#5362

THE ENERGY KERNEL SYSTEM

Charlesworth P%, Clarke J A#, Hammond G%, Irving A@, James K#, Lee B@
Lockley S, Mac Randal D@, Tang D#, Wiltshire T J*, Wright A J**

ABSTRACT

The Energy Kernel System (EKS) project has reached the final year of its three year duration. The *modus operandi* has been designed, a class taxonomy devised and the software implementation process commenced. This paper describes the elements of the EKS as it is now envisaged and elaborates on the object oriented programming (OOP) approach being employed in its construction. In particular, the form and content of the EKS classes, the role of inheritance and the scope of the in-built domain theories are elaborated.

INTRODUCTION

Between 1979 and 1985, the 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 reviewed the future of building energy modelling and its relationship to computer-aided building design tools. 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 which would foster the collaborative development of the next generation of performance assessment programs. The intention was that this environment would form the nucleus, or kernel, of future model building activities, permitting the rapid prototyping of any program and facilitating a coherent approach to development, validation and maintenance of such programs. This environment has subsequently become known as the *Energy Kernel System* (EKS) (Wright et al 1990; Charlesworth et al 1991). 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.
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.

This paper focusses on the form and content of the EKS. For information on related work in the field, and the benefits which are may accrue for the practitioner, the reader is referred elsewhere (Sowell 1991).

THE EKS IN OUTLINE

Figure 1. shows the architecture of the EKS. At the core of the system lies an object oriented database (OODB), ONTOS (1989), which has knowledge of entities such as climatic data and material properties encapsulated as persistent objects. The OODB also has knowledge of the interface specification of the various EKS building blocks, called classes, which can be used to construct a wide range of program architectures. These classes are an encapsulation of an entity's description (in the form of data) and behaviour (in the form of functions). The OODB also contains the usual support data – climate, material properties, test sets and so on – in the form of persistent objects.

Most Program Builders will, at some time or other, want to modify or replace specific classes supplied

% School of Architecture and Building Engineering, University of Bath
* School of Architecture, University of Newcastle
# Energy Simulation Research Unit, University of Strathclyde
@ Informatics Department, Rutherford Appleton Laboratory

with the EKS. Provided the interface of the class does not change, this is achieved by deriving a new class from the existing EKS one, notifying the EKS of its existence and referencing it from the EKS' user interface called the Template.

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), in plant systems (components, connections, controllers) and in programs (numerical solvers, time control). An in-built inheritance mechanism gives the EKS user access to alternative implementations of these classes where appropriate and supports the addition of new implementations.

The EKS template is a framework for defining a particular program which can then be stored in the OODB 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.

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 buiding 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 (Bjork 1990). 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 OOP approach based on real-world entities, as in the EKS, will facilitate development in this area.

EKS USER TYPES

There are three separate levels at which users interact with the EKS, corresponding to 3 user stereotypes. Of course, real users will almost certainly range across these stereotypes at different times, but they are useful for illustrative purposes.

EKS Developer/ Extender

This user type will be concerned with developing/ extending the EKS class taxonomy and its associated software tools (see Template section). While the EKS, as released, will provide an extensive class taxonomy, it will be necessary, in the future, to enhance the functionality of existing classes, provide alternative implementations of existing functionality or extend the taxonomy by adding completely new classes. This stereotype breaks down into two levels, minor modifications/ extensions to existing classes, for example deriving from an existing class and replacing some functions, 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, for example the ONTOS database and Unix.

Program Builder

Users within this category will, typically, use the Template to select the required classes from the overall class taxonomy as known to the OODB at any time. In-built within the taxonomy is the knowledge of class dependencies. For example, a "Room" class 'knows' that it can use (among others) an "Air_volume" class 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 the "Air_volume" class. The selected "Air_volume" is then checked to ensure that it provides at least the functionality that the selected "Room" class expects. After all the required classes have been identified, the program specification is stored in the OODB. This is further described in the sections on the Template and Metaclasses. 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 Model 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 CAD packages or intelligent front-ends. For this reason only a rudimentary problem description interface facility will initially be provided with the EKS itself - more sophisticated variants, for example based on emerging Intelligent Front-End systems (Clarke and MacRandal 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" class, 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, alternatively, they can be transferred to disc files or displayed directly. Since they interact only with the "Context" class run-time interface, these users will not be able to distinguish EKS built programs from the current hand built versions.

CLASS TAXONOMY

The principal entity of the EKS is its underlying class taxonomy which comprises the object types from which alternative programs can be built. Figure 2 shows this taxonomy in terms of the principal class types from which programs are built. The diagram also shows the object control paths and the domain theory class types. Note however that the diagram does not show those classes which are internal to the EKS and are used to encapsulate the data associated with message passing. Neither does it identify the different variants (or abstraction levels) of these principal and domain theory classes which the EKS offers.

Each principal class in the taxonomy is thus a generic class, capable of being implemented in many different ways. For example, the Conduction class, whose function is to represent the conduction process, can be implemented in different ways, depending on whether the conduction is to be modelled, for example, as a steady state process, a response function process, a finite volume time varying process. These variants of the principal class can be organised into a class hierarchy, such as that shown in Figure 3. The class hierarchies can be considered to be orthogonal to the class taxonomy, with a different class hierarchy rooted on each class in the taxonomy. Within the hierarchy, the object oriented inheritance mechanism is used to reduce coding and improve reliability. Inheritance means new derived classes need only add code for the extra functionality they are providing since they automatically use the parents' code, presumably already "validated" to some extent, for their inherited functionality.

Of course, inheritance is also used outside the above class hierarchies, for the same reasons. Here the classes being developed have a much wider spread of functionality, for example "Room", "Site" and "Matrix". There is no single "best" way to structure these classes, and any structure for a real-world system will be biased towards a particular view, such as thermal simulation. However, a good structure should be adaptable for other areas (by using the more general base classes), and be easy to use and extend. As an example, consider the classes derived from "Physical_entity" as shown in the figure:

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).

The EKS must be able to configure its classes in many different ways in order to support the building of alternative program types. Since there are many permutations and combinations of data and functions, it is not possible, or necessary, to include all the resultant classes in the EKS. For example, if there is a choice for the "Room" class of four types of "Convection", three types of "Occupant", two types of "Equipment" and two types of "Window", there are potentially 48 (4 x 3 x 2 x 2) different "Room" classes. On the other hand, many classes will be suitable for all, or a large number, of different programs – for example "Location", "Polygon", "Material" and "Layer". In the EKS Metaclasses are used to configure new classes according to the requirements of the program builder. This means that the more complex classes, such as "Room", "Building", "Site" and "Climate" can, if required, be specialised for a particular program or set of programs. In addition, since not all combinations of classes are possible (response function heat conduction calculations might not be meaningful when used with a degree-day representation of climate), rules are built into the EKS Template to prevent such combinations.

The design of the EKS taxonomy, which is now complete, involved a three stage process as follows. Firstly, the domain (building energy modelling) was decomposed into a number of primitive functions such as sun position calculation, heat transfer theory generation, equation solving, polygon operations and data processing.

Secondly, for each of these functions, the data requirements were determined. For example, a conduction function requires thermophysical properties which depend on the level of implementation of its mathematical model, while a matrix inversion function requires the topology and coefficient values (or the means to determine such values) for a set of equations.

Finally, the functions were grouped into classes in a way which ensured that the data members of each class, as required to support its functions, could be guaranteed to be available at run-time to the object made from the class. For example, the "Layer" class possesses a function to calculate its thermal resistance which requires that the class has thickness and conductivity as its data, while the "Room" class possesses a function to determine its shortwave response which requires it to have room geometry and knowledge of its

surfaces, and hence surface finishes, as part of its private data. More interestingly, perhaps, while the "Construction" class possesses a function to determine an overall, reference U-value (which has prescribed surface resistance values), it does not contain a function to calculate the effective U-value. Instead, the "Building" class, which has the data members to support the calculation (exposure, surface finish, convective regimes, sky radiation), contains the function. In this way the data requirements of functions are satisfied by the private data of the objects which contain them and by the data supplied by other objects whose existence is assured by the Template.

The significance of the approach is that, at run-time, object control is ordered with the control flow acting down the taxonomy hierarchy. Anyone 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 OOP approach where it is the responsibility of the function (sometimes called a method) to get its data. To do this the function would need to know about its context.

THE TEMPLATE

The EKS will initially consist 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 components shown on the taxonomy diagram, and there will be potentially several alternative implementations of these components, 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, that is its Template, for future use.

The Template for a particular simulation program is actually 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, (actually carried out by the Template class constructor function) there is an opportunity to prevent the attempted use of illegal or incomplete combinations of EKS classes. In order to be able 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 has been provided to hold this and related information. Metaclasses are described in more detail in the following section.

Of course, the end user is not interested in merely having a specification for a simulation program, they actually 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. EKS class instantiation is achieved using the Metaclass mechanism. 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.

The Template class provided with the EKS Demonstrator will be a full functional version, but will not have a particularly sophisticated interface. What is envisaged for the future would be a powerful user interface to the facilities provided by the template. 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 Finally, this user interface should also provide a "validation console", that is a mechanism by which the various validation tools supplied with the EKS could be "attached" to the appropriate places in the program.

In summary, 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 specified simulation program to their specific problem.

METACLASSES

The power and flexibility of the "mix & 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, akin to ensuring there are no compilation errors in a conventional program. In OOP, 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 Metaclasses 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 (termed MetaLayer) 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.

In summary, a Metaclass has three main tasks:

- To hold information about the 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.

In relation to this object creation control mechanism, two points are relevant. Firstly, the Metaclass of a class which contains state variables may not instantiate a class which does not. This is because classes with state variables – for example "Layer" and "Surface" – must have an instance for each physical occurrence. Other classes, such as "Conduction" and "Finish", present the possibility of shared instances. Such objects are instead created by MetaContext which has access to the 'options' data defined by the model builder via the template. Secondly and typically, objects will instantiate more than one class. This is necessary in all cases where the dissemination of object pointers must be synchronised. For example "Building" instantiates one "RoomLinker" and several "Room" classes as required. This order is important because it is the "RoomLinker" object's responsibility to make the required number of "Construction" objects. Only then can "Building" make its "Room" objects, passing them their "Construction" object pointers as it does so.

DOMAIN THEORIES

At present 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 only to those specific 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 physical 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 physical 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. For example, Figure 4 shows several derived classes of the Conduction class. Note that the Finite_difference implementation is the full three-dimensional case and can be made to reduce to the two- and one-dimensional case or, in the limit, to the steady-state case. In the approach, the C++ 'derived class' and 'virtual function' constructs are used

(Stroustrup 1987). For example the Conduction base class comprises virtual functions whose signatures specify the interface of the derived classes. The derived classes – that is the different implementations of Conduction – reimplement the virtual functions. The base class will never be directly used. The specification of the base class ensures that the derived classes have the same interface.

As an example of the process, consider the Conduction class. 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.

All internal equation–based theories within the EKS are handled in the form of 'vectorised state–equations'. For example, a spatially discretised wall in a room may be represented in terms of its temperature function T by equation 1:.

$$\begin{cases} c_1\dfrac{dT_1}{dt} = h_o(T_o-T_1) + R_1(T_2-T_1) + \sigma\varepsilon_1(T_{so}^4-T_1^4) \\ c_2\dfrac{dT_2}{dt} = R_1(T_1-T_2) + R_2(T_3-T_2) + q_2 \\ \cdots \quad \cdots \quad \cdots \\ c_n\dfrac{dT_n}{dt} = h_i(T_i-T_n) + R_{n-1}(T_{n-1}-T_n) + \sigma\varepsilon_n(T_{si}^4-T_n^4) \end{cases} \qquad (1)$$

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 (for example

In the EKS, this equation–set is represented using the vector symbolism as shown in equation 2.

$$C\,\frac{d}{dt}\,T(t) = A(T,U)\,T(t) + B(T,U)\,U(t) \qquad (2)$$

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

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

in which $a_{11} = -(h_o+R_1+h_{ro})$, $a_{12} = -R_1$, $a_{21} = -R_1$, ...,

$a_{nn} = -(h_i+R_{n-1}+h_{ri})$; $h_{ro} = \sigma\varepsilon_1(T_{so}^2+T_1^2)\,(T_{so}+T_1)$,

$h_{ri} = \sigma\varepsilon_n(T_{si}^2+T_n^2)\,(T_{si}+T_n)$,

$T(t) = [T_1(t), T_2(t), \ldots, T_n(t)]^T$,

$B = [b_{i,j}]$; $(i = 1,2,\ldots,n;\ j = 1,2,3,4)$

and $U(t) = [T_o, T_{so}, q^o, T_i, T_{si}]^T$.

Note, that there is no transformation from equation 1 to 2, only the grouping of the variables are different.

Theory in the form of equation 2 is manipulated via five purpose–built classes: Equation_set_list, Equation_set, Equation, Coefficient, and Region_id. Figure 5, for example, shows schematically the use of these encapsulation classes. In the case of layer conduction, each individual conduction equation – a steady state layer will possess one, a one–dimensional finite difference formulation one per node in a normal direction – will be encapsulated within an Equation object and will refer to a unique region. Equation objects contain Coefficient objects as private data and these Coefficient objects contain, in turn, the coefficient values and state variable identifiers. Finally, Equation objects are encapsulated within Equation_sets for each physical class and domain theory. These Equations_sets are then passed up the class taxonomy from Layer to Context so that related Equation_sets (for other domain theories and physical regions) may be appended. In this way composite or homogeneous equation structures are passed to the Solver where they may be transformed into one of several formats for integration. For example, the Solver may create a matrix equation by using the Equation's Region_id to define the row and the Coefficient's Region_id the column. In effect this theory encapsulation ensures that the returns from all equation–based theory classes can be handled in a coherent manner and provides the possibility for the Solver class to be involved at any level of the problem building process.

In its final form the EKS will offer a range of theories for each topic – conduction, convection, air flow, shortwave & longwave response, casual gains, control behaviour and so on. Furthermore these theories will span from a lowest order approach – the time invariant, user specified case for example – through intermediate order formulations, to 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.

Based on the above theory encapsulation, the Solver class is able to offer a wide range of numerical integration techniques and equation manipulation tools.

System operator classes support the fundamental operations for matrix system: for example, vector inner product, matrix product, vector norm, matrix determinant, matrix inverse, pseudo–inverse, singular value decomposition, matrix eigen–values and matrix eigen–vectors.

System solution classes include solvers for linear or non–linear differential equations as well as algebraic equations. For the case of differential, stiff systems, the Gear and implicit Rung Kutta method are two of a range of methods on offer; for linear, non–stiff systems (and to ensure generality of approach), methods such as explicit finite difference are also supported. For the case of algebraic systems, various elimination methods and iterative methods are offered. In particular, the EKS possesses methods which store only the non–zero entries of the system matrix and create minimal non–zero entries during the elimination process.

System optimisation classes provide tools for equation manipulation including sorting, grouping and partitioning which can be used to organise the complete system in support of efficient solution. To achieve this,

algebraic theoretical methods, such as linear and non-linear programming techniques and gradient searching methods, are offered. On the other hand, graph theoretical methods can be used to the same end and are also available. This includes methods to locate the minimal (maximal) tree from a system network, to locate the minimal cut-set for a system and therefore to determine an optimal solution sequence. The EKS also provides the means to sort a large system network into quotient trees of subsystems and thereby to ensure that the independent solution of the subsystems is identical to the solution of the complete system. This technique will have particular relevance in the case of models which attempt the simultaneous solution of the building energy and 3-D turbulent air flow system.

VALIDATION IN THE EKS CONTEXT

The EKS is a model building environment and as validation is being considered from the outset several validation features are being incorporated into the system. Consider analytical testing: traditionally this involves extracting the algorithms from the main body of the code and testing them in isolation. It is often difficult to ensure that the algorithm then behaves as it would when part of the whole model. The OOP nature of the EKS enables specific processes to be individually examined in this manner as they already exist as distinct entities. In order to develop object validation procedures, internal convection has been chosen as an exemplar. This has enabled the development of an accreditation methodology which can then be applied to a growing number of EKS objects in future. A practical manifestation of this work is a standard "header block" for each class which indicates the type and outcome of any validation procedures previously applied to the object.

One of the main features of the EKS is that a validation infrastructure will be in place and available for use by future model developers. This infrastructure, in the form of a tool kit, will allow validation, verification and quality assurance to become a natural part of the model construction and proving process. In essence, the tool kit allows the time series data produced by a program to be quantitatively compared against other data produced by experimental or theoretical means. In addition, a user is able to establish the sensitivity and accuracy of the solutions produced by the program (Irving 1987). These validation tools may also be used within the program at run-time to perform mass or energy balance checking and to monitor the program's performance when instabilities are encountered.

Central to validation, verification and quality assurance is the estimation and application of statistical measures. Qualitative measures such as graphical comparison are subjective; however, their attributes are complementary to the quantitative methods and should be used in conjunction. Commonly used statistical measures are, for example, the mean value, the square residuals and the variance. Such measures are available within the EKS and can be used as the basis of the testing of a statistical hypothesis where the testing is the process of inferring from the data if the hypothesis is to be accepted or not[#].

As an example, it may be desirable to perform an empirical verification on an output parameter from an EKS program. This would require that, firstly, available and appropriate empirical data sets be obtained (note that the EKS offers several such data sets). The algorithm is then executed and the mean difference and variance between the solution and the experimental data are estimated. The ratio of mean difference to variance forms the basis of a matched pair students t-test which may then be used to decide if the algorithm represents reality within a predefined confidence interval – for example the 20:1 level adopted in many areas of engineering (Kline and McClintock 1953, Moffat 1982).

Clearly, the statistical test may fail for many reasons but in overall terms it is usually because of errors in the empirical data set or errors within the algorithm, presuming that the input data to the program is correct. If the discrepancies are being caused by errors in the program, identifying the cause is usually unique to the code and very difficult in large complex simulation models. However some classes of problems may be resolved using the diagnostics generated when the methodology is used. Some examples are reported elsewhere (Irving 1987).

In addition to the single parameter testing procedure outlined above, the EKS validation tool kit also has methods for whole model testing. These methods are based on multivariate statistical procedures and are in principle the same as the single parameter case (Balin and Sargent 1982, Irving 1987).

The whole model and single parameter testing procedures discussed so far are essentially off-line methods and may be used independent to the EKS program construction process. However, as mentioned above, the tool kit may also be used in EKS models themselves as run-time validation techniques. This practice can be illustrated by considering the run-time energy or mass balance checking. The tools for the sum, the sum of squares and statistical comparison would be included within the program to determine the square residual for some mass balance at each time-step. The t-test is a more consistent and efficient test than the chi squared test and may be nonparametric. In addition the Students t-test may be generalised as the Hotelling's T^2 test and applied to the whole model at run-time or off-line (Irving 1987, Balin et al 1982).

Dimensioning is another issue being addressed. Many errors in computer programs dealing with physical processes result from the fact that in many languages, for example FORTRAN or C, dimensioned quantities are typed as real numbers. This makes it difficult to spot errors in equations where, for example, the dimensions of the right hand side do not match the dimensions of the left hand side.

In the present context, the hypothesis, typically, is that the program adequately represents the reality.

One way of addressing this problem is to "type" the quantities by the appropriate dimension. This is made possible in OOP by making a separate class for each dimensioned quantity. Not only are errors more easy to trap, the code also becomes more readable.

Obviously, it is desirable to be able to perform arithmetic with the dimensioned classes, but not to have to define the operators for every dimensioned class. One solution is to make use of inheritance and make all the dimensioned quantities descend from a Dimension class, with the operators defined only in this class. The Dimension class holds a real number as part of its data, and the operators work on this internally. Because Length and Time, for example, all descend from Dimension, they all inherit the operators. In fact, all that needs to be defined for each class are functions for creating a new object of the class; called constructors in C++. Because the classes are so simple, it is easy to create a new class when the need for a new quantity arises.

This approach makes a contribution to the validation process by improving code readability and dimension checking. At present, automatic generation of the correct dimension is not possible, all calculations return a Dimension, but this will normally be changed to the correct dimension by putting a constructor of the appropriate class into the code. For example, *length1*length2* gives a Dimension, but *Area(length1*length2)* gives an Area.

CONCLUSIONS

Advances in building simulation are currently being constrained by the difficulty in creating, maintaining and extending programs written in conventional languages.

The EKS provides a new approach to the construction of such programs, based on the increasingly popular (and now proven) object oriented paradigm. In essence it provides a set of 'building block' classes organised in a manner which will support and foster further specialisations to support new developments. The development of this class taxonomy, with its inheritance relationships and internal data structures, has proved a major task, with no single solution. This is why the work to date has focussed on the principles underlying class definition.

Inherent in the classes are validation and documentation mechanisms, and strong type checking in the form of complex objects and dimensioned quantities. These factors combine to make it easier to produce reliable and understandable code. In addition, a mechanism has been developed for the definition and instantiation of objects, using data and class information stored in an object oriented database. Example programs are currently under development and will be included with the first release of the EKS environment.

The EKS should be viewed as one element in the program construction/ maintenance process. In essence, it is the essential complement to other contemporary developments in, for example, user interface design and product modelling.

REFERENCES

Balin O and Sargent R G 1982 "Validation of Multivariate Response Simulation Models using Hotelling's Two Sample T^2 Test" *Simulation* pp185–92.

Bjork B 1990 "Computer–integrated Construction Process: Issues in the development of a building product model standard" *Building Research and Practice* CIB, no. 1.

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 ?" In *Proc. of BEP'91* (Canterbury, UK, Apr. 10–11) pp223–236.

Clarke J A and Mac Randal D F 1991 "An Intelligent Front–End for Computer–Aided Building Design" *Int. J. of A.I. in Eng.* N1 (January), V6 pp36–45.

Gielingh W 1990 "Computer Integrated Construction: A major STEP forward" *Computer Integrated Construction* 1(3), Wix McLelland Ltd.

Irving A D 1987 "Application of Statistical Techniques to the Validation of Multivariable Time Series Simulators" SERC/ BRE Validation Exercise Report, V5.

Kline S J and McClintock F A 1953 "Describing Uncertainty in Single–Sample Experiments" *Mechanical Engineering* pp3–8.

Moffat R J 1982 "Contributions to the Theory of Single Sample Uncertainty" *Translations of ASME*, V 104 pp250–60.

ONTOS 1989 "Object Database Documentation" Ontologic Inc, Three Burlington Woods, Burlington, MA 01803, USA.

Sowell E F 1991 "Next Generation Building Services Engineering Software: Opportunities for the Practitioner" In *Proc. of BEP'91* (Canterbury, UK, Apr. 10–11) pp1–9.

Stroustrup B 1987 *The C++ Programming Language* Addison–Wesley Publishing Company Inc.

Turner J A 1990 "The Conceptual Modelling of a building: Part 1" *Computer Integrated Construction* 1(1), Wix McLelland Ltd.

Wright A J, Clarke J A, Hammond G, Irving A D, James K, Lockley S, Mac Randal D F, Tang D and Wiltshire T J 1990 "The Use of Object–Oriented Programming Techniques in the UK Energy Kernel System for Building Simulation" In *Proc. of 1990 European Simulation Multiconference* (Nuremberg, June 10–13) pp548–554.

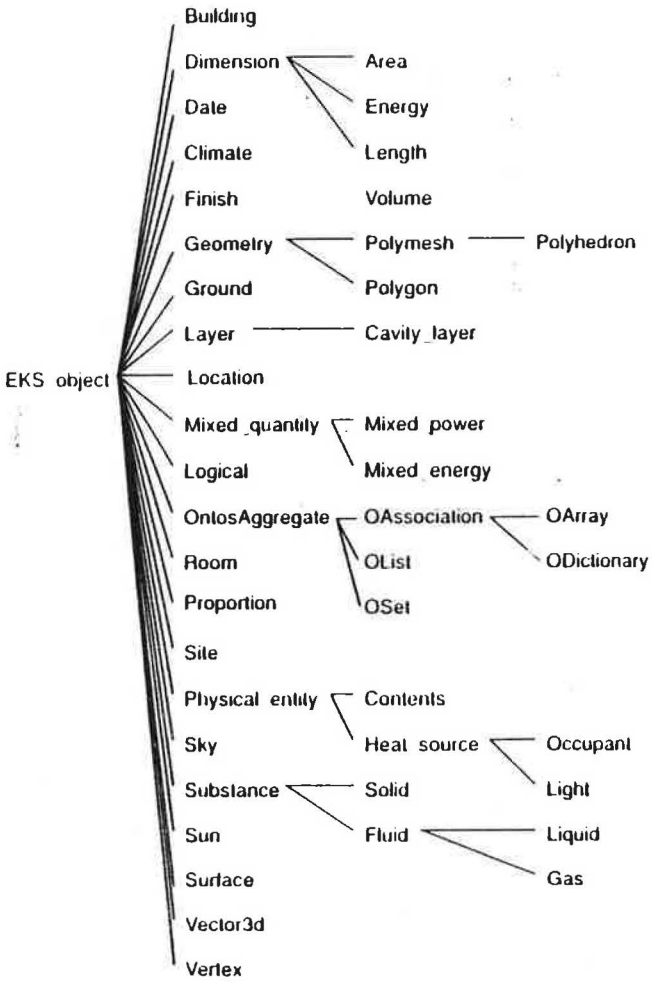

Fig. 3. Inheritance Structure of the EKS.

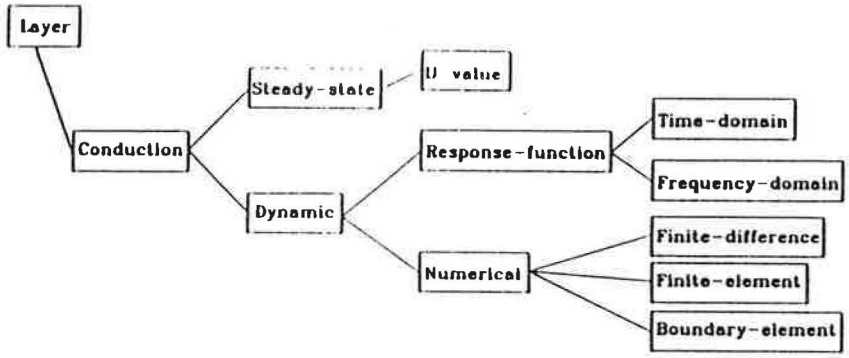

Fig. 4. Derived classes of the Conduction class.

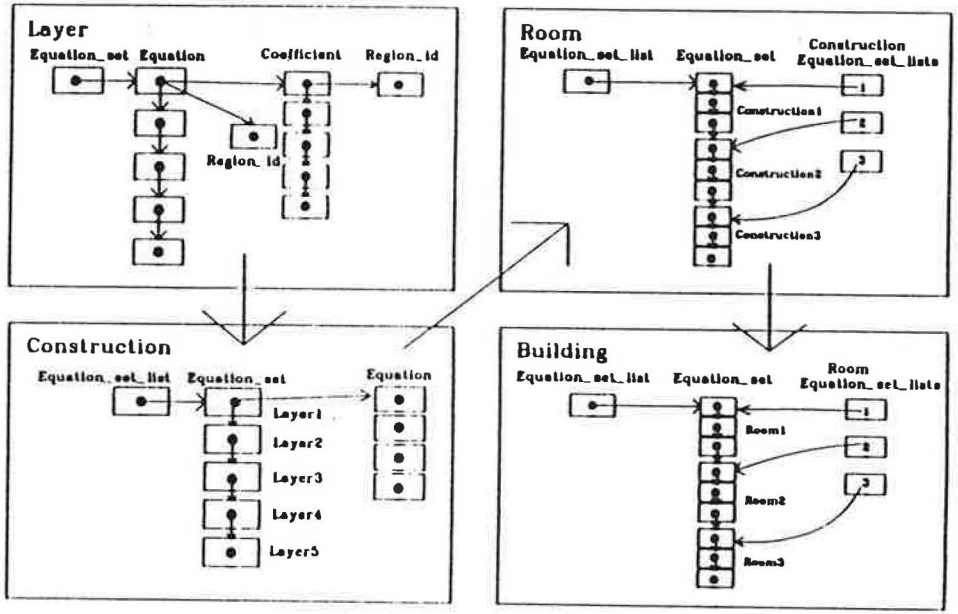

Fig. 5. Classes used to encapsulate theory.