A Framework For Knowledge Reuse

John Debenham

School of Computing Sciences
University of Technology, Sydney,
PO Box 123 Broadway, NSW 2007, Australia
debenham@socs.uts.edu.au

Abstract

Knowledge 'objects' are proposed as a modelling framework which facilitates the reuse of knowledge. Knowledge objects are operators which may be used to construct the 'items' in the conceptual model. Items are a single framework for describing the data, information and knowledge things in an application. The conceptual model is constructed by applying object operators to basic 'data items'. Items are expressed in terms of their particular 'components'. The knowledge embedded within a knowledge item can not be readily applied in the context of different components. Items do not facilitate reuse. Objects are not expressed in terms of particular components and do facilitate reuse.

Introduction

Conceptual modelling is the second step in a complete, four-step design methodology for knowledge-based systems. The conceptual model is "unified" in the sense that no distinction is made between the 'knowledge component' and 'database component'. In this conceptual model the 'knowledge', the 'information' and the 'data' are represented in the same way. The conceptual model is expressed in terms of "items" and "objects" which are described in the following sections. Both items and objects contain two classes of constraints; thus the conceptual model employs constraints in a uniform way for both 'knowledge', 'information' and 'data'. The work described here has a rigorous, formal theoretical basis expressed in terms of the λ-calculus [1] [2]; the work may also presented informally in terms of schema. Schema are used to construct a conceptual model in practice.

Items are expressed in terms of their particular 'components'. The knowledge embedded within a knowledge item can not be readily applied in the context of different components. Items do not facilitate reuse. Objects are not expressed in terms of particular components and do facilitate reuse. Objects are a context independent, well defined framework that can be used to describe either simple rules or complex rule bases.

A *first generation methodology*, developed in a collaborative research project between the University of Technology, Sydney and the CSIRO Division of Information Technology; that methodology did not address reuse effectively. That methodology was supported by a Computer Assisted Knowledge Engineering tool, or CAKE tool. An experimental version of this tool was trialed in a commercial environment. A *second generation methodology* takes a unified approach to design. It generates systems that are inherently easy to maintain, and employs knowledge constraints [8] to further protect the system against the introduction of update anomalies. The second generation methodology addresses knowledge reuse by representing knowledge as object operators.

The terms 'data', 'information' and 'knowledge' are used here in a rather idiosyncratic sense. The *data* in an application are those things which are taken as the fundamental, indivisible things in that application; data things can be represented as simple constants or variables. The *information* is those things which are "implicit" associations between data things. An *implicit* association is one that has no succinct, computable representation. Information things can be represented as tuples or relations. The *knowledge* is those things which are "explicit" associations between information things or data things. An *explicit* association is one that has a succinct, computable representation. Knowledge things can be represented either as programs in an imperative language or as rules in a declarative language.

Conceptual Modelling

If a conceptual model represents only the individual data, information and knowledge things and the relationships between those things then that model is called *hierarchic*. An *item* is a representation in a hierarchic conceptual model of a data, information or knowledge thing. A hierarchic conceptual model is 'hierarchic' in the sense that information items are "built out of" data items, and that knowledge items are "built out of data and information items". In a hierarchic model the data items which are used to build an information item are called the *components* of that information item. Likewise the data or information items which are used to build a knowledge item are called the *components* of that knowledge item [9].

A knowledge-based systems design methodology should address reuse [10]. A design methodology will specify the form of its conceptual model. The particular form of the conceptual model used by the methodology will influence both the cost of representing knowledge, and the cost of subsequently reusing that knowledge. A "unified conceptual model" is described which facilitates knowledge reuse.

The *unified conceptual model* consists of:

- a conceptual view, and
- a coupling map.

The *conceptual view* is a collection of items each of which represents a data, information or knowledge thing. The *coupling map* is a vertex labelled graph. For each item in the conceptual view there is a vertex in the coupling map labelled with that item's name. A *coupling relationship* links two items in the conceptual model if modification to one item will, in general, require that the other item should at least be checked for correctness if the integrity of the conceptual model is to be preserved. The coupling relationships form a chaining structure: a modification leads to checking which may in turn lead to further modifications and hence to further checking and so on. If a coupling relationship exists between two items then that coupling relationship is represented in the coupling map as an arc which joins the corresponding two vertices. The coupling map is used to support maintenance.

In §2.1 the unified representation for items is described. Items, together with the notion of item normalisation described below, fail to provide an adequate basis for the simplification of the coupling map. Items are built up from components; the component structure of items means that the expression of a knowledge item will be inextricably involved with its components. As an item's components are an integral part of that item, items do not facilitate knowledge reuse. To overcome this inadequacy "objects" are introduced in §2.2. Objects are item building operators.

Items and objects are introduced now in general terms. Item names are written in italics. Object names are written in bold italics. Suppose that the conceptual view already contains the item "*part*" which represents spare part things, and the item "*cost-price*" which represents cost price things; then the information thing "spare parts have a cost price" can be represented by "*part/cost-price*" which may be built by applying the "***costs***" object to *part* and *cost-price*:

part/cost-price = ***costs***(*part*, *cost-price*)

Suppose that the conceptual view already contains the item "*part/sale-price*" which represents the association between spare parts and their corresponding selling price, and the item "*mark-up*" which represents the data thing a universal mark-up factor; then the knowledge thing "spare parts are marked up by a universal mark up factor" can be represented by *[part/sale-price, part/cost-price, mark-up]* which may be built by applying the "***mark-up-rule***" object to the items "*part/sale-price*", "*part/cost-price*" and "*mark-up*":

[part/sale-price, part/cost-price, mark-up] =
mark-up-rule(*part/sale-price*,
part/cost-price, *mark-up*)

Items and objects are described formally in §2.1 and §2.2.

The conceptual view consists of items. A fundamental set of data items in the conceptual view are called the "basis". The remaining items in the conceptual view are built by applying object operators to other items in the conceptual view.

Items

Items are a formalism for describing the things in an application. They have three principal properties: items have a unified format in that they may represent data, information or knowledge things [2]; items incorporate two powerful classes of constraints, and a single rule of "normalisation" can be specified for items. The key to this unified representation is the way in which the "meaning" of an item, called its *semantics*, is specified.

Items have a name which by convention is written in italics. The semantics of an item is a function which *recognises* the members of the "value set" of that item. The value set of an item will change in time τ, but the item's semantics should remain constant. The value set of an information item at a certain time τ is the set of tuples which are associated with a relational implementation of that item at that time; for example, the value set of the item named *part/cost-price* could be the set of tuples in the "part/cost-price" relation at time τ. Knowledge items have value sets too. Consider the rule "the sale price of parts is the cost price marked up by a universal mark-up factor"; suppose that this rule is represented by the item named *[part/sale-price, part/cost-price, mark-up]*, then this item could have the value set of quintuples that satisfy this rule. This idea of defining the semantics of items as recognising functions for the members of their value set extends to complex, recursive knowledge items too [2]. Items thus provide a unified framework for describing all of the things in the conceptual model. An operation on items is defined in the following section in terms of the λ-calculus representation.

Formally, given a unique name A, an n-tuple $(m_1, m_2, \ldots, m_n)$, $M = \sum_{i=1}^{n} m_i$, if:

- S_A is an M-argument expression of the form:

$$\lambda y_1^1 \ldots y_{m_1}^1 \ldots y_{m_n}^n \bullet [S_{A_1}(y_1^1, \ldots, y_{m_1}^1) \wedge \ \ldots\ldots \wedge \\ S_{A_n}(y_1^n, \ldots, y_{m_n}^n) \wedge \ J(y_1^1, \ldots, y_{m_1}^1, \ldots, y_{m_n}^n)] \bullet$$

where $\{A_1, \ldots, A_n\}$ is an ordered set of not necessarily distinct items, each item in this set is called a *component* of item A;

- V_A is an M-argument expression of the form:

$$\lambda y_1^1 \ldots y_{m_1}^1 \ldots y_{m_n}^n \bullet [V_{A_1}(y_1^1, \ldots, y_{m_1}^1) \wedge \;\ldots\ldots \wedge\; V_{A_n}(y_1^n, \ldots, y_{m_n}^n) \wedge \; K(y_1^1, \ldots, y_{m_1}^1, \ldots, y_{m_n}^n)]\bullet$$

where $\{A_1, \ldots, A_n\}$ are the components of item A, and

- C_A is an expression of the form:

$$C_{A_1} \wedge C_{A_2} \wedge \ldots \wedge C_{A_n} \wedge (L)_A$$

where L is a logical combination of:

  - Card lies in some numerical range;
  - Uni(A_i) for some i, $1 \leq i \leq n$, and
  - Can(A_i, X) for some i, $1 \leq i \leq n$, where X is a non-empty subset of $\{A_1, \ldots, A_n\} - \{A_i\}$;

subscripted with the name of the item A, and "Uni(a)" is a *universal constraint* which means that "all members of the value set of item a must be in this association", "Can(b, A)" is a *candidate constraint* which means that "the value set of the set of items A functionally determines the value set of item b", and "Card" means "the number of things in the value set". The subscripts indicate the item's components to which that set constraint applies.

then the named triple $A[\, S_A, V_A, C_A]$ is an n-adic *item* with *item name* A, S_A is called the *semantics* of A, V_A is called the *value constraints* of A and C_A is called the *set constraints* of A.

For example, an application may contain an association whereby each *part* is associated with a *cost-price*. This association could be subject to the "value constraint" that parts whose part-number is less that 1,999 will be associated with a cost price of no more than $300. This association could also be subject to the "set constraints" that every part must be in this association, and that each part is associated with a unique cost-price. This association could be represented by the information item named *part/cost-price*; the λ-calculus form for this item is:

name		item name
name1	*name2*	item components
x	y	dummy variables
meaning of item		item semantics
constraints on values		value constraints
set constraints		set constraints

Figure 1 i-schema format

$$\textit{part/cost-price}[\; \lambda xy\bullet[S_{part}(x) \wedge S_{cost\text{-}price}(y) \wedge \text{costs}(x, y)]\bullet,$$
$$\lambda xy\bullet[\; V_{part}(x) \wedge V_{cost\text{-}price}(y) \wedge ((x < 1999) \rightarrow (y \leq 300))\;]\bullet,$$
$$(\text{Uni}(\textit{part}) \wedge \text{Can}(\textit{cost-price}, \{\textit{part}\}))_{part/cost\text{-}price}\;]$$

The λ-calculus form for items is not intended for practical use. In practice items are presented as i-schema. The i-schema format is shown in Figure 1. The i-schema for the item *part/cost-price* is shown in Figure 2. Rules, or knowledge, can also be defined as items. The i-schema for the knowledge item *[part/sale-price, part/cost-price, mark-up]* is also shown in Figure 2; this i-schema has four set constraints.

Objects

Items and objects may be used in such a way as to facilitate knowledge reuse. Objects are item building operators. They have four principal properties: objects have a unified format no matter whether they represent data, information or knowledge operators; objects incorporate two powerful classes of constraints; objects enable items to be built in such a way as to reveal their inherent structure, and a single rule of 'normalisation' can be specified for objects. Objects may either be represented

part/cost-price	
part	*cost-price*
x	y
costs(x,y)	
x<1999 → y≤300	
∀	
------------	o

[part/sale-price, part/cost-price, mark-up]		
part/sale-price	*part/cost-price*	*mark-up*
(x, w)	(x, y)	z
→ (w = z × y)		
→ w > y		
∀	∀	
------------------------------------		o
o	------------------------------------	
------------	o	------------

Figure 2 i-schemas

name		object name
(tuple1)/type1	(tuple2)/type2	tuple type pairs
meaning of object		object semantics
constraints on variables		value constraints
set constraints		set constraints

costs	
$(x)/\mathbf{X}^1$	$(y)/\mathbf{X}^1$
'x' costs 'y'	
$x<1999 \rightarrow y\leq300$	
$\forall$	
----------------	o

Figure 3 o-schema format and the object '***costs***'

mark-up-rule		
$(x,w)/\mathbf{I}^2$	$(x,y)/\mathbf{I}^2$	$(z)/\mathbf{D}^1$
$\rightarrow\ w = z \times y$		
$\rightarrow\ w > y$		
$\forall$	$\forall$	
------------------------------------		o
o	------------------------------------	
----------------	o	----------------

Figure 4 o-schema for object '***mark-up-rule***'

informally as "o-schema" or formally as λ-calculus expressions. Object names are written in bold italic script.

An n-adic object is an operator which maps n given items into another item for some value of n. The specification of each object will presume that the set of items to which that object may be applied are of a specific "type". The *type* of an m-adic item is determined both by whether it is a data item, an information item or a knowledge item and by the value of m; these types are denoted respectively by $\mathbf{D}^m$, $\mathbf{I}^m$ and $\mathbf{K}^m$. Unspecified, or free, type which is denoted by $\mathbf{X}^m$ is also permitted. The definition of an object is similar to that of an item. An *object* consists of: an object name, argument types, object semantics, object value constraints, and object set constraints. The *object name* is written in bold italics. The *argument types* are a set of types of items to which that object operator may be applied. The *object semantics* is an expression which recognises the members of the value set of any item generated by the object. The *object value constraints* is an expression which must be satisfied by all members of the value set of any item generated by the object. The *object set constraints* are constraints on the structure of the value set of any item generated by the object.

Objects may be defined formally as λ-calculus expressions in a similar way to the formal, λ-calculus definition of items. The formal definition of objects will not be described here. In practice objects are presented as o-schema. The o-schema format for objects is shown in Figure 3. If two objects have the property that the semantics of one logically implies the semantics of the other then the first object is a *sub-object* of the second. If two objects have the property that they are each sub-objects of each other then those two objects are said to be *equivalent*.

For example, the o-schema for the ***costs*** object is shown in Figure 3. The o-schema for the ***costs*** object contains two set constraints. The '∀' symbol has the obvious extension of meaning to its use for items; likewise for the horizontal line and the 'o' symbol. The o-schema for the ***mark-up-rule*** knowledge object is shown in Figure 4. Data objects provide a representation of sub-typing. For example, the ***cost-price*** object shown as a box with sloping sides in Figure 5 may be applied to the *price* item to generate the *cost-price* item which is a sub-type of *price*. The n-adic operator ***comp*** with trivial "constant 'true'" semantics is called the *compound operator*; it can be used to generate compound items from a set of n items.

Objects are used to construct the items in the conceptual view. A *conceptual view* consists of:

- a *basis*, which is a fundamental set of data items on which the conceptual model is founded;
- an *object library*, which is a set of object operators which are used to construct the items in the conceptual view with the exception of the items in the basis, and
- a *conceptual diagram*, which shows how the objects in the object library are used to construct the items in the conceptual view.

A simple conceptual diagram is shown in Figure 5.

Objects facilitate knowledge reuse. For example the ***mark-up-rule*** knowledge object described above may be applied to any three items of type $(\mathbf{I}^2, \mathbf{I}^2, \mathbf{D}^1)$ to generate a knowledge item. Any particular knowledge item generated in this way will contain the same basic wisdom in the ***mark-up-rule*** knowledge object. In [1] a join operator is defined. This operator enables knowledge objects to be combined so that they may represent combinations of complex rules. In theory, at least, an object operator can be constructed in this way to represent the essence of the knowledge in a entire knowledge base. Such an object operator may be applied to any appropriate set of basis data items to generate a particular knowledge base.

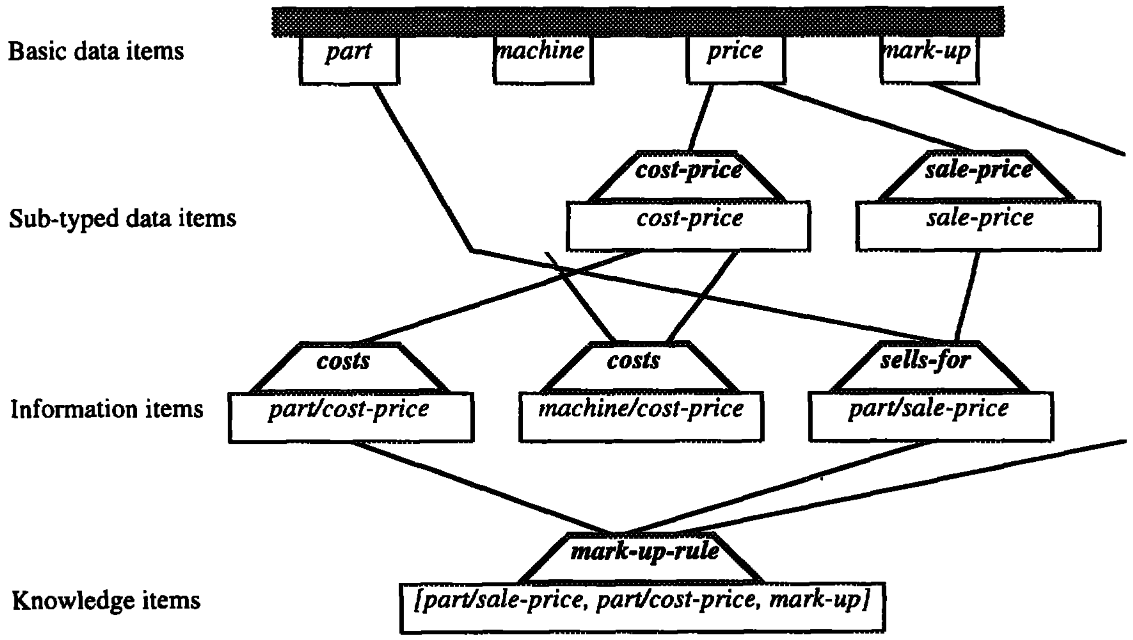

Figure 5 Simple conceptual diagram

References

1. Debenham, J.K. "Knowledge Simplification", in *proceedings 9th International Symposium on Methodologies for Intelligent Systems ISMIS'96*, Zakopane, Poland, June 1996.
2. Debenham, J.K. "Integrating Knowledge Base and Database", in *proceedings 10th ACM Annual Symposium on Applied Computing SAC'96*, Philadelphia, February 1996, pp28-32.
3. Debenham J.K. "A Unified Approach to Requirements Specification and System Analysis in the Design of Knowledge-Based Systems", in *proceedings Seventh International Conference on Software Engineering and Knowledge Engineering SEKE'95*, Washington DC, June 1995, pp144-146.
4. Lehner, F., Hofman, H.F., Setzer, R. and Maier, R. "Maintenance of Knowledge Bases", in *proceedings Fourth International Conference DEXA93*, Prague, September 1993, pp436-447.
5. Debenham, J.K. "Understanding Expert Systems Maintenance", in *proceedings Sixth International Conference on Database and Expert Systems Applications DEXA'95*, London, September 1995.
6. Kang, B., Gambetta, W. and Compton, P. "Validation and Verification with Ripple Down Rules", *International Journal of Human Computer Studies* Vol 44 (2) pp257-270 (1996).
7. Coenen F. and Bench-Capon, T. "Building Knowledge Based Systems for Maintainability", in *proceedings Third International Conference on Database and Expert Systems Applications DEXA'92*, Valencia. Spain, September, 1992, pp415-420.
8. Debenham, J.K. "Knowledge Constraints", in *proceedings Eighth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems IEA/AIE'95*, Melbourne, June 1995, pp553-562.
9. Tayar, N. "A Model for Developing Large Shared Knowledge Bases" in *proceedings Second International Conference on Information and Knowledge Management*, Washington, November 1993, pp717-719.
10. Katsuno, H. and Mendelzon, A.O. "On the Difference between Updating a Knowledge Base and Revising It", in *proceedings Second International Conference on Principles of Knowledge Representation and Reasoning, KR'91*, Morgan Kaufmann, 1991.
11. Debenham, J.K. *"Knowledge Systems Design"*, Prentice Hall, 1989.
12. Gray, P.M.D. "Expert Systems and Object-Oriented Databases: Evolving a New Software Architecture", in *"Research and Development in Expert Systems V"*, Cambridge University Press, 1989, pp 284-295.