Methods for CASE: a Generic Framework

Mike Brough

Department of Computer Science, Keele University,
Staffs., ST5 5BG

This paper examines some of the method-related issues for CASE. In particular, it discusses the need to move away from the 'pencil and paper' mind-set to a more 'multi-dimensional' approach; a modelling framework, both conceptual and for design; and considers related problems of standards, portability and openness. It is intended that this framework can be used for CASE developers to incorporate generic system modelling techniques, rather than build tools for a single proprietary method.

1 Introduction: why is there a problem?

It might be thought by some that things are getting better for systems engineers in a nice smooth and monotonic fashion. We used to have no methods, then we had methods, better methods, computer-aided software engineering (CASE) tools, better CASE tools ... Is this not a continuing evolutionary improvement in our system engineering method? Unfortunately not. As elsewhere, things do not always get better, sometimes they get worse (and there are often evolutionary dead-ends). The process of evolution is not as gradual as one might think. In biological evolution, the fashionable approach is 'punctuated equilibrium', where long periods of gradual change are broken by sudden changes.

1.1 What are system development methods?

A method is a set of well-defined procedures that can be followed in a systematic way to achieve a goal. A system development method is a framework within which a system developer can work having some confidence that the delivered system will be: the one required; the 'best' way of meeting these requirements; and delivered in the most cost-effective and controlled manner.

This paper discusses methods for identifying system requirements and for designing and constructing systems to meet these requirements (system modelling). It will not be concerned with how to allocate resources, meet deadlines, control and motivate staff (project management). Some system development methods attempt to address both of these concern areas, but they are probably better separated. (Software tools for these two disciplines are usually described as CASE and project management tools, respectively.)

1.2 Why is a method needed?

In order to deal with any complex problem, the human mind requires that we abstract the more important features of the problem and suppress the specific detail. We do this by mapping the problem onto more abstract concepts. These concepts are organised into a framework for our thinking. For example, when writing procedural code, we have concepts such as 'statement', 'iteration', 'block', 'procedure', 'parameter' etc. These are the constructs provided by the framework of structured and modular programming. (Another good

example of a conceptual framework is that provided by the ISO Open Systems Interconnection Model for communications.)

For large systems, we need to think about concepts related to the conceptual (policy) requirements, the run-time units used, interfaces, communications protocols, timing problems, file structures, etc.

Even if a CASE tool allowed very complex systems to be built, it would not be usable without a conceptual framework for using it. This framework is needed to understand what the tool does. Navigation around the models that the CASE tool supports requires a mental view of the 'where we are'. This conceptual framework is provided by a method. Without such a method the tool is unusable.

1.3 What is wrong with existing methods?

Why not use the 'old and tried' methods? Answer: we continue to find new ways of doing things that are: more cost-effective, easier, or more likely to produced a well-engineered system meeting the real requirements. The methods get better.

Additionally, the methods in use at present are those that were invented for pencil and paper. Now that we have finally admitted (!) that it would be a good idea to use computers to help us design and construct systems, surely there are some improvements that we could make. In fact, as yet, most CASE tools have only 'dabbled' with methods. Mostly, they just take the diagrams and supporting specifications that were used manually and allow the designer to store them (this is sometimes referred to as 'upper CASE'). Other tools are concerned with more efficient generation of code etc. (sometimes referred to as 'lower CASE'). CASE manufacturers claim to have 'integrated CASE' (iCASE), but even its name reveals its origin as a miscellany of manual and automated techniques.

Shifting target Methods must therefore evolve. However, this brings its own problems. The whole point about a method is that it should provide a standard framework within which we are confident we know what we are doing. If the framework is everchanging, then we are building on quicksand.

On a personal note, when I worked for Yourdon, I often went round to companies that used the 'Yourdon method' (naturally). We had what we thought was the best way of doing things (!). It was what we taught in seminars and used in consultancy (as documented by [1] and [2]). Most of the project teams that we trained used these methods and were converted to them. However, their management had often encountered 'the Yourdon approach' some years before and liked it (which is why we were working with them). What the managers had encountered was a different version of the method ([3], [4]). As a consequence, the managers did not always appreciate all of what their project team were doing. This confusion led Yourdon Inc. to name the method 'Yourdon Structured Method' (YSM), to distinguish it from the earlier structured analysis/structured design.

Eventually, to gain any degree of stability, methods have to be put under change control and properly documented. Historically, one of the most significant things about Structured Systems Analysis and Design Method (SSADM) was that there was version 2 (and then 3 and then 4 and ...). Yourdon also did the same thing (though rather later), designating the two previous versions of the method as YSM_1 and YSM_2, with the current method as YSM_3 [5]. For both CASE and successful project management, the version of

the method in use must be well-defined. We will discuss these standardisation issues in §4.1.

Pencil and paper 'mindset' The most serious problem with existing methods is that they regard the model as *consisting of* a set of diagrams, possibly with supporting text. Each diagram is two-dimensional, but, of course the system space is multi-dimensional. Different aspects of the system appear on different diagrams, seen from different points of view. To be sure that a consistent, integrated model of the requirements has been formulated, the diagrams and other specifications have to be 'cross-checked'.

CASE vendors talk about this as if it is fundamental to the process of understanding the system requirements. It is not. It reflects a way of thinking tied up with the communication medium. In many cases, what they have provided is an 'electronic pencil and paper' system. Concepts such as 'vertical balancing' and 'horizontal balancing' reflect a way of thinking that is still conditioned by sheets of paper. To get the full benefit from the possibilities of CASE we must modify methods to get full benefit from CASE. Not only will the methods be supported by CASE, but also CASE should drive the methods. Once the break from 'model = diagram set' is made, there are many interesting new possiblities for methods and CASE.

Lack of semantic differentiation One technical problem flaw in some CASE tools is that there is a single 'name space' for all components seen on diagrams. Each item is named and and its specification held in a single data dictionary, identified by the name of the item. This causes: 1. problems in scoping names in large systems, 2. difficulty in distinguishing between items with different semantics. This problem is also a hangover from the restricted technology available in the past.

Ill-defined semantics As methods developed, the semantics of the modelling tools used were not always well-defined. Modellers could perceive the issues of syntax: "Is the box round, or is it a circle ... what do we call this shape on the diagram ... ?". Differing semantics that could be implicit (rarely explicit) in the tool are less obvious.

For example, consider the semantics of an entity relationship diagram. Some methods only allow binary relationships (some even disallow M:N relationships); others allow higher-order relationships. Even then, there may be more subtle differences. For example, both Ward/Mellor and YSM_3 allow higher-order relationships [2], [5]. However, there are differences between these two versions of the ERD. Ward/Mellor define a relationship as a 'linked list' (so that a 1:N relationship is a single unit, consisting of one entity, with a list of occurrences of the other linked to it). On the other hand, YSM_3 regards a relationship to be a set of relationship occurrences; each occurrence of the relationship is the same as any other; one relationship occurrence refers to specific occurrences of each of the entities that participate in the relationship. This difference is not visible in the ERD notations (they are the same), but the underlying semantics are very different. The cardinality of the relationship is a structural constraint in Ward/Mellor, but in YSM_3 it is a specific type of participation constraint. (Participation constraints define rules about which occurrences of relationships are allowed.)

This example is typical of the ambiguity that can arise in interpreting diagrams. Because of these ambiguities, there have been many moves to place diagrammatic modelling

techniques on a more formal basis. It is no longer acceptable to draw ERDs (for example) with sloppy ambiguity during analysis and defend the practice by saying that we will 'firm up' on what we really mean in design.

Of course, when a lower CASE tool is used, the semantics of the modelling tools provided are defined operationally (in terms of what happens). This may not always correspond to the way the system modeller thinks it operates!

The realisation that better defined semantics is needed to 'run' models has caused gradual changes in methods. For example, in the 'Yourdon' methods, there has been a gradual shift from functional decomposition (SA/SD, as described in [3]) through event-partitioning (YSM, as described in [2]) to a complete virtual machine [5] (see §2.4).

2 Moving away from pencil and paper

The main requirement is to move away from the pencil and paper 'mind-set'. This is achieved by a shift in the interpretation of the term 'model'. Rather than interpreting it as 'a set of diagrams, with supporting text', we will interpret it as 'a complete representation of the system'. In other words, it will include all issues of system behaviour and performance that we need to discuss.

The way the model is stored and organised can now break away from the 'page' unit. Of course, we still expect to be able to visualise specific areas of concern using diagrams (and text). These should be regarded as 'views into the model', rather the component parts of the model itself. For example, Figure 1 shows three views into a specific model.

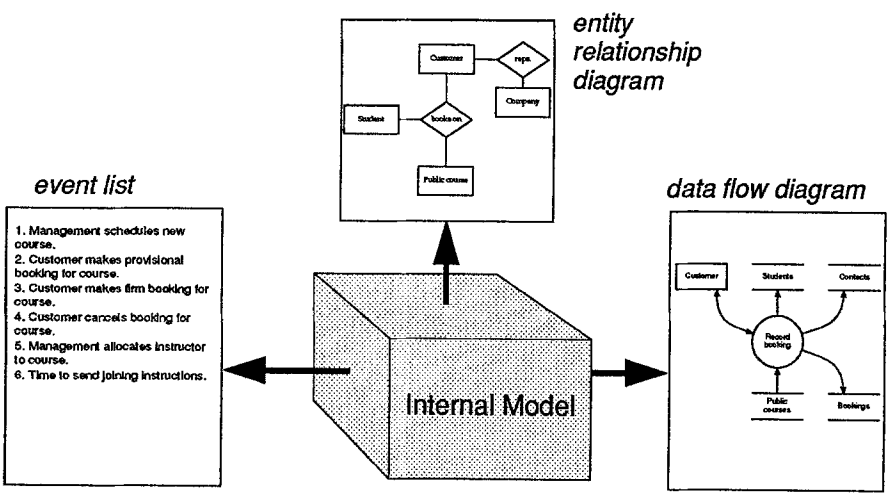

Figure 1: Three views into the internal model.

2.1 The internal model

In Figure 1, the model is labelled 'internal model'. This is because the way it is represented internally is of no interest to the system modeller. Of course, the CASE tool developer will be concerned with the way the internal model is stored. (See also §5.1.) It is a 'black box';

only by means of the views can its contents be defined or examined. The grammars of the internal model and external views are chosen with different aims. For the external views, the choice is to give 'user-friendliness' without confusing the user. For the internal model, the grammar is chosen to give rigour and aid emulation, checking for completeness etc.

2.2 An end to cross-checking

It is meaningless to talk about 'cross-checking' the views. There is only one internal model, so if there is an overlap between two views, two views into it *must* be consistent. In fact, this is a slight simplification because of the need to support way information is gathered.

A CASE tool must allow users to leave views incomplete (e.g. 'dangling' relationships) or inconsistent with other views ("I know I have two views showing the same relationship, but referring to different entities — I'll go away and think about it"). To deal with this, the CASE tool must implement a 'play' or 'pending' area. The possible states of different parts of the model are therefore: agreed (a baseline model), consistent (a working model) and incomplete (pending model fragments).

2.3 Many overlapping views

When a CASE tool supports the internal model and external views approach, the many overlapping views are a plus point (rather than a source of potential inconsistencies). As long as enough overlapping views have been used to capture all required aspects of the model, extra views do not give problems. The way is also open to views being generated automatically from the internal model. This can increase insight into the proposed system.

A view is a 'filter' and a 'translation' at the same time. By filter, we mean it shows some of the internal components only. This filtering may be by type, or subject area, or (more usually) both. An ERD, for example, shows a selection of entities and relationships — it does not show all entities and relationships, nor does it show functions. By a translation we mean it shows then in some user-friendly format. The same components can appear on different types of view. For example, figures 2 and 3 show some model components for a system in a company that runs training courses on a public basis.

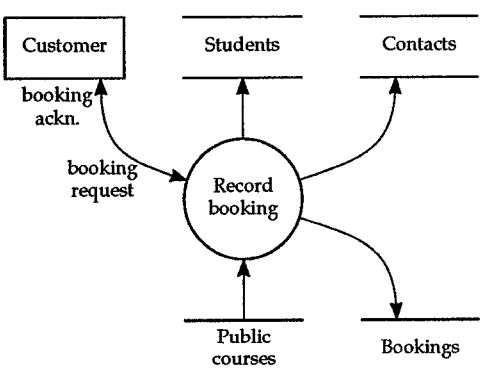

Figure 2: A data flow diagram.

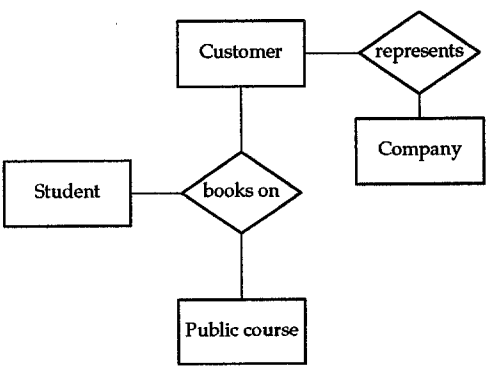

Figure 3: An entity relationship diagram.

Identification of common components How do we know that the store 'Students' is the same as the entity 'Student'; or 'Contacts', the same as 'Customer', '<Customer> represents <Company>' and 'Company'? The 'pencil and paper' approach would have a data dictionary containing something like:

Students = {Student}

and, more awkwardly:

Contacts = {Customer} + {represents} + {Company}

In a CASE tool that supported the identification of stores and ERA components, we might have a 'frame specification' like that shown in figure 4.

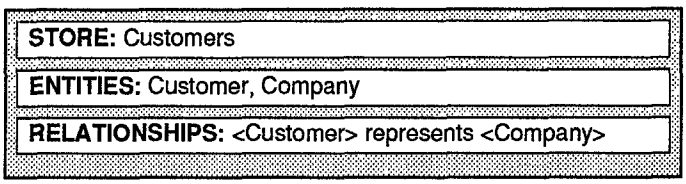

Figure 4: A 'frame' specification.

Of course, this is still really in the 'pencil and paper mind-set' approach. There are many better ways of declaring that a store icon on a DFD corresponds to a given area of an ERD. For example, we could show an ERD and DFD view in two windows on the screen at the same time. Association of ERD fragments with a store could then be achieved by 'pointing' at a store and then the corresponding ERD components. Once this association had been set up, an ERD store could be 'popped open' at any time to reveal an mini-ERD showing the same model components from a different point of view. (The way this association is maintained should be hidden within the CASE tool and not visible to the user.)

2.4 Virtual machine for conceptual model

Most methods distinguish between conceptual and implementation modelling. Conceptual modelling identifies the required policy for the system, without any implementation details; implementation modelling identifies a choice of technology for the system. The specification of the required policy is called the conceptual model (or the essential model, following [1]). The specification (or 'blueprint') of how the system will be constructed is called the implementation model (many other names are in use for this).

What kind of things form the minimal conceptual model? This depends on the method. The main criterion for what goes in the minimal model is that it should be complete enough to run in some virtual machine. The characteristics of this virtual machine are method-dependent. For example, we might use a virtual machine with:

- data transformations carried out by data processes;
- information organised around entities, relationships, attributes and abstract data types (*all* information, not just stored information);
- data manipulation operations that operate on ERA components;

- a set of primitive operations (together with the data manipulation operations, these are used to construct data processes, using a well-defined grammar;
- state machines to synchronise and control.

To avoid distorting the requirements, the performance of this virtual machine should be 'perfect', with: infinite, non-volatile memory; zero instruction time; zero cost, weight, power consumption and size; an infinite mean time between failures; all i-o carried out in zero time using 'extra-sensory perception'; ability to run many processes simultaneously.

There is no technology with these characteristics, of course! It is a convenient mental fiction to visualise the conceptual requirements. However, a CASE tool could emulate it to some degree, as long as the finite processing speed is accepted. The model would not (and could not) run in real time. However, it is an important requirement for CASE tools. 'Running' the models will be an expected feature. To allow this, the semantics of each model component type must be sufficiently well-defined.

2.5 Virtual machines for implementation model

For implementation models, we need to model technical issues such as: processor performance, execution units, data structures and access mechanisms, library routines, etc. The model must allow us to check that they are used to achieve the policy stated in the conceptual model.

It is important to decide what needs to be modelled and make sure these issues are covered by the minimal model. This should be done *before* deciding which views are to be provided. Fixing on a set of modelling tools (particularly diagram types) and then deciding how to cross-check them is not acceptable. That would be a pencil and paper approach.

We can again use the concept of virtual machines. For example, a run-time model uses an architecture that is the same as the one chosen in the actual run-time environment. This model acknowledges the effect of the architecture: the operating system, communications software, packages, DBMS, etc. In other words, we run the model in an virtual machine that has the same architecture as the final system.

More than one type of virtual machine could be used in design. A 'logically perfect' virtual machine can be used to run the model in given processors, with given performance (but suppressing the software architecture). A virtual machine that interpreted source code could be used to 'run' the model at the source code level.

These implementation virtual machines has an important characteristic that the conceptual processor does not have — they correspond to real technology (we shall return to this in §3.3).

2.6 Semantics and syntax

For a given view, we will allow different ways in which it can be presented. For example, a state machine can be modelled using a state transition diagram, or it can be modelled using the tabular equivalent. These are just different presentations of the same information. More formally, we use a modelling tool that prescribes how each model component type is presented. The presentation is derived from the model components shown in the view (acting as a 'filter') using the modelling tool syntax to carry out the translation. This distinction between view and presentation covers such concepts as different graphic icons (trivial), different procedural text grammars and different types of tool (e.g. the STD or

tabular equivalents). Translating a view from one presentation to another using a different tool can be automated (default placement information would need to be generated for graphic modelling tools). Figure 5 shows three different presentations of inheritance, for example.

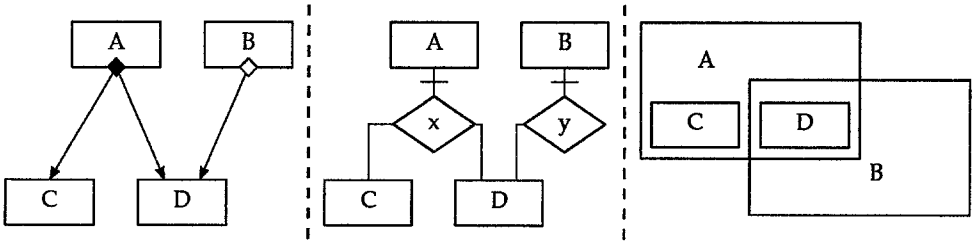

Figure 5: Three ways of showing 'inheritance'.

There is no semantic difference between them, only the graphic modelling tool has been changed for the same view. (Note: although the third diagram is perhaps, the more intuitive, there is nowhere for a CASE tool to provide a navigation path to the model concern over whether the subtyping is complete or partial. The other versions are better in this respect.) This separation of syntax from semantics is now well recognised [6].

Textual modelling tools and dialects The presentation concept extends considerably beyond graphic notation. A function defined internally in terms of its input and output parameters could be referenced in different ways in different presentations. The same program source code (or minispec) could be presented in different presentations of the same view in different ways for different audiences. For example, 'A := B;', 'LET A = B', 'assign (a b)', 'MOVE A TO B', 'Give A the same value as B' are equivalent to the same internal function with different presentation grammars (Modula, BASIC, LISP, COBOL, and 'minispec'). In fact, these are so similar that they might not merit the importance of being called different modelling tools. We might define a single procedural, block-structured language and refer to these as being different dialects of the same tool. A graphic tool (e.g. a Nassi-Schneiderman chart etc.), on the other hand, would be a different modelling tool, and thus give different presentations of the same view.

Different semantics is non-trivial Different methods do not just prescribe different syntax for modelling tools. They often have different semantics too. For example, different methods might use entity-relationship-attribute modelling in analysis. One might allow higher-order relationships, but not the other. An information model built in the more general framework can be translated to one in the less general, but not vice-versa [7]. Extra semantic input is required.

2.7 Generic tools

In the past, methods have used generic modelling tools. For example, DFDs have been used to model conceptual functions, processors, execution units and even source code modules and their interfaces. Each of these is slightly different in terms what the units on the

diagram mean and how they would 'execute'. To resolve this ambiguity, the data flow diagram can be defined as being available in several 'flavours'. Each inherits some general properties (for example processes always represent active units and data flow represent transient interfaces), but there is additional semantic content. For example, on a module diagram (showing source code modules), it is illegal to show 'enables' between two units in the same execution unit (i.e. thread).

One specific area where the generic tool concept is useful is in distinguishing between conceptual and implementation models. Thus a generic DFD can be subtyped into conceptual data flow, processor, run–time unit, and module diagrams. Each shows different layers of technology. This is further discussed in §3.3. (Note: at present, many object-oriented approaches use generic tools and fail to distinguish between conceptual and implementation objects.)

Generic DFDs, STDs, etc. may inherit characteristics from even more generic types of tool, such as graphic tools, frame specifications, production grammars, and tables [5].

3 Framework for methods

We will assume that the method will be based around the idea of models, each consisting of a number of components. Each component is one of a number of pre-determined types; these types are specified by the method (e.g. entity, abstract data type). These components may be modelled using a range of modelling tools. The method will prescribe activities such as model construction and verification, with traceability built in.

3.1 The F^IT triangle

One very important framework concept is the idea of perspective [2], [8], [9]. The system can be modelled from different points of view: a functional, data or some other perspective. These are often overlapping to some extent, but an 'orthogonal' set of axes would be function, information and time[1] — the F^IT triangle.
See figure 6.

These are chosen because a function can be defined, irrespective of when it is used (the function perspective); an event defined irrespective of what is done when it occurs (the time perspective); information modelled irrespective of what it is used for or when. Of course, we do need to model when information is used (the information–time plane); changes of system behaviour caused by events (the time–function plane); the information accessed by functions (the function–information plane).

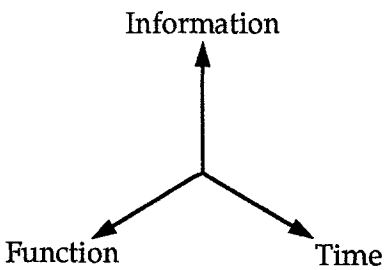

Figure 6: The Function-Information-Time triangle.

1 [2] and [8] chose different perspectives that were not orthogonal.

This framework is not only useful in conceptual modelling; it also persists into implementation. For example: system functions are implemented as 'functional' units (code); events become interrupts, program completion events, etc.; information becomes data structures.[2] In addition, the links between the perspectives are maintained. Conceptual information usage (in the function–information plane) becomes accesses to data structures. Activations (in the time–function plane can be modelled as 'enables', 'triggers' etc. in conceptual models; in an implementation they become the use of activation mechanisms (O-S and procedure calls) in a delivered system. The F^{I}T triangle acts as a useful organisation concept in checking allocations have been carried out (traceability) [9].

As described in [10], the existing system modelling tools can also be related to the F^{I}T triangle. See figure 7.

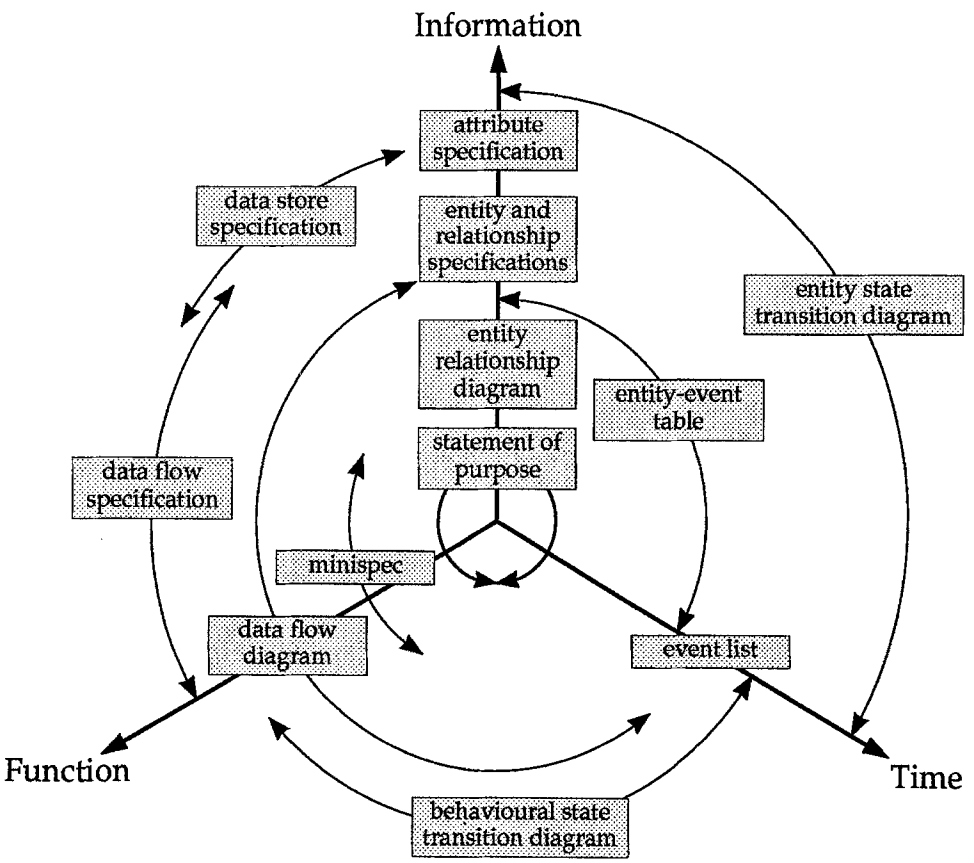

Figure 7: Modelling tools and the F^{I}T triangle.

2 Not absolutely true, because functions can be represented as rules (data) with interpreters, but it is more common for units not to cross perspective boundaries when they are mapped onto the technology.

3.2 Conceptual and implementation models

Conceptual modelling — complete or not? The conceptual model can be partial or complete. Some methods do not cover all parts of the F^{I}T triangle. They may also suggest that the conceptual model is deliberately high-level and incomplete. However, the author thinks that this is a dangerous stance and prefers the model to be complete. In any case, emulation requires that all policy is covered by the model. This wish to run models is one of the main reasons for moving away from the 'set of diagrams' approach.

One problem that needs to be addressed is what to do about 'fuzzy' decisions. Most system modelling methods require deterministic logic. YSM3, for example, requires that all processes are completely specified by state machines or minispecs [5]. The minispecs must unambiguously prescribe how the inputs are transformed to the outputs each time the process is activated. Minispecs are written using a well-defined grammar, that could in principle, allow them to be 'run'. In some systems, however, there are decisions that would be impossible to model in this way (this is often true when the intended processors include people). In these cases, a CASE tool could not emulate the policy, except by 'breaking out' of the model to ask the system modeller what the actions should be, each time the function is activated.

3.3 The relationship between system implementation models and the architecture

The implementation model shows allocation to technology. This includes allocating system policy to implementation units such as processors, execution units, files, data structures, modules, etc. As mentioned in §3.1, this generally maintains the position in the F^{I}T triangle.

Architecture structures and units A design cannot just be generic — it is always a design for a specific architecture. The run-time model must be organised to take account of the architectures of each intended processor type. Source code models must take account of the architecture of the programming language.

This is an important type of constraint that has been long realised. Experienced designers implicitly design for a specific architecture. However, methods have not usually dealt with this very satisfactorily[3]. There are two concepts that are useful in this area: the distinction between conceptual, technology and allocation views; strong typing of the implementation model.

Conceptual views Figure 2 is a conceptual view, showing a system function "Reserve place". All components on this diagram must correspond to an instance (or aggregate of instances) of one of the model components that can be seen in this type of view. For example:

- the process is view of a function (which also has a required internal view — a minispec);

---

3 Note: some CASE tools are specifically intended for design for a single architecture. They are not subject to this criticism. However, they are of less general application and we therefore ignore them.

- the data store is a view of a collection of stored data items, which are organised as occurrences of entities and/or relationships, together with their stored attributes;
- data accesses (between process and store) are a view of the access to stored information by the function (this can also be 'filtered' out of the corresponding minispec);
- the terminator is something outside the model scope that the model interacts with;
- data flows are a view of temporary data items (at the elemental level, each component is an attribute — possibly temporary, and not retained);

The fact that each component on the diagram represents an instance of a specific minimal component types is referred to as the 'strong typing of the conceptual model'. It provides semantic security, rather than just a 'pretty picture'. For the typing to be complete, we require (for example):

- All data accesses must use an allowed data access operation provided for a conceptual data model (probably an ERA model);
- All data items must be typed as an abstract data type (in rare cases, we might allow 'anonymous' ADTs, where comparisons do not need to be carried out).

Technology views In a run-time model, a particular execution unit might need to access several data structures. (By this stage in the design process both of these issues will have been addressed.) Figure 8 shows such an execution unit.

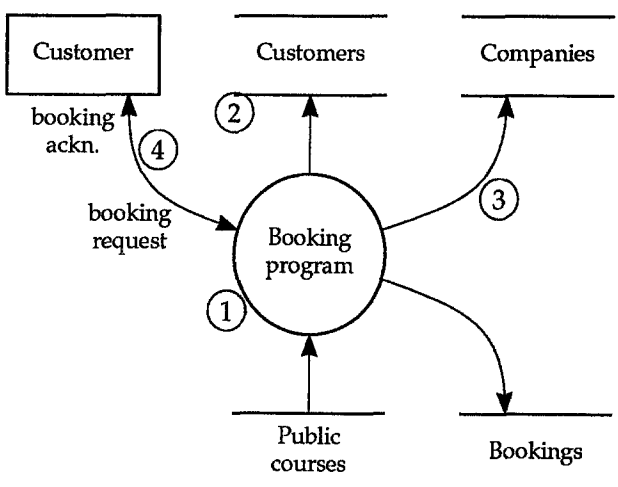

Figure 8: Strongly typed implementation view.

This is again a view into the minimal model (this time the implementation model), but the filter shows technology units. Each of these must correspond to technology provided by the run-time architecture. For example:

- The 'Booking program' must be identified with a type of execution unit allowed in this architecture.

- The 'Customers' store must correspond to an allowed run-time data structure (for example, an SQL table).
- Access to a data structure must be via the data manipulation operations provided by the architecture (e.g. 'select', 'project', etc. for a relational DBMS).
- User input-output must be associated with available technology, both hardware and software (including screen 'look and feel' standards).

This diagram is therefore a technology diagram, showing technology units. It is strongly typed by requiring that all units shown must be an instance of a type of unit provided by the run-time architecture. The F[I]T concept is useful here — for example, functions are usually implemented by 'processing type' units, as previously discussed.

Allocation views We can also show the conceptual 'fragments' allocated to a specific implementation unit. This is an allocation view. Allocation views can be used both for: functions allocated to an implementation unit (i.e. DFDs); information allocated to data structures (i.e. ERDs); functions, information and dynamics allocated to an implementation unit (i.e. object diagrams). In the example, we could show the ERD fragments allocated to the data structure Bookings (the entity Student and the relationship <Customer> reserves place for <Student> on <Public course>).

Resources and strong typing The link between systems models and reusable resources (which should be regarded as enterprise resources) is a particularly important one. Very few systems are built in isolation. The relationship between system models and resources such as the O-S and DBMS are particularly important. Only by checking that these interfaces are correct can the design be verified. One convenient way of thinking of an implementation model is shown in figure 9. This shows a system implementation model (the Code model, see [2]) as a conventionally levelled set of generic DFDs (of different subtypes). In addition, we see that each component of these technology diagrams is associated or 'tied' to a resource provided by the architecture. In this diagram, four resources are shown. Each provides a set of component types and services to a 'higher' layer. Each of these resources can be modelled using the same strategy. For example, a model of a communications service would interact with a client application; it would also use lower-layer resources such as 'hardware'.

Note: it is particularly attractive to model each of these resources as providing a set of objects that can be used by other (client) resources, including the application layer.

Resource libraries All available resources, whether execution unit types, functions, subroutines, abstract data types or communications protocols (for example) are enterprise resources. A CASE tool should hold a representation of the facilities they provide in a central repository. This repository is separate from the system models. This allows various desirable features to be provided by a CASE tool. One of these is the ability to 'browse' through a set of available resources when carrying out design. Conceptual units can be allocated to these resources on an interactive basis and this association retained by the CASE tool (internally) [11]. Technology and allocation views could then be automatically provided.

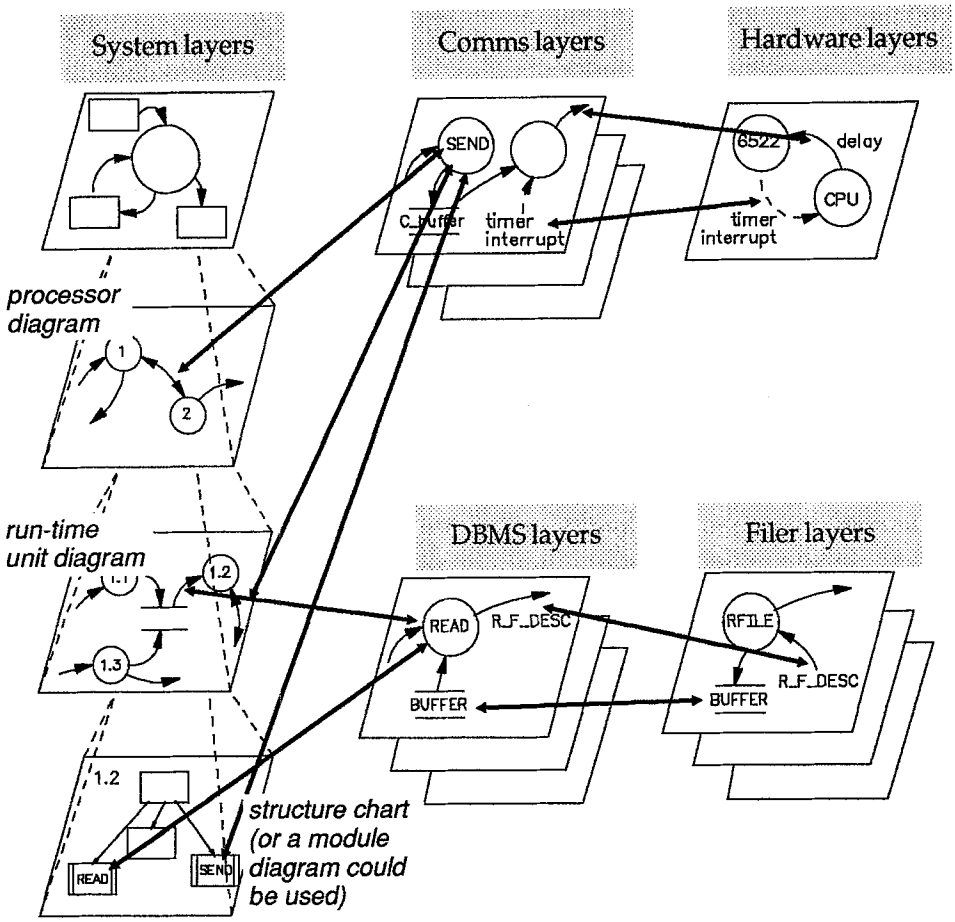

Figure 9: The links between technology layers.

Another benefit would be that individual resource types can be 'parameterised'. For example, an execution unit type might have a certain storage and activation overhead. We might allocate a certain function to a process of this type to provide part of a system's functionality. The function would have a complexity, which would require storage and processing resources for each instantiation of that function. In a sophisticated CASE environment, putative allocations could be carried out and a calculation of whether system performance constraints are met.

Parameterisation of this type would be a very large undertaking if all architectures were tackled, but a gradual 'capturing' of this data is feasible, once a framework is agreed. It is likely that the F^IT framework will be useful in this parameterisation. Additionally, the parameterisation needs to be at both the type and instantiation level. Sometimes there may be multiple instantiation level overheads. We might have a component for implementing an object on an execution unit and processor level, as well as the instance level.

In the P+D paradigm, data and functions are allocated in parallel. In the object paradigm, they are encapsulated (possibly even in analysis) and then allocated together.

3.4 Process+Data versus Object paradigms

The above framework is very general. The concept of technology layers of technology is still valid, whether the system is thought of as processes + data (information, data structures → execution units + data structures) or objects (conceptual objects → implementation objects).

Of course, the organisation of the models would be different. In the P+D approach, functions are allocated to processors, then execution units, then source code units. This progressively pushes down a 'technology line' towards the final organisation, as shown on the left of figure 9. On the other hand, if a conceptual object is distributed across processors, then the processors will appear 'within' the objects (in fact, hidden in transducer objects). 'Inter-task' transducer logic appears within an object when it is distributed to different run-time units (the 'inter-process' transducer logic disappears within these). None the less, we can still think about technology views (showing implementation objects and structures) and allocation diagrams (showing conceptual units allocated to them).

In either approach, strong typing of the implementation model requires that all services used in one resource must be typed to be an instance of a service type exported by another resource.

3.5 System scope

One of the more important characteristics of mature methods is the way in which they integrate multiple systems. Most methods now recognise the existence of both enterprise and system models [5]. The enterprise model is the summation of all system models within some larger scope (often the whole business enterprise). (Note: the enterprise model is different from the resource library. The enterprise model is a 'super-model', used to check access to shared items, such as information; the library is a collection of re-usable components.)

This scope distinction can be generalised so that all areas of activity can be considered to be 'chunks', consisting of information, activity and dynamics (they could also be described as 'conceptual objects'). In the middle range, we have system-size chunks; larger chunks correspond to enterprises; smaller chunks to objects.

The interfaces between the chunks are: shared information, messages, functions within one chunk used by another chunk, and events (actions in one chunk that cause responses in other chunks). Even interfaces to systems that are external to the enterprise can be described in the same way. This rationalisation allows another simplification in the traditional system-oriented, process+data approach. What was regarded as a terminator in a DFD can now be seen to be a chunk. If it is truly a black box, it can be thought of as a source or sink (terminator); if more insight is available, it can be modelled as an object, with visible external interface functions (or methods).

4 Method definition

4.1 Who owns methods?

The ownership of methods is an important issue. There are effectively four possibilities:

1 international standards body or government 'quango';

2 individual person, who develops method and advocates it via books and papers;
3 commercial company that specialises in consultancy, training and/or CASE;
4 open market-place standards set by common practice;

Each has some advantages and disadvantages:

Sponsor	Advantages	Disadvantages
Standard	Available, permanent, good support	Inertia, not best current practice
Individual	Often well-thought out and intuitive	Little support or permanence
Commercial	Good support	Expensive, inflexible. Often driven by commercial factors, leading to inertia. Best practitioners find this environment unattractive to work in.
Market-place	Available, dynamic	Ill-defined, dynamic. May lead to choosing a tool and method pair that is a good tool, supporting out of date method.

Probably the best compromise would be the development of 'method forums'. Individuals and companies who are interested in method enhancement would contribute to technical debate and development. They might also sponsor the development of software tools and training seminars. These groups might be organised on a regional or industry-type basis. This will give the flexibility of fairly rapid incorporation of new concepts, with the advantages of wide review and sufficient financial leverage for development of delivery vehicles and tools. Syndicates of this type have been used in some areas of software engineering (for example, expert systems), but it is not clear whether an similar initiative for method development would be successful.

4.2 Requirements for methods

What kind of thing should we be looking for in a method? Rather than getting trapped in the specific details, there are certain characteristics that we would expect to see in such a method.

Flexible It should be possible to use the same method in different ways for different ends. A method which is only applicable to a very small range of problems will not gain wide acceptance. There will be few tools available to support it and training will not be cost-effective. There will be problems with obtaining people who understand the method.

The method should also cater for individual tastes in approaches. A method that prescribed a single way of doing things is:

- boring: and hence not conducive to quality work;
- not cost-effective: different problems may allow different 'short cuts' to be taken to achieve the desired goal; always requiring the same approach ignores this;

- not realistic: a single 'cookbook' approach will not always work, in spite of its advocates trying to sell 'new clothes';

Two extreme approaches to system development might be to:

- Completely specify the system using 'structured techniques' before building any of it;
- Build a little, test a little, ... (the prototyping approach).

Neither of these approaches should be followed in too blinkered a way. Specifying without some prototyping is difficult and laid about by many pitfalls; prototyping without some modelling discipline becomes 'hacking', leading to very poor systems. We would therefore reject the advice of anyone who said "You only need to prototype", or "Prototyping should not be used".

The method may have features that are not applicable to all kinds of system. For example, it is often stated that state machines are not needed for business systems. The justification would be that adequate models can be built without them[4]. We would certainly allow this sort of thing (though not, perhaps, this specific example). We require that the same framework is used for a wide range of systems, even if all the facilities within the framework are not used on a specific project.

Adjustable rigour Not all users of a method will want to use the same level of rigour. Greater rigour may give increased confidence in quality of delivered systems, but at a cost. Not all projects justify this. The freedom to be more or less formal should be fundamental to the method. Similarly, building models with different degrees of completeness should also be allowed . In both cases, this should be a continuous spectrum.

Regarding a minispec in structured text as an informal modelling tool and the equivalent, written in Z is *not* the way to take advantages that software support can provide. In this specific case, the internal model/view/presentation approach works well. The internal model is a complete specification of the function. It can then be presented in different views using modelling grammars aimed at different audiences. The 'grammar' of the internal model is rigorous; the external views are derived.

Extensibility Methods should not be monolithic. As new modelling tools, models, heuristics etc. are developed, it should be possible to integrate them into the existing method to extend it. The approach proposed here should allow this extensibility, within the F^IT triangle. For example a new method heuristic (function) might be defined; a new model integrity rule (information) have their place in this framework.

Modelling concepts A method should provide conceptual building blocks of the same type (but not necessarily identical to) the ones presented in this paper: models, modelling tools, views, presentations, heuristics, rules, guidelines.

---

4 In fact, there are few systems that always operate to the same policy at all times. State logic is therefore probably required.

Scoping Any method must be allow modelling at several scope levels. Preferably, these levels would not be hard-coded into the method, but as a minimum, object, system and enterprise scopes must be recognised. Re-use of units (functions; objects; abstract data types; and implementation units such as packages, subroutines etc.) must be allowed for by providing a resource library modelling concept.

Models There should be a range of models, each addressing specific issues. Not all models need to be used in any one project, but the method should allow them if required. Each model should be defined as consisting of components that run in a virtual architecture of some type. If such a virtual machine cannot be defined, then the model is incomplete or ill-defined.

Modelling tools The method should provide a range of tools to provide views into the models. Each of these should have well-defined semantics in terms of which internal components are visible (filtering) and how they are shown in the view (translation). It should be possible to relate the modelling tools to some abstract framework, such as the F^IT triangle. For example, a behavioural state transition diagram 'lies in' the time–function plane. See figure 7.

Notations For each modelling tool, there should be one (or more) well-documented notation. This is a syntax, rather than a semantic issue. For a DFD, we could allow different icons; for a procedural minispec, we might allow different presentations (e.g. in COBOL, Fortran, Z, structured text). (See §2.6.)

Heuristics, rules and guidelines There should be a set of rules and guidelines for completing and reviewing the models. Guidelines should also cover the definition of suitable views ('coherence' concept), layout of diagrams, naming of components, etc.

There should be techniques for construction of the models. These should range from heuristics, through open rules to closed rules. (A closed rule is one that is sufficiently well-defined to carry out without any human input, for example, as implemented by a compiler; an open rule is one that requires some ad-hoc human input [9].)

Management framework Although this paper suggests that system modelling should be distinguished from project management, the correct 'hooks' in system development methods are required. For example, the method must provide concepts such as deliverables, models, views, review sessions. Resource and construction cost requirements can also be related to the internal components (although probably only described as seen in views). As a management framework, building any system can be regarded as a construction problem in the F^IT domain (a function).

5 The way forward

How shall we improve our systems engineering techniques? We assume that CASE and methods will develop hand in hand to increase our ability to construct quality systems for complex requirements.

One important issue is avoiding 'lock-in'. Being forced to use a single tool or method indefinitely, even when it is no longer the best, is clearly not a good idea, even apart from the commercial advantages of 'shopping around'. Monolithic CASE tools (typical of most currently available) have this disadvantage.

5.1 IRDS facilities

One of the important initiative in this area is the development of Information Resource Dictionary System (IRDS) frameworks, and hopefully, products. These separate the storage of models from the CASE tools, allowing some freedom to change tools. In the longer term, these will also incorporate the ability to accommodate different methods in the same IRDS and possibly event 'tailor' the methods.

There are several standards bodies working on this (ANSI, CDIF, ISO, and PCTE are the main groups), but there is a gradual convergence towards the initiative carried out under the auspices of the ISO standards committee [12], [13].

5.2 Framework for methods

To allow an IRDS to have the freedom to deal with different semantics and methods, several layers of abstraction are required. Figure 10 shows some of the concepts mentioned in this paper, organised according to the ISO framework.

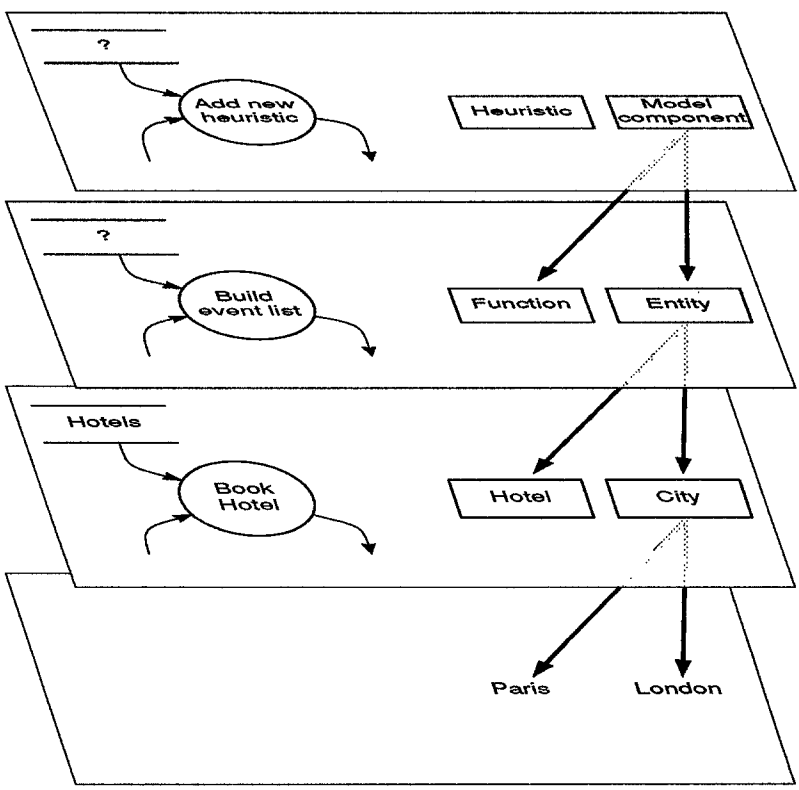

Figure 10: Layers of abstraction in system modelling

There are four layers of abstraction.

1 Application level. This level corresponds to the delivered system. Instance data corresponds to the actual real-world things that the system deals with.
2 Model level. This is where most people tend to think of methods and CASE. Instance data corresponds to information about which entities, processes, state machines etc. are required for the delivered system.
3 Method level. At this level the instance data corresponds to the modelling components used. For example, most methods use components such as entities, relationships, ...
4 Method definition level. At this level the instance data corresponds to concepts used in the method. For example, we might have models, views, presentations, modelling tools, heuristics, etc.

The semantics of any building block is always defined at a higher level. Discussion on whether relationships in YSM_3 ([5]) are the same as in YSM_2 ([2]) — they aren't — are discussions at the method definition level.

We can have functional type things too, shown as processes in figure 10. At the application. level, there are functions as seen on a typical system DFD. At the model level, they are activities like build model, cancel project, check event list. At an even higher level they are activities such as add new model, identify implementation for essential system component, add heuristic to method. We can go even higher, but this is probably too philosophically vague. We do not want activities like define new system modelling framework. This is particularly important if tools and IRDS facilities are to allow upwards-extension.

We can also look at the time domain. System events, modelling events such as essential model completed, customer agrees change to interface, management sets new project goals. At the method level, new version of method released, CASE module built, etc. are valid time points.

Note that in the case of information there is a 'pair' structure. An entity at one level has instances in the next level down, which is why the system is at the lowest level. Each function creates (deletes, modifies etc.) instances at the level below.

5.3 Contents modules

It would be ill-advised to try to define all of the IRDS framework in 'one go'. The ISO framework therefore provides for contents modules for standard system modelling techniques. We could thus have a contents module for a modelling tool, structure of a model, heuristic, a grammar, what constitutes a view, complete method, etc. It is important that these modules are able to reference each other and this has been provided for. At present, several contents modules are actively under development. After they have been delivered, it is hoped that guidelines for production of such modules will be formulated.

Generally speaking, contents modules cannot be defined in isolation. They will need to refer to existing contents modules. Less obvious perhaps, is the requirement that they tend to depend on higher-layers (in the ISO sense [12]). For example, we might attempt to define a content module for ERA modelling components (there would need to be several variants of this, of course). In this, an entity (for example) could either be regarded as:

- a component of an ERD, which is an instance of diagram type; each model consists of multiple instances of each diagram; each component has a corresponding specification ...;
- an instance of model component; it can be seen in various views; each view can be represented with different modelling tools ...

In other words, meta-modelling (discussion of what an entity is) requires meta^2-modelling to define model component, view, etc.

5.4 Implementation of this strategy

There are certain considerations of a political nature. Not all methods are openly available and documented (although one would be ill-advised to use a closed, proprietary method anyway). The support, adoption and promulgation of suitable standards will also be important. One trap that is not always recognised is trying to impose methods that do not work. Often, companies claim they use methods, when, in reality, the methods are talked about by the managers and politicians, but not actually used in the way they are described (the Emperor's new clothes problem).

Standards should cover: the modelling framework, lifecycle model, IRDS, modelling tools. Standards should be extensible, in the sense that there should be freedom to customise by: changing names, notation or syntax; adding or deleting method components and so on.

Education Books, consulting and education should promote the concepts, not notation or proprietary methods and tools. There should be a generic approach, with evolution within a framework to provide stability. This will require end-users to be aware of the issues and avoid getting 'locked into' specific commercial approaches.

Psychology There are also psychological factors that need to be considered. One is that people hide behind (and defend) a named method or CASE product, rather than the concepts behind it. For example, some people would get into very heated arguments about how important DFDs are in the system life cycle. Truth to tell, they are good at showing interfaces between functions, no more and no less. Is this all we are concerned about? No. Then we need other tools. They are not more or less important, they are other tools.

People can also get very aggressive about notation for no good reason. What research has been done indicates that notation should be sparse (but well-defined), rather than rich. Yet many still want to invent (and enrich) notations for all sorts of things.

The framework of the method used by a CASE tools should always be intuitive [14]. Arcane and esoteric ways of doing things should be seen for what they are — inefficient and outmoded. Most people accept the revolution of the GUI. On the other hand, we all cling to the word-processor that we first learnt. I have been through three word-processors now (and numerous text editors) and I still get into heated (sometimes aggressive) debates about which is the best!

Technology The CASE technology should support the generic approach that we have proposed in this paper. As IRDS concepts will never quite work as we mean them to (open

availability to several CASE tools), it is important that standard import/export interfaces are defined too.

The semantic definitions should not be hard-coded into the CASE tool, but modelled as information and rules in the IRDS. This is some way off at present, and we may have to do with a range of generic techniques provided by the CASE tool.

The technology should win on adherence to standards, price, performance, user-friendliness etc. and not on 'our method is best'. It is in the customers interest to apply pressure to ensure this happens.

6 Conclusion

The above ideas are presented as an evolution of the ideas of other system engineers, particularly those who have used 'Yourdon-like' approaches. Few ideas are revolutionary, yet hopefully they will provide a framework for thinking about system modelling techniques using CASE.

Minimal models, views and presentations should provide a common philosophy for interacting with CASE products to produce models; association of implementation units with architecture features (strong typing), will enhance productivity and confidence in the design stages.

7 References

1. S. McMenamin and J. Palmer, Essential Systems Analysis. Yourdon Press, 1984
2. P. Ward and S. Mellor, Structured Development for Real-Time Systems. Yourdon Press, 1985
3. T. DeMarco, Structured Analysis and System Specification. Yourdon Press, 1978
4. E. Yourdon and L. Constantine, Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. Yourdon Press, 1978
5. J. Baker, M. Brough and N. Matzke, Yourdon Reference Manual. (obtainable from CGI Systems), 1990
6. D. Redmond-Pyle, Can formal methods be user-friendly? American Programmer, May 1991
7. M. Brough, Standards for CASE. European CASE II, Blenheim-Online, 1990
8. T. DeMarco, Controlling Software Projects. Yourdon Press, 1982
9. M. Brough, A Framework for Relating System Development Methods. Structured Development Forum X, San Francisco, 1988
10. M. Brough, Methods First, then Tools. DECUS (Methods, Languages and Tools special interest group), 1990
11. D. Budgen and G. Friel, Augmenting the Design Process: Transformations from Abstract Design Representations, CAiSE-92, Springer-Verlag, 1992
12. ISO/IEC 10027, Information Resource Dictionary Systems - Framework, 1990
13. Information Resource Dictionary System (IRDS) Services Interface, ISO/IEC draft standard 10728, 1991
14. R. Holmes, private communication, 1990