ac_duct_def"	Supply data for "Ac_duct" e.g. dimensions, thermo-physical properties of air, etc.
"Ac_fan"	Generates a state-space, 1-node fan model.
"Ac_fan_def"	Supply data for "Ac_fan" e.g. overall efficiency, power, mass, etc.
"Ac_mixbox"	Generates a state-space, 1-node air mixing box model.
"Ac_mixbox_def"	Supply data for "Ac_mixbox" e.g. overall mass, specific heat, etc.
"Boundary"	Generates boundary condition for "Esystem".
"Boundary_def"	Supply data for "Boundary" e.g. type of each boundary.
"EComponent"	Generic class for state-space type components.
"EComponent_def"	Supply data for "EComponent_def" e.g. component name.
"EConnection"	Creates a multi-variable connection between two components.
"EConnection_def"	Supply data for "EConnection" e.g. the the connection pairs for the state-variables of source and target components.
"ESystem"	Creates a state-space type plant system.
"ESystem_def"	Supply data for "ESystem" e.g. name of the system, lists of components and connections.

Class	Description
Principal cont'd	
"Site":	
"Berdahl_and_Martin"	Calculation of sky temperature by the Berdahl_and_Martin model.
"Berdahl_and_Martin_def"	Supply data for for the "Berdahl_and_Martin" e.g. degree of cloud cover.
"Climate_met"	Create a climate data set.
"Climate_met_def"	Supply name of climate data set.
"Perez"	Establish the sky irradiance by the Perez model.
"Perez_def"	Supply data for "Perez" e.g degree of anisotropy.
"Site_basic"	Creates a basic site to represent climatic data.
"Site_basic_def"	Supply data for for the "Site_basic" e.g site name and location.
"Site_sun"	Creates a site to represent climatic data and solar irradiance.
"Site_sun_def"	Supply data for for the "Site_sun" e.g site name and location.
"Sky_basic"	Creates a basic sky to represent sky irradiance and temperature.
"Sky_basic_def"	Supply data for for the "Sky_basic" e.g. sky type.
"Sun_basic"	Calculates sun position using the Michalsky method.
"Sun_basic_def"	Supply data for for the "Sun_basic" e.g. time zone.
"Solver":	
"Gauss_column_pivot"	Solve the simultaneous equations using the Gauss column pivot method.
"Gauss_seidle"	Solve the simultaneous equations using the Gauss-Seidle iterative method.
"Lu_factorise"	Solve the simultaneous equation set using the LU factorisation method.
theory:	
"Conduction_fd"	Finite difference model of unsteady heat conduction using central differencing of spatial coordinates.

Class	Description
"Convection_fd_1"	Supplement system equation set by creating convective couplings using simplified convective heat transfer coefficient.
"Radiation_fd_1"	Room internal longwave radiative heat exchange between surfaces.
"Shortwave_beam"	Generates the shortwave irradiation os a surface.
"Shortwave_beam_def"	Supply data for the "Shortwave_beam".
"Shortwave_smear"	Generates the shortwave irradiance distribution within a space.
"Shortwave_smear_def"	Supply data for the Shortwave_beam e.g. irradiances from outside and from adjacent rooms and pointers to surfaces.

Class	Description
intrinsic classes	
data:	
kew67	Original climatic data for Kew 1967.
"X_def"	Pointer to data object containing information defining X.
dimensions:	
"Dimension"	Base class for dimension types which provide operation and type security for the following 21 types.
"Angle"	
"Area"	
"Density"	
"Diffusivity"	
"Energy"	
"Length"	
"LuminousIntensity"	
"Mass"	
"NonDimensional"	
"Power"	
"Proportion"	
"Quantities"	
"Quantity"	
"SpecificHeatCapacity"	
"Speed"	
"Temperature"	
"ThermalConductivity"	
"ThermalResistance"	
"Time"	
"Viscosity"	
"Volume"	
infrastructure:	
"EKSObject"	Base class for all EKS classes.
"Error_handling"	Basic error handling.
"Metaclass"	Class selection control facility.
"Metaproto"	Used to make new "Metaclass" classes.
"Template"	High level description of program composition.
transport:	
"ClimateRecord"	Encapsulates climatic data.
"ClimateRegime"	Encapsulates climatic data and solar irradiances on south, north, west, east and horizontal surfaces.
"ClimateSet"	Creates a set of climate data records.
"Coefficient"	Encapsulates type, value and corresponding state-variable of a coefficient.

Class	Description
"Equation"	Encapsulates a list of coefficients and a self-coupling state-variable.
"Equation_set"	Encapsulates a list of "Equation".
"FilledVolume"	A volume filled with a specified "Substance".
"Finish"	Finish type of a surface e.g. emissivity, reflectivity, roughness and colour.
"Irradiance"	Encapsulates components of the solar irradiance with reference to the orientation.
"Location"	Location of a site expressed in terms of latitude and longitude difference from the reference meridian.
"Matrix"	A two dimension matrix.
"Orientation"	Defines the orientation of a surface in terms of azimuth, elevation and tilt angle.
"PlaneEquation"	Equation of a plane.
"Polygon"	Coordinates of a polygon.
"Profile"	A set of time series data to represent a profile.
"State_variable"	Equation element containing type and value of a state variable.
"State_vector"	A list of state variables.
"Substance"	Encapsulates thermo-physical properties of a "Material".
"Sun_position"	Encapsulates sun position in terms of azimuth, elevation and declination.
"Time_of_day"	Represents time in year, month, day, hour, minute and second format.
"Vector"	One dimensional vector and operations.
"Vertex"	X, Y, Z coordinates of a vertex in 3D space.
Developed but not fully integrated in the EKS Demonstrator:	
flow:	Classes to represent flow networks.
"Flow_arc"	
"Flow_arc_def"	
"Flow_network"	
"Flow_network_def"	
"Flow_node"	
"Flow_node_def"	
"Type010_flow"	
"Type010_flow_def"	
"Type015_flow"	
graph_theory:	Classes to handle graph theoretic techniques.
"Graph"	
"Graph_arc"	
"Graph_node"	