KNOWLEDGE ANALYSIS OF TASKS AS A BASIS FOR INTERFACE DESIGN OF COMPLEX DEVELOPMENTAL SYSTEMS

Alan W. Colman and Ying K. Leung,
Swinburne Computer-Human Interaction Laboratory,
School of Computer Science & Software Engineering,
Swinburne University of Technology, Hawthorn Campus,
Victoria 3122.
e-mail: {awc,ying}@jupiter.cs.swin.oz.au

ABSTRACT

Complex development systems are interactive software systems used for the manipulation, design or development in complex problem domains. This paper highlights some of the limitations of Johnson's Knowledge Analysis of Tasks (KAT) (Johnson, 1989, 1992) and proposes a modified version of KAT where task knowledge can be analysed and grouped in a way that will make it useful in the derivation of interfaces in complex developmental systems. This modified form has been applied to the domain of intelligent distributed control systems in an attempt to develop interface concepts for the development, installation and documentation of such systems. The paper further shows how this extended version of KAT may prove a useful input to object-oriented analysis.

INTRODUCTION

Developing methods of task analysis has been a major endeavour in the field of Human-Computer Interaction (HCI). The aim of task analysis is to ground the analysis and design of systems in the study of users interaction with such systems. Diaper (1989) gives an overview of the various and diverse methods of task analysis as they have developed from the study of human factors and ergonomics. Some of these methods include Hierarchical Task Analysis (HTA), Task-Action Grammar (TAG) (Payne and Green 1989); and Task Analysis for Knowledge Descriptions (TAKD) and Knowledge Analysis of Tasks (KAT) both developed by Johnson (1989). All the methods developed recognise the importance of the cognitive aspects of tasks in studying HCI, although there are differences in approach as to what extent task analysis has to be based on the systematic *observation* of people performing tasks (Diaper 1989 p11). The transition from task analysis to system design has also been problematic. This paper is an attempt to apply Johnson's task analysis method KAT to what we have characterised as "complex developmental systems", in a way that will prove useful for the design of software interfaces.

Complex developmental systems are interactive software systems used for the manipulation, design or development of complex problem domains. Examples of such systems are CASE

tools, simulation software, control network development systems, etc. These systems are complex in the sense that the problem domain is structurally complex and multi-faceted. Information on the state of facets of the problem domain needs to be presented to the user, in order for the user to manipulate the problem domain or its representation effectively. Structurally complex domains can be represented semantically as sets of relationships between different types of objects. For example, plain text in this sense is not structurally complex (although it can be structurally large); so whilst a word processor is a complex piece of interactive software, it is not representing a structurally complex domain[1]. Complex developmental systems are developmental in the sense that they are used in the process of design, and thus the representation of the structure of the problem domain changes over time. A real time monitoring system may be interactive and structurally complex, but is not developmental.

Complex developmental systems exhibit a number of features that impact on the way we analyse tasks performed on such systems in order to provide an input to interface design. These can be summarised as follows:

- the development tool can be separated from the problem domain;
- the development process is opportunistic;
- the problem domain is structurally complex and the user must maintain a sophisticated mental model of the problem domain in order to interact with it. To represent this domain the system must present multiple system-images.

Knowledge Analysis of Task (KAT) is a method for identifying a user's task knowledge structures proposed by Johnson (1989). Johnson identifies three distinct parts to the KAT method:

(1) identifying knowledge - the identification and collection of knowledge that people use in the performing of tasks.
(2) analysis of knowledge - the identification of the representativeness and centrality of a particular task knowledge component and establishing generic task knowledge.
(3) construction of a Task Knowledge Structure (TKS) - a TKS is knowledge of tasks "acquired through learning and performing a given and associated task" (Johnson 1989, p167).

Johnson categorises these knowledge structures into three substructures:

Goal substructure - knowledge about goals and subgoals. This substructure includes enabling and conditional states that must prevail if a goal or subgoal is to be achieved.

Taxonomic substructure - knowledge about the properties of task knowledge and their associated actions. The taxonomic substructure "identifies the object properties and attributes, including class membership, the procedures in which it is commonly used, [and] its relation to other objects and actions..."

Procedural substructure - knowledge about achieving a goal. Procedural substructures are developed for "well-practised" tasks and are "different from goal substructures in so much as they are executable in a single unit".

---

[1] Text, of course, has a grammatical structure, but a writer is primarily concerned with the meaning of the words, not the structure of the grammar. A word processor is unstructured in that it cannot differentiate the semantic structure of the text. Text can have semantic structure in terms of section headings, chapters etc and some modern word processors such as outliners and folding editors understand such structures.

Johnson is concerned with the learnability of user interfaces. The theory of TKS assumes:

> "as people learn and perform tasks, they acquire (or more correctly develop) knowledge structures that are established from previous task experiences and applied to future task performance in a dynamic process.... Usability and learnability are directly related to the amount of existing knowledge that a user is able to transfer from the existing tasks to the revised form of those tasks as they should optimally be performed using the designed software" (Johnson 1989 p.162).

The analysis of task knowledge involves identifying the "representativeness" and "centrality" of objects and actions, in order that the task knowledge structures can be made more general, and thus transferable (learnable).

This paper suggests some extensions to the KAT method which can be applied to complex developmental systems. These modifications arose from an attempt to apply KAT to analysis of a development system for intelligent distributed control networks, the aim being to use KAT as input to the design of an interface for such a development system.

COMPLEX DEVELOPMENTAL SYSTEMS

The Separation of Development Tool from Problem Domain

In a complex developmental system we need to distinguish knowledge of two domains: (1) the **tools** for manipulating and developing the problem domain, and (2) the **problem domain** itself[2]. In computer systems, tools can vary from very simple to complex - with the complexity the task differing accordingly; for example, from the bold button on a word-processing tool-bar to a network traffic analyser for a distributed control system. In any description of tasks, the description of the action-object pairs will differ depending on whether we are describing interface objects as tools (e.g. "with a mouse, move the cursor to the button and click the left button"), or whether the objects we are describing are problem domain objects (e.g "define the network variables for the alarm node").

An analogy with an architect's task is apposite. If we want to describe the knowledge required to design a house, we need to be concerned with knowledge of design principles, building regulations and the current state of the design, rather than just a knowledge of drafting skills. The knowledge of the tools is secondary (knowledge of how to *draft* a house design may change depending on whether you are using a pencil or computer) to the knowledge of the problem domain (knowledge of how to design a house and knowledge of the state of the design).

Traditionally, much human-computer interaction (HCI) research into tasks has focused on describing task *execution* at the tool level - e.g. Keystroke Level Model (Card et al. 1979). This is, in part, due to the fact that the tool is intrinsic to the interface; the interface is seen as separate from the "real" underlying application. It is also easier to quantify performance of task execution at the tool level, rather than to quantify task acquisition, or execution, at the

---

[2] This distinction between tool and problem domain is similar to the distinction between the lexical/syntactic and semantic layers of an interface in the 'Seeheim Model' see Johnson, Drake ,Wilson (1990). Carroll and Olson (1988) make a similar distinction between "Task-knowing" and "System interface knowing" in mental models, but adds knowledge of how the system works - "System-architecture knowing"

problem-domain level. Understanding task acquisition or execution at the problem-domain level relies on concepts such the user's "mental model" which is difficult to substantiate.[3]

To interact with a system the user requires problem-domain knowledge of two types:

- task acquisition requires the user to have an understanding of the current state of the particular problem-domain development. The system interface needs to provide feedback to the user on this current state. To develop an interface we need to develop system-images to represent the problem domain. In complex domains these images are more easily understood if the problem domain is amenable to representation in graphical form (Larkin & Simon, 1987).
- task execution requires problem-domain knowledge on *how to* manipulate the problem domain. Such knowledge is often held by the user rather than by the system. Systems that incorporate such knowledge include rule-based expert systems. The incorporation of such knowledge into the system is outside our definition of a complex developmental system.[4]

The interface for a developmental system must, therefore, represent both the *problem domain* over time, and provide a *tool* for manipulating the problem domain. Figure 1 illustrates the inherent interaction between the user and the problem domain.

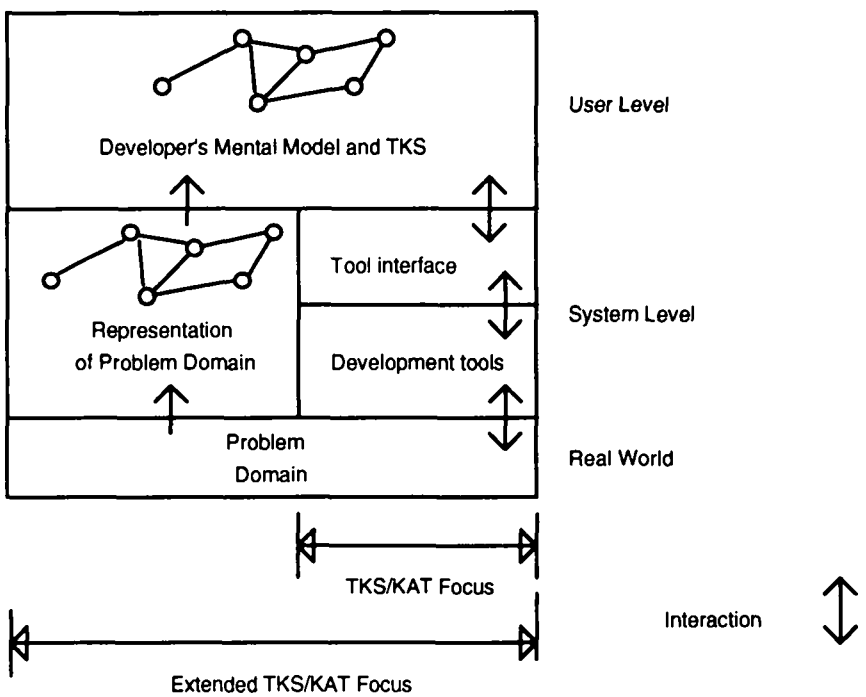

Figure 1 Interaction/knowledge transfer in a complex developmental system

When analysing the user's knowledge of a system, to what extent should we be analysing knowledge of the problem domain, compared with analysing knowledge of the tools? It can be argued the most effective performance of tasks occurs when the user feels as if he or she is

---

[3] See Norman's (1986) 'Theory of Action' for a framework for discussing task "execution" and "evaluation".

[4] Non-expert systems, such as CASE tools and the LON development system (see example below), can often check a design for consistency with "how to" knowledge on the syntactic level.

manipulating the problem domain directly. The tool should appear **transparent** to the user. This transparency can be a result of thorough familiarity with the tool so that the knowledge in the procedural substructure is executed automatically as a block. This transparency can also be a result of an appropriate, well-learned metaphor or the result of a direct manipulation interface style. Such a style tries to minimise the number of conceptual levels operating between the user and the problem domain.

To produce the tool-transparency of a direct manipulation style interface, we therefore need to give primacy to task analysis from the perspective of the problem domain. That is, in any given interaction, we are more concerned with describing the object in the problem domain that is being viewed or manipulated, than with describing the tools being used to manipulate the object. There are also design methodology imperatives for focussing on the problem domain rather than the tools; tools change, whereas the problem domain tends to be more stable. Software built on stable objects is more extensible (Meyer, 1988).

The knowledge of tasks, as Johnson describes it, is a knowledge of the application of *tools* to a problem domain. It does not take into account the knowledge a user may have of the problem domain that is *independent* of the user's previous tool-level interaction with that domain. Such knowledge - a mental model the user develops of the problem domain - can be represented in similar terms to a TKS taxonomy, but needs to be described in more complex terms than a simple response to a task-action. To Johnson, a mental model (e.g. a desktop metaphor) is a mechanism for transferring knowledge of tasks (Johnson 1989, p. 179), rather than informing the user about the state of the system (e.g. the state of the file system) so he/she can acquire and execute the task[5].

How will the process of task analysis differ when the focus is on analysing the user interaction with the problem domain rather than the user's interaction with the tools? The analysis of tasks (object-action pairs) will be an analysis of the actions on problem domain objects, rather than a description of user's manipulation of the tool. If the source of the data is the detailed observation of the user's interaction with a system, the analyst will need to abstract the task knowledge of the problem domain from the user's interaction with the tool. More accessible sources for problem domain task knowledge are likely to be problem-domain descriptions, documentation, existing system-images and so on. In the latter case, the domain objects are likely to be described in a way that is already generified: in Johnson's terms the objects/actions will already be *central* and *representative*. In this case, the role of observational task analysis of subject interaction would be limited to the verification that the knowledge (as represented by the various KAT substructures) is the same in the documentation/system-image as it is in the subject's head. Even if the knowledge representations differ, it is problematic whether this difference is the result of the inadequate *learning* of the knowledge by the subject, or inadequacies in the documentation/system-image representation of that knowledge. In summary, in a structurally complex domain the interface needs to provide a representation of the problem domain. The taxonomic

---

[5] Johnson makes no mention of the concept of a mental model other than metaphorical models. He points out that "Metaphorical relations do not, themselves, identify general or any knowledge in detail .. [and] should be thought of as a mechanism for facilitating transfer rather than as a means of identifying knowledge that could be transferred." (Johnson, 1989, p176)

substructure will be derived from the abstractions of the problem domain in a form that is already generic.

Opportunism

Following Johnson's approach, we wish to describe the **knowledge** required for the execution of tasks. In particular, we wish to describe the tasks related to software system development. Researchers have identified the highly opportunistic nature of the software design process (Guindon Krasner & Curtis, 1987; Visser, 1991; Khushalani, Smith & Howard, 1994) wherein the designer bounces back and forth across different levels of abstraction in developing the design. Top-down balanced development only occurs when the designer fully understands the problem. Johnson's goal substructure cannot, in this case, be taken as a high level procedural description, although obviously the user/designer uses some such structure to keep track of the progress of the development.

In attempting to describe the tasks involved in the development of a system, and in providing a suitable interface for the carrying out of these tasks, it is not possible to describe a **single** sequence of procedures. We can, however, describe a set of goals and sub-goals that needs to be achieved in order to complete the task. While the process of development as a whole may not be a linear sequence, there are set sequences of tasks that must be followed in a particular development stage (e.g. the function of a node must be determined before one can link it into a network). These prerequisites, or 'enabling states' as Johnson refers to them, can provide us with the basis of a generalised goal substructure.

The procedural substructure, as Johnson defines it, is a description of tasks executed as a single block. Tasks described at the problem-domain level do not usually exhibit such automatic sequences, but need to be continually re-evaluated in terms of the goals. Procedural substructures are therefore more relevant to a task description at the tool level. For example, a trip in a car can be described at the tool level as a series of actions manipulating interface objects such as the steering wheel, gear lever, brake pedal etc. From the user's point of view, such actions, after a while, become automatic with no conscious thought required. They are also easily transferable from one car to the next, because of the similarity of car-user interfaces. On the other hand, from the problem-domain level, the description of the trip is a description of the route from A to B, with the turnings at road intersections continually needing to be evaluated in terms of getting to the goal B, given problem-domain knowledge such as shortest route, traffic conditions, etc.. The driver needs feedback on the current state of the problem domain (he/she needs to know where they are on the road), and this understanding of the status of the problem domain is fundamental to applying their goal knowledge.

The dynamics of an interface for a complex developmental system needs to support the opportunistic nature of the interaction, thus the description of the dynamics of the interface at problem-domain level needs to be goal, rather than procedurally, oriented.

Complexity - Context and Multiple System-images

In a complex system like a development system the user/designer needs a complex and theoretical mental model of the problem domain in order to interact with it, because the problem domain *is* complex and theoretical. Such a model may be independent from the learning or performing of tasks. An interface needs to represent this model; the information

on the ***state*** of the problem domain the interface gives back to the engineer is critical to the usability and learnability of the interface. Knowledge of the state of the problem domain is knowledge that gives ***context*** to the task being undertaken. As Johnson points out (1989, p.165): "A task can be more precisely identified and defined by taking into account the context in which it is found to occur". The context of a task can be defined from any number of perspectives. Johnson (1989, p168) mentions role, temporal, experiential contexts, but makes no mention of the context provided by a mental model of the problem domain. In a complex system, the inter-object relations that are specific to the problem domain (that make up a context) need to be modelled in both the user's head and the interface (Figure 1). If there are a number of system-images provided by an interface, then these images should also be contextualised within the system as a whole. In other words the links between the various system-images should be "visible" from within a particular system-image. If a system develops over time, as in a developmental system, then a temporal or goal context also needs to be provided.

The representations of the problem domain, as it develops over time, can be thought of as a series of *system-images*. A system-image is a representation of a set of objects, with relationships linking those objects. A system-image needs to provide *context* for the user and to comprehensively describe one aspect of the system. System-images will vary depending on the nature of the problem domain, but in the case of the example studied (see below) these vary depending on:

- the stage of development;
- the type of model (e.g. logical or physical);
- the level of detail.

System-images of different model types should describe a mutually exclusive set of relationships. On the other hand, system-images at different stages of development, or different levels of detail, should contain the same set of objects and relationships. As our ultimate aim is to develop a (set of) system representations from these objects and their relationships (an object model) we have derived from the taxonomic substructure, we need to classify objects in terms of the sort of representation to which they belong.

Difficulties arise in defining the relationship **between** different system-images, because the *objects* in the system-image, and the *notation* with which we refer to those objects, may differ depending on which system-image we are using. If these difficulties are to be overcome, system-images need to provide the following:

- Context - a particular system-image should be able to be contextualised within the system as a whole. In other words, not only should the context of interface objects be contextualised within a system-image, but the links between various system-images should be "visible" from within a particular system-image.
- Stability - there should be a core of objects whose user representation is stable over the development process and common to various images of the system. For Shneiderman (1987), the *continuous representation of the object of interest* is a key property of direct manipulation interfaces.

Object-oriented analysis suggests a solution to this problem of linking system-images. By first defining the *objects* in a system, we create greater *stability* of our system representation

than can be achieved through approaches such as functional decomposition.[6] Objects that are common to the various system-images should be used as the means of linking the various system-images. Ideally, system-images of a network should be like different perspectives of the same network (Figure 2). These perspectives would give different views of the same objects, depending on the stage in the development process, the model used, and the level of detail described.

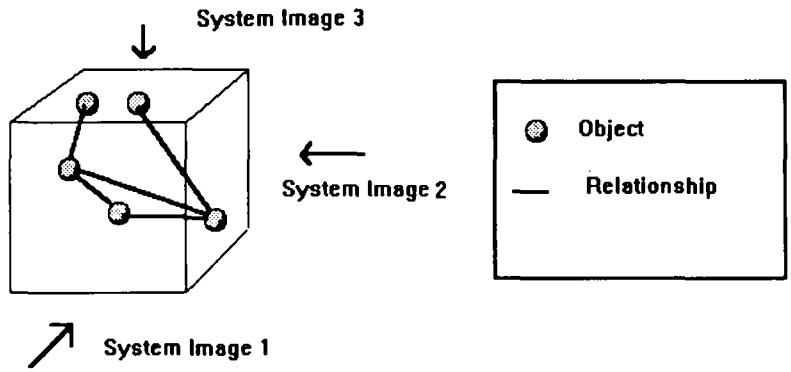

Figure 2 Different system-image perspectives of common objects in a system
Relationships between objects will differ depending on system-image.

The stability of the objects across the different system-images provides the link between those system-images. The process of interface analysis and design becomes a process of **elaboration** of the same objects and the relationship between those objects from different perspectives. This linking provides the context of a system-image in relation to other system-images.

Johnson (1992, p182) suggests the identification of representative, central and generic properties of tasks, so that task knowledge is generalised, and thus transferable (learnable) between applications. As stated above, in complex systems that change during the development process, we also need to be concerned with knowledge transfer from the system to the user, so that the user can maintain his or her representation of the problem domain. The identification of task-objects that are visible in different system-images, can provide a basis for the identification of interface objects.

Analysis and Design Guidelines

In view of the above characterisation of a complex developmental system we can identify the following broad analysis and design guidelines for the interfaces of such systems:

(1) The focus of the task analysis and design should be the problem domain rather than the development tools. Development tools should be as transparent as possible.

(2) Problem domain **objects** used in tasks, should be the initial focus of analysis, rather than functionality or procedures. The identification of interface objects should be on the basis of representativeness, centrality and stability across system-images. The description of the dynamics of the interface should initially be goal, rather than procedurally, oriented

[6] Meyer (1988) makes this points with regard to the software development process. The same insight seems applicable to the design and implementation in complex domains.

(3) The representation of problem domain objects should be:

- **encapsulated** in interface objects. The interface should be a "direct manipulation" so that, as far as possible, the objects identified in the taxonomic knowledge substructure are represented by interface objects.
- **contextualised** in relation to other objects. The relationships between problem-domain objects should be explicitly represented by the interface as objects.
- **notationally stable** over different stages of the development process and across different models of the system. Where an object is present in more than one system-image, such a stable representation will provide the link between different system-images.

A REVISED VERSION OF KNOWLEDGE ANALYSIS

Given the nature of complex development systems discussed above, we can summarise the modifications to the knowledge analysis phase of KAT, in Table I (below).

	KAT	Proposed method
Problem Domain	Simple (no mental model required - only knowledge of tool level tasks)	Complex (mental model required)
	Unstructured	Structured/generified
	Static	Developmental
Design purpose of task analysis	Transfer of knowledge	Feedback on state of problem domain
Focus of analysis	User/tool interface	User/Problem domain interface
Analysis criteria	Representativeness	Visibility in various system-images
	Centrality	Stability

Table I A summary of the modifications to the knowledge analysis phase of KAT

The revised method can be summarised as follows:

(1) Identification of Task Knowledge

- identify objects and actions
- identify goal sub-structure

(2) Analysis of Knowledge

- different representations of the system have to be identified.
- objects and actions are analysed for centrality and representativeness. Such knowledge structures are already likely to be generic.

- analyse objects for visibility in, and stability across, different representations of the system (e.g. control and physical attributes) and at different stages of the development.
- analyse the visibility of the attributes of various objects at different stages of the system.

(3) Develop a TKS of taxonomic and goal knowledge, based on the stable objects in the system.

LINKING KAT WITH AN OBJECT-ORIENTED METHODOLOGY

We have seen how an extended KAT can be used as an input for the derivation of interface representations. Can the above revised method of knowledge analysis provide input to an object-oriented analysis? There is some morphological similarity between the KAT model of a user's knowledge, and the system representation of the Object Modelling Technique (OMT) of Rumbaugh et al (1991).

- Johnson's definition of the taxonomic substructure as identifying "the object properties and attributes, including class membership, the procedures in which it is commonly used, [and] its relation to other objects and actions..." (Johnson 1989, p.168) could be a definition of the Object Model if applied to problem domain or system representations. OMT's object model diagrams provide a useful tool for representing the complexity of objects and relationships in a taxonomic substructure (see Figure 4 below).
- The Procedural Substructure has some similarity with the Dynamic Model in OMT (for "learnt procedure for a well-practised task" read "program control"). The major difference is that in KAT, procedures are small single executable units (an attempt to maintain psychological validity for the model), whereas a computer system has no such limits on the size of memory chunk thus its executable units can be much larger. In this sense there is some overlap between the Dynamic Model in OMT and the Goal Substructure in KAT.
- Both the Goal Substructure in KAT and Functional Model in OMT are logical-level descriptions of *what* is to done. The Goal Substructure describes the *intentions* of the user as established by KAT (either articulated or observed behaviour), and the Functional Model describes how the *requirements* of the system are met, in terms of the transformation of input to output given the constraints[7].

In the case of task analysis at the problem-domain level, the **granularity** of the analysis means that we do not deal with the *procedural substructure* (tasks executed in a single unit). The mapping process for the input of KAT into Object Modelling Techniques (OMT) can be represented by the following block diagram (Figure 3).

It should be noted that task analysis is undertaken at a problem-domain and a tool level, producing two different TKS knowledge representations. The TKS at the problem-domain level is more fundamental and provides the semantic structure to the interface. The TKS at the tool level would describe the user knowledge of interaction styles used in the interface.

---

[7] The Goal Substructure is more important to interface design, than the OMT's Functional Model has proved to be in practice to system design. - see Graham (1994, pp238-9)

The more the style matches existing standards, with which the user may be familiar (e.g. Windows), the easier the interface will be to learn.

The OMT's object model notation provides a rich notation for describing the objects in the taxonomic substructure: their properties, attributes, class membership, procedures and relationships to other objects and actions. By using the same notation for both the taxonomic substructure and the object model, what is in the user's head can have a more direct mapping to the system representation.

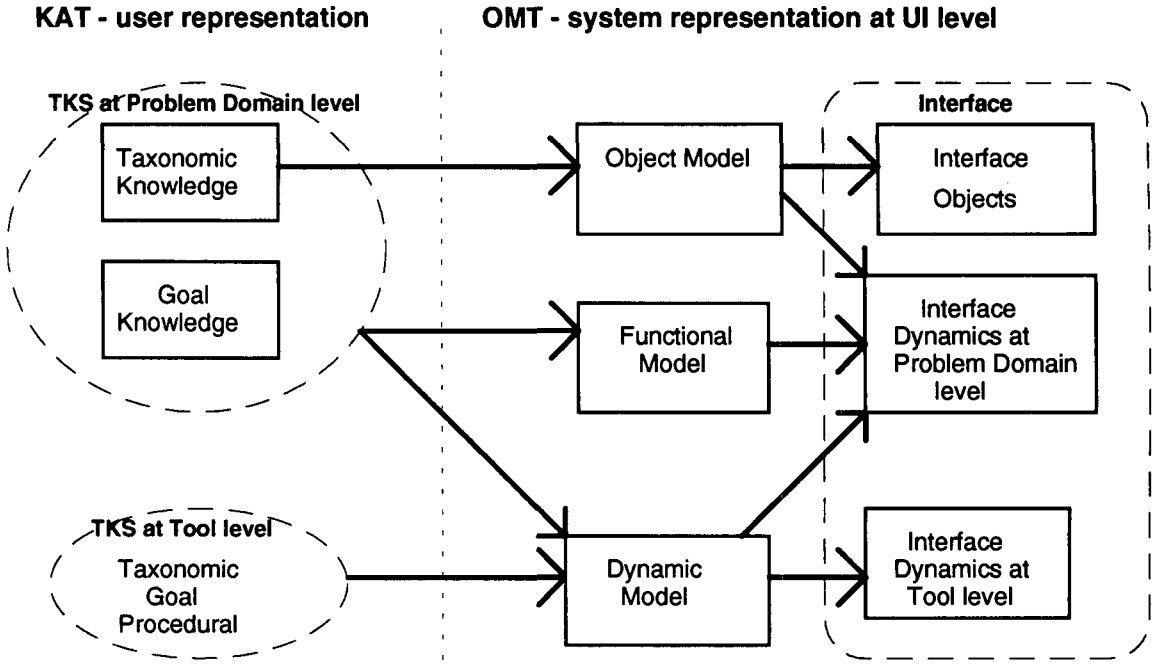

Figure 3 Mapping between the elements of KAT and OMT

While the distinction between tool and problem-domain task analysis may be useful analytically, in the design of the interface artefact these aspects of the interface need to be integrated. Further work needs to be done to examine the relationship between the dynamic model of a user interface, and the user's procedural and goal substructures.

AN EXAMPLE

The *complex development system* on which the task analysis was originally undertaken for this paper was the LONBUILDER™ system developed by Echelon Corporation for the design and construction of LON intelligent distributed control networks. LONBUILDER provides an integrated environment for a set of tools used for node and network software design and implementation.

The following example is a simplified version of the analysis done on the LON development system. The example illustrates the results of the main stages of the revised KAT task analysis as outlined above.

Identification of Problem-Domain Objects and Goal Substructure

Analysis of the existing system was undertaken to identify objects in the problem domain, and a high level goal knowledge. The problem-domain objects were derived from the

existing system and its documentation, and thus were already generified as 'representative' and 'central'. These descriptions have not been reproduced here, but the high-level goal description provided the basis for the identification of different developmental stages, and thus a classification for the different system-images. These images consisted of a physical model and control (logical) model of the system, as well as a series of stages (goals) for a node developer, and a series of stages (goals) for a someone using the system to install and configure a network.

Classification of Objects into Visibility in System-images and the Identification of Stability of Objects Across System-images

Once the problem-domain objects and system-images had been identified, the objects were analysed for visibility in, and stability across, system-images using the grid (Table II).

Object	Aggregation/ Attribute	System-image												
		Model		Node Development Stage						Installation Stage				
		Control	Physical	1	2	3	4	5	6	1	2	3	4	5-7
Intra - nodal objects														
application node		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	name	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	self-doc text string	✓						✓	✓				✓	
	physical location		✓								✓	✓	✓	✓
	function	✓			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	logical address	✓							✓		✓	✓	✓	✓
	target hardware		✓	✓	✓	✓	✓	✓	✓					
Neuron chip			✓				✓	✓						
	eeprom		✓				✓	✓						
	ram		✓				✓	✓						
	application buffers		✓					✓	✓					
	network buffers		✓					✓						
	rom (optional)		✓					✓	✓					
	clock and timers		✓					✓	✓					
	application i/o block		✓					✓	✓					
	scheduler	✓						✓	✓					
	watch dog timer	✓						✓	✓					
i/o circuitry			✓				✓	✓	✓					
application image		✓					✓	✓	✓	✓	✓			✓
actuator		✓	✓	✓	✓			✓	✓	✓	✓			✓
sensor		✓	✓	✓	✓			✓	✓	✓	✓			✓
router node		✓	✓				✓	✓	✓					
nw manage node		✓	✓							✓	✓			✓
transceiver			✓			✓		✓	✓		✓	✓		
Inter-nodal Objects														
media			✓					✓	✓		✓	✓		✓
network connection	network variable	✓				✓	✓	✓	✓		✓		✓	✓
	explicit message	✓						✓	✓		✓			✓
LONTalk protocol		✓	✓				✓	✓	✓		✓		✓	✓
network image		✓							✓	✓	✓		✓	✓

Table II - Object visibility and stability grid for various system-images

It will be noted for the above grid that the *application node* is the only problem-domain object present in all system-images. If we wish to build an interface where notationally stable objects provide the context between different system-images, in this case the application node needs to have a consistent representation through all system-images. The different system-images would represent an elaboration the relationships between the application nodes, with the level of detail of each image indicated by the visibility, or otherwise, of other objects and attributes.

Taxonomic substructure / Object model

An example of the object model notation used to described a LON system is illustrated below. Such a diagram could be the product of KAT task analysis or object-oriented analysis of the problem domain. The diagram has been simplified to show the objects only (not their attributes or methods). In a complete diagram, the boxes representing objects would also contain the attributes of the object, and the actions (tasks) performed on the object, as shown in the key to the diagram. Such a diagrammatic notation allows a much richer description of the objects than a simple listing of object-action pairs.

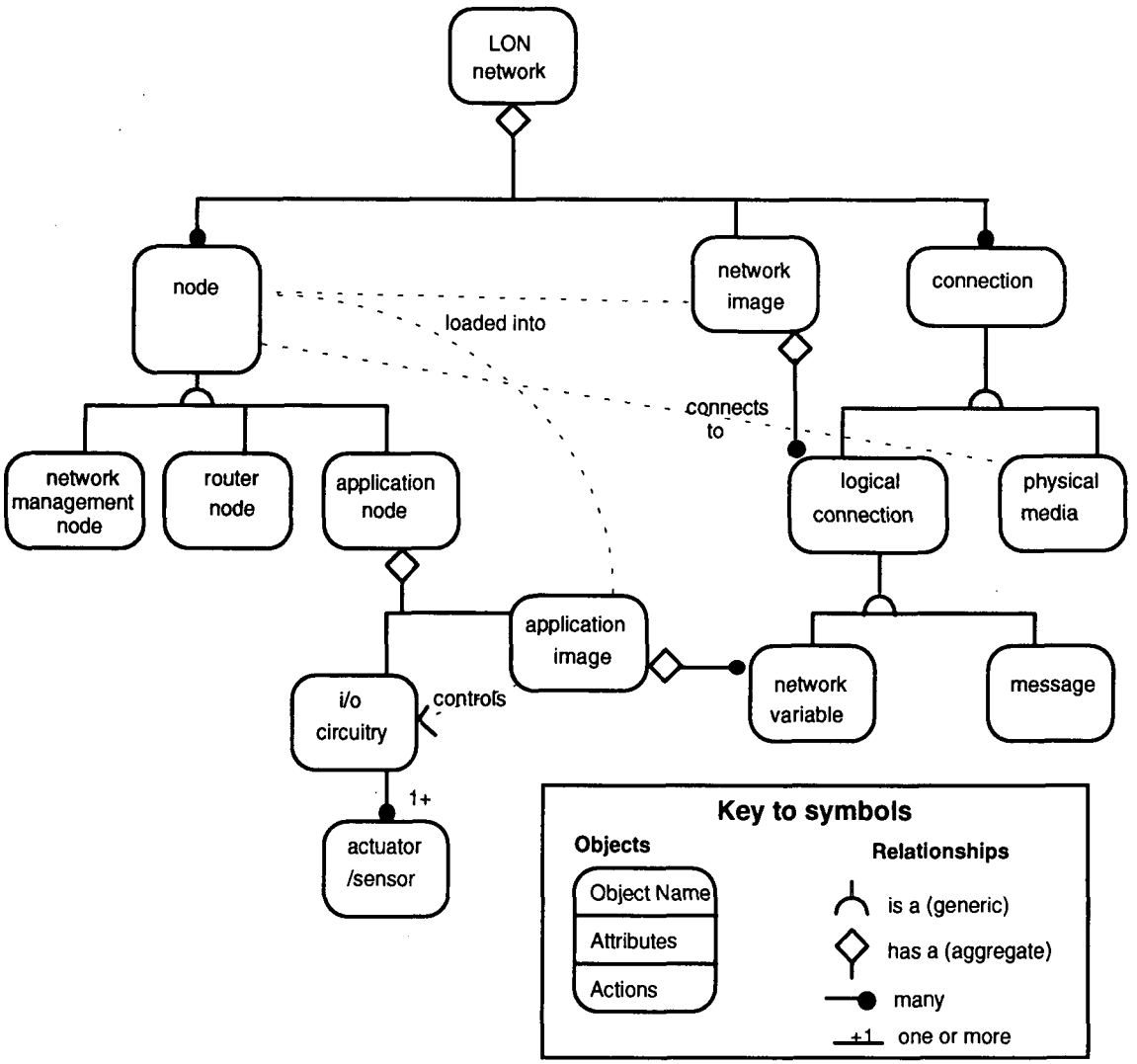

Figure 4 Top level taxonomy of problem domain using simplified OMT notation

Each of these objects can then be given a detailed taxonomic description depicting the object's attributes, the actions performed on the object, and the context of the object (superordinate categories, membership of system-images, relationships and so on).

Deriving interface objects

As with the objects themselves, the attributes, actions and relationships of an object can be analysed using a visibility grid similar to the one above. From such a grid we can then derive the characteristics of a representation of the object must have in a particular system-image - the *appearance* of the object. In other words, we can derive the problem-domain-level attributes and behaviours of an interface object. If the process is followed, we should have interface objects that match the user's taxonomic knowledge of the problem domain, and provide a image of the object appropriate to the particular sub-goal of the user (or stage of development from a system view-point). To develop a complete system-image we need to analyse the *appearance* of all objects, and relationships between those objects, visible at a particular sub-goal stage.

The description of the interface given by such a process can be thought of as a series of snap-shots of system states each relating to user sub-goal. What we know about the interface at that stage is which objects and relationships are visible, the *appearance* of those objects in terms of problem-domain attributes, and the problem-domain-level actions we can perform on the objects and relationships. It is important to note that while the analysis may provide an input into the problem-domain content and structure of the interface, we have derived a description of the interface at the problem-domain level - rather than a graphical representation of the objects, or their behaviour at a tool level.

CONCLUDING REMARKS

KAT type analysis of knowledge can provide an input to the object-oriented analysis and design process, and the construction of the interface. By developing a taxonomic substructure based on stable objects, and describing it in a notation consistent with object-oriented analysis, we can provide an input to the object model of the system and thus to the design of the interface objects and system-images. It may also provide an input to the dynamic model of the interface.

The identification-of-knowledge stage of KAT needs to take account of the degree of generification or abstraction already present in the problem domain. It should also differentiate between knowledge of *tools* and knowledge of the *problem domain.*

To provide a design input with complex developmental systems, the analysis of knowledge stage of KAT needs to be extended to take account of

- the provision of feedback to the user's mental model;
- different subsets of that mental model as represented by system-images;
- the stability of objects across those system-images.

There is strong compatibility and complementarity between KAT and OMT. While the focus of these techniques is different, they have the potential to develop a mutually beneficial relationship if combined in the development process. KAT can offer OOA more rigour in

ensuring user-centredness. OMT can offer a framework and notation through which the task analysis using KAT can be used in the development of software systems.

Further development of the revised KAT methodology is currently under consideration, involving its application to a particular design. Specific work to be considered includes the following:

- validating the efficacy of the revised KAT methodology by applying it to a real-world interface such as LONBUILDER;
- further exploring the role of task knowledge in creating the dynamic model of an interface;
- exploring the applicability of the method to other problem domains.

REFERENCES

Card S., Moran T., & Newell A.(1979) **The Keystroke Level Model for User Performance Time with Interactive Systems.** Report SSL-79-1 (March) Xerox PARC.

Carroll J. & Olson J. (1988) "Mental models in human-computer interaction", in Helander, M. (Ed.), **Handbook of Human-Computer Interaction,** Amsterdam: Elsevier Science Publishers pp. 45-65.

Diaper, D. (1989) **Task Analysis for Human-Computer Interaction,** Chichester: Ellis Horwood.

Graham, I. (1994) **Object Oriented Methods**, Second Edition, Wokingham: Addison Wesley.

Guindon, R., Krasner, H, & Curtis, B. (1987) "Breakdowns and processes during the early activities of software design by professionals", in Olson, G.M., Sheppard, C. and Soloway, E. (Eds) **Empirical Studies of Programmers, Second Workshop**, Norwood, N.J.: Ablex, pp 65-82.

Johnson, P. (1989) "Supporting system design by analysing current task knowledge", in Diaper, D. (Ed.) **Task Analysis for Human-Computer Interaction**, Chichester: Ellis Horwood, pp.160-184.

Johnson, P (1992) **Human Computer Interaction - Psychology, Task Analysis and Software Engineering**, London: McGraw-Hill Book Company.

Khushalani, A., Smith, R. & Howard, S. (1994) "What happens when designers don't play the rules: Towards a model of opportunistic behaviour in design", **Australian Journal of Information Systems,** Vol 1, No 2, pp.13-31.

Larkin, J. & Simon, H. (1987) "Why a diagram is (sometimes) worth ten thousand words", **Cognitive Science**, Vol 11, pp.65-99.

Meyer, B. (1988), **Object-oriented Software Construction**, Eaglewood Cliffs, N.J.: Prentice-Hall.

Norman, D.A. (1986) "Cognitive Engineering" in Norman and Draper (Eds.), **User Centered Design,** Hillsdale, N.J.: Lawrence Erlbaum Associates.

Payne S. and Green, T. (1989) "Task-Action Grammar: the model and its developments" in Diaper, D. (Ed.) **Task Analysis for Human-Computer Interaction**, Chichester: Ellis Horwood, pp.75-105.

Rumbaugh J., Blaha M., Pemerlani W., Eddy F., & Lorensen W. (1991) **Object-oriented Modelling and Design,** Eaglewood Cliffs, N.J: Prentice-Hall.

Shneiderman, B. (1987) **Designing the User Interface**, Addison Wesley.

Visser, W. (1991) "The cognitive psychology viewpoint on design: Examples from empirical studies", in Gero, J. (Ed.), **Artificial Intelligence in Design**, Oxford: Butterworth-Heinemann, pp.505-524.