

Knowledge Base Maintenance through Knowledge Representation

John Debenham

University of Technology, Sydney,
Faculty of Information Technology,
PO Box 123, NSW 2007, Australia
debenham@it.uts.edu.au

Abstract. The problem of maintaining a knowledge base is substantially concerned with keeping track of rules that share common wisdom. A knowledge representation is described in which a collection of rules that are based on common wisdom are represented as a single 'item'. For items, maintenance hazards, caused by one item being partially hidden within another, still remain. 'Objects' are introduced as item building operators so enabling these hidden links to be identified and made explicit. A single operation for objects enables some of these hidden links to be removed thus simplifying maintenance.

1. Introduction

Maintaining the integrity of a knowledge base is a complex practical problem. A *rule* is a chunk of knowledge in an if-then form. If knowledge is represented as rules then maintenance hazards occur when wisdom embedded in one rule is also embedded in another. A knowledge representation is described in which a collection of rules that are based on common wisdom are represented as a single 'item'. So a single item encapsulates a set of declarative rules, and in turn each rule encapsulates a set of imperative programs. The item representation makes no distinction between data, information and knowledge things [1]. The *data* things in an application are the fundamental, indivisible things. Data things can be represented as simple constants or variables. If an association between things *cannot* be defined as a succinct, computable rule then it is an *implicit* association. Otherwise it is an *explicit* association. An *information* thing in an application is an implicit association between data things. Information things can be represented as tuples or relations. A *knowledge* thing in an application is an explicit association between information and/or data things. Items make it difficult to analyse the structure of a whole application. To make the structure clear, 'objects' are introduced as item building operators. Objects are an abstraction representing the structure of knowledge. Items are *either* represented informally as "i-schema" *or* formally as λ -calculus expressions [2]. The i-schema notation is used in practice. Items contain two classes of constraints that apply equally to knowledge and to data. In [3] these constraints are generalised to fuzzy acceptability measures of knowledge base integrity.

For items, maintenance hazards, caused by one item being partially hidden within another, still remain [4]. The use of objects enables these maintenance links to be made explicit. A single rule for item and object decomposition enables some of these

links to be removed so simplifying maintenance [5]. Classical database normalisation [6] is a special case of the application of that single rule. In the 1980s there was considerable interest in building expert systems. At that time declarative formalisms, in particular if-then formalisms such as logic programming, provided one way of computing with knowledge that was far easier to use than imperative formalisms. The comparative ease of use of if-then formalisms was responsible for a misapprehension that knowledge should be thought of as “if-then stuff”. In a sense this is true because even if a chunk of knowledge has a number of if-then interpretations then it is unlikely that more than one of those interpretations will be useful in a particular application. One consequence of this misapprehension is that changes in the validity of a single rule that may not even be identified may have subtle and serious implications for the validity of a number of chunks of knowledge that have been identified and represented. So a knowledge representation with the property that a single chunk of knowledge, which may have a number of if-then interpretations, can be represented as a single entity should be a superior practical formalism for knowledge base design. The unified knowledge representation has this property. The unified representation is at a level of abstraction that is far closer to ‘reality’ than traditional declarative formalisms. This is shown by the hierarchy: a real chunk of knowledge is represented as a single “item”. Each item has a number of interpretations as if-then forms. Each if-then form, or rule, has a number of interpretations as imperative programs.

2. Declarative, rule-based formalisms

In a *declarative formalism* an if-then interpretation of a knowledge thing is represented as an “if-then” form as a rule. Two difficulties with declarative formalisms are illustrated using Horn clause logic as the representation formalism [7]. Logic is chosen because it is a widely understood notation, and *not* for any practical reason.

Consider the chunk of knowledge: [K1] “The sale price of a part is the cost price of that part marked up by the markup rate for that part”. This single chunk is a simple statement of fact: it is *not* in an if-then form. Under a reasonable understanding of the meaning of chunk [K1] it admits three different if-then interpretations:

$$\begin{aligned} \text{part/sale-price}(x, y) \leftarrow \text{part/cost-price}(x, z), \\ \text{part/mark-up}(x, w), y = (z \times w) \end{aligned} \quad \text{[C1.1]}$$

$$\begin{aligned} \text{part/cost-price}(x, z) \leftarrow \text{part/sale-price}(x, y), \\ \text{part/mark-up}(x, w), y = (z \times w) \end{aligned} \quad \text{[C1.2]}$$

$$\begin{aligned} \text{part/mark-up}(x, w) \leftarrow \text{part/sale-price}(x, y), \\ \text{part/cost-price}(x, z), y = (z \times w) \end{aligned} \quad \text{[C1.3]}$$

For the third of these if-then interpretations—with “part/mark-up” as its head—there is a possibility of round-off error. The three clauses [C1.1]—[C1.3] are a rather inconvenient representation of the single chunk [K1] because *one* statement of fact has been represented as *three* logical statements. Consider another chunk of knowledge: [K2] “The profit on a part is the difference between the sale price of that part and the cost price of that part.” As for [K1], the chunk [K2] is not in an if-then form. Under a reasonable understanding of its meaning, it also admits three if-then interpretations:

$$\begin{aligned} \text{part/profit}(x, y) \leftarrow & \text{part/sale-price}(x, z), \\ & \text{part/cost-price}(x, w), y = z - w \end{aligned} \quad [\text{C2.1}]$$

$$\begin{aligned} \text{part/sale-price}(x, z) \leftarrow & \text{part/profit}(x, y), \\ & \text{part/cost-price}(x, w), y = z - w \end{aligned} \quad [\text{C2.2}]$$

$$\begin{aligned} \text{part/cost-price}(x, w) \leftarrow & \text{part/sale-price}(x, z), \\ & \text{part/profit}(x, y), y = z - w \end{aligned} \quad [\text{C2.3}]$$

The six Horn clauses [C1.1]—[C2.3] may be combined using resolution to give additional potentially useful clauses:

$$\begin{aligned} \text{part/profit}(x, y) \leftarrow & \text{part/cost-price}(x, w), \\ & \text{part/mark-up}(x, u), z = (w \times u), y = z - w \end{aligned} \quad [\text{C3.1}]$$

$$\begin{aligned} \text{part/profit}(x, y) \leftarrow & \text{part/sale-price}(x, z), \\ & \text{part/mark-up}(x, u), z = (w \times u), y = z - w \end{aligned} \quad [\text{C3.2}]$$

$$\begin{aligned} \text{part/mark-up}(x, w) \leftarrow & \text{part/sale-price}(x, y), \\ & \text{part/profit}(x, u), u = y - z, y = (z \times w) \end{aligned} \quad [\text{C3.3}]$$

as well as some rather useless clauses such as:

$$\begin{aligned} \text{part/sale-price}(x, y) \leftarrow & \text{part/sale-price}(x, v), \text{part/profit}(x, u), u = v - z, \\ & \text{part/mark-up}(x, w), y = (z \times w) \end{aligned} \quad [\text{C3.4}]$$

A danger with all of [C3.1]—[C3.4] is that they are assembled from particular if-then interpretations of two chunks of knowledge, namely [K1] and [K2]. If the meaning of either of those two chunks should change then [C3.1]—[C3.4] may all have to be changed as well. This is not a problem if the relationships between [C3.1]—[C3.4] and both [K1] and [K2] are represented. But in the declarative representation, it is not clear that the meanings of [K1] and [K2] are buried in the four clauses [C3.1]—[C3.4]. So [C3.1]—[C3.4] and any other clauses derived from the original six clauses above, are potential maintenance hazards [10]. Given the six original clauses [C1.1]—[C2.3] there is no reason to combine them as illustrated in [C3.1]—[C3.4]. But clauses such as [C3.1]—[C3.4] are valid and could have been constructed by a knowledge engineer as part of a knowledge base. If they form part of a knowledge-based implementation then they may constitute a maintenance hazard [8].

Two problems with declarative formalisms have been identified. First, they have insufficient expressive power to represent all that a single chunk of knowledge has to say. Second there is no mechanism to prevent rules from being constructed whose validity relies on the validity of unrepresented sub-rules so leading to a potential maintenance hazard [9].

3. Items

Consider again the chunk of knowledge [K1]. Suppose that that chunk is represented as a thing—which we will call an *item*—with the name $[part/sale-price, part/cost-price, part/mark-up]$. The data associated with this item may be presented as a rather messy relation; such a relation is called the *value set* of the item, a possible value set is shown in Fig. 1. The meaning of an item A —called its *semantics* S_A —

<i>part/sale-price</i>		<i>part/cost-price</i>		<i>part/mark-up</i>	
1234	1.44	1234	1.20	1234	1.2
2468	2.99	2468	2.30	2468	1.3
3579	4.14	3579	3.45	3579	1.2
1357	10.35	1357	4.50	1357	2.3
9753	12.06	9753	6.70	9753	1.8
8642	12.78	8642	8.52	8642	1.5
4321	5.67	4321	2.70	4321	2.1

Fig. 1. Value set for a knowledge item.

is an expression that recognises the members of that item's value set. For the item considered above the semantics could be:

$$\lambda uvwxyz \bullet [S_{part/sale-price}(u, v) \wedge S_{part/cost-price}(w, x) \\ \wedge S_{part/mark-up}(y, z) \wedge (u = w = y) \wedge (v = x \times z)] \bullet$$

where $S_{part/sale-price} = \lambda xy \bullet [S_{part}(x) \wedge S_{sale-price}(y) \wedge \text{sells-for}(x, y)] \bullet$
and where $S_{part} = \lambda x \bullet [\text{is-a}[x:P]] \bullet$ for some suitable domain P where:

$$\text{is-a}[x:P] \begin{cases} = \mathbf{T} & \text{if } x \text{ is in } P \\ = \mathbf{F} & \text{otherwise} \end{cases}$$

In this example the knowledge represented by the item is very simple. The components of this knowledge item are *part/sale-price*, *part/cost-price* and *part/mark-up*. These components each occur once in the value set shown in Fig. 1. The value sets of some knowledge items have multiple occurrences of some components to enable complex item semantics to be expressed. For example, consider the rule whose meaning is defined by the pair of recursive clauses:

$$P(w, x) \leftarrow P(y, z), P(u, v), Q(w, y, u), R(x, z, v), x > 5 \\ P(w, x) \leftarrow Q(w, x, x), x \leq 5$$

The semantics of any item that represents this rule will employ multiple occurrences of the component P. The value set of any such item will also contain multiple occurrences of the component P. The semantics of complex, recursive knowledge items are represented in this way. The semantics of *all* knowledge items is λ -calculus "recognising functions" for the items' values sets.

Items are a single formalism for describing data, information and knowledge. Items may be presented informally as i-schema; that is, as they are employed in practice. Items may also be presented formally in the λ -calculus. Items have three important properties: items have a uniform format no matter whether they represent data, information or knowledge things; items incorporate two powerful classes of constraints, and a single rule of 'decomposition' can be specified for all items. Item 'decomposition' is a process which can be applied to the conceptual model to make it 'maintainable' in a precise sense. In particular, item decomposition removes potential maintenance hazards of the type illustrated above.

<i>name</i>		item name	<i>part/cost-price</i>	
<i>name1</i>	<i>name2</i>	item components	<i>part</i>	<i>cost-price</i>
x	y	dummy variables	x	y
meaning of item		item semantics	costs(x,y)	
constraints on values		value constraints	x < 1 999 → y ≤ 300	
set constraints		set constraints	∇	
			-----	○

Fig. 2. i-schema

An *item* consists of: an item name, item components, item semantics, item value constraints, and item set constraints. The *item name* is usually written in italics, A . The *item components* is a set of the names of items which this item represents some association between. The *item semantics* is an expression which recognises the members of the value set of this item, S_A . The *item value constraints* is an expression which must be satisfied by all members of the value set of this item, V_A . The *item set constraints* are constraints on the structure of this item's value set, C_A . The i-schema format for items is shown in Fig. 2. If two items have the property that the semantics of one logically implies the semantics of the other then the first item is a *sub-item* of the second. If two items have the property that they are both sub-items of each other then those two items are said to be *equivalent*.

For example, in an application there are 'parts' represented by the *part* data item. Each 'part' is identified by a 'part-number'. The value constraint for *part* requires that part numbers should lie in the range (1 000, 9 999). The set constraint for *part* requires that there are less than 100 different part numbers. Further there are 'costs' represented by the *cost-price* data item. Each *part* is associated with a *cost-price*. This association is represented by the information item *part/cost-price*. That item is subject to the "value constraint" that parts whose part-number is less than 1 999 will be associated with a cost price of no more than \$300. That item is subject to the "set constraints" that every part must be in this association, and that each part is associated with a unique cost-price. The i-schema for the *part/cost-price* item is shown in Fig. 2; it contains two set constraints. The '∇' in the *part* component column specifies a *universal constraint*; denoted by "Uni" in the formal λ -calculus representation. This universal constraint requires that all members of the value set of *part* must be in the value set of *part/cost-price*. The horizontal line in the *part* column and the '○' in the *cost-price* column specifies a *candidate constraint*; denoted by "Can" in the formal λ -calculus representation. This candidate constraint requires that members of the value set of *part* functionally determine the members of the value set of *cost-price* in this item—in traditional database jargon, part is a potential key of the part/cost-price relation. In general, the horizontal lines identify a set of components whose value sets functionally determine the value set of the single component identified by '○'. Candidate constraints for a knowledge item identify potential if-then interpretations of that item.

Items are a universal formalism in that they can describe knowledge just as naturally as information (ie relations) or data. For example, consider the chunk of knowledge [K1]. [K1] is represented as the knowledge item [*part/sale-price*, *part/cost-price*, *part/mark-up*] whose i-schema is shown in Fig. 3. That i-schema has four set constraints. The last three of those set constraints identify the three if-then

<i>[part/sale-price, part/cost-price, part/mark-up]</i>		
<i>part/sale-price</i>	<i>part/cost-price</i>	<i>part/mark-up</i>
(w, x)	(w, y)	(w, z)
$x = y \times z$		
$\rightarrow x > y$		
\forall	\forall	
o	-----	
-----		o
-----	o	-----

<i>[part/profit, part/sale-price, part/cost-price]</i>		
<i>part/profit</i>	<i>part/sale-price</i>	<i>part/cost-price</i>
(y, v)	(y, x)	(y, z)
$(v = x - z)$		
$\rightarrow v > 0$		
\forall	\forall	
o	-----	
-----		o
-----	o	-----

Fig. 3. Knowledge items for [K1] and [K2]

interpretations represented above as the clauses [C1.1]—[C1.3]. So the item shown in Fig. 3 represents *all* that the chunk of knowledge [K1] says. Items appear to have resolved the first difficulty with rules identified above. Items are more than a convenient way of representing a set of rules; they can be manipulated, combined and decomposed. Computing with a collection of items amounts to computing with the set of rules represented within those items. Manipulating items present no greater level of computational complexity than rule-based formalisms.

The i-schema for the chunk of knowledge [K2] is shown in Fig. 3. Item join is an operation that may be applied to two items *A* and *B* that share a set of common components *E* to construct a third item called the *join* of *A* and *B* on *E*, and denoted by: $A \otimes_E B$. Item join is defined formally below [2]. The following is an example of the use of the join operator:

$$\begin{aligned}
 & [part/profit, part/sale-price, part/cost-price, part/mark-up] = \\
 & [part/sale-price, part/cost-price, part/mark-up] \otimes_{\{part/sale-price, part/cost-price\}} \\
 & \quad [part/profit, part/sale-price, part/cost-price]
 \end{aligned}$$

The i-schema for this item is shown in Fig. 4. The i-schema for a sub-item of this item is shown in Fig. 5. Each candidate constraint in an item identifies an if-then interpretation of that item. The if-then interpretation of the sub-item in Fig. 5 identified by the third row from the bottom of that i-schema is the rule [C3.1]. That i-schema also contains two other if-then interpretations—identified by the last two

<i>[part/profit, part/sale-price, part/cost-price, part/mark-up]</i>			
<i>part/profit</i>	<i>part/sale-price</i>	<i>part/cost-price</i>	<i>part/mark-up</i>
(y, v)	(y, x)	(y, z)	(y, w)
$(v = x - z) \wedge (x = z \times w)$			
$\rightarrow ((x > z) \wedge (v > 0))$			
\forall	\forall	\forall	
\circ	-----		
-----		\circ	
-----	\circ	-----	
	\circ	-----	
	-----		\circ
	-----	\circ	-----

Fig. 4. Join of items for [K1] and [K2] on {part/sale-price, part/cost-price}

<i>[part/profit, part/cost-price, part/mark-up]</i>		
<i>part/profit</i>	<i>part/cost-price</i>	<i>part/mark-up</i>
(y, v)	(y, z)	(y, w)
$(v = z \times (w - 1))$		
$\rightarrow v > 0$		
\forall	\forall	
\circ	-----	
-----		\circ
-----	\circ	-----

Fig. 5. Sub-item of item in Fig. 4

rows of that i-schema—that have not been expressed here [2]. Using the rule of composition \otimes , knowledge items, information items and data items may be joined with one another regardless of type [11]. For example, the knowledge item:

$$[cost-price, tax] [\lambda xy \bullet [x = y \times 0.05] \bullet, \lambda xy \bullet [x < y] \bullet, \\ (Uni(cost-price) \wedge Can(tax, \{cost-price\}))_{[cost-price, tax]}]$$

can be joined with the information item *part/cost-price* on the set {cost-price} to give the information item *part/cost-price/tax*. In other words:

$$[cost-price, tax] \otimes_{\{cost-price\}} part/cost-price = \\ part/cost-price/tax [\lambda xyz \bullet [costs(x, y) \wedge z = y \times 0.05] \bullet, \\ \lambda xyz \bullet [((1000 < x < 1999) \rightarrow (0 < y \leq 300)) \wedge (z < y)] \bullet, \\ (Uni(part) \wedge \\ Can(cost-price, \{part\}) \wedge Can(tax, \{cost-price\}))_{part/cost-price/tax}]$$

<i>name</i>		object name	<i>costs</i>	
type1	type2	argument type	D ¹	D ¹
x	y	dummy variables	x	y
meaning of object		object semantics	costs(x,y)	
constraints on values		object value constraints	x < 1999 → y ≤ 300	
set constraints		object set constraints	∇	
			-----	○

Fig. 6. o-schema format and the object 'costs'

In this way items may be joined together to form more complex items [12]. Alternatively, the \otimes operator may form the basis of a theory of decomposition in which each item may be replaced by a set of simpler items. An item I is *decomposable* into the set of items $D = \{I_1, I_2, \dots, I_n\}$ if:

- I_i has non-trivial semantics for all i ,
- $I = I_1 \otimes I_2 \otimes \dots \otimes I_n$, where
- each join is *monotonic*; that is, each term in this composition contributes at least one component to I .

If item I is decomposable then it will not necessarily have a unique decomposition. So items can be combined using the \otimes operator. The \otimes operator also forms the basis of a theory of decomposition that removes hidden relationships from the represented knowledge. So items overcome the second difficulty described above for rules.

Items are a universal formalism for knowledge representation but make it difficult to analyse the structure of the whole application. For example, two chunks of knowledge that share the same basic wisdom may be expressed in terms of quite different components; this could obscure their common wisdom [13]. To make the inherent structure of items clear 'objects' are introduced as item building operators [4].

4. Objects

The introduction of objects enables the structure of the conceptual model to be analysed. In [4] it is shown that objects support a formal structure which may be used to manage maintenance. Objects are item building operators which have four important properties: objects have a uniform format no matter whether they represent data, information or knowledge things; objects incorporate 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 'decomposition' can be specified for objects such that if a 'decomposed' set of object operators is applied to a 'decomposed' set of 'base' items then the resulting conceptual model will contain only 'decomposed' items and will thus be maintainable. As for items, objects may either be represented informally as "o-schema" or formally as λ -calculus expressions. Object names are written in bold italic script. For example, suppose that ***costs*** is an operator that generates the information item *part/cost-price* from its components *part* and *cost-price*; that is: $part/cost-price = costs(part, cost-price)$

Further, suppose that the object *mark-up-chunk* is an operator which generates the knowledge item $[part/sale-price, part/cost-price, part/mark-up]$ from its components; that is:

$$[part/sale-price, part/cost-price, part/mark-up] = \mathbf{mark-up-chunk}(part/sale-price, part/cost-price, part/mark-up)$$

In general, an n-adic object is an operator which maps n given items into another item for some value of n. Further, 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. The type is 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 usually 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. The o-schema format for objects is shown in Fig. 6. 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 Fig. 6. The o-schema for the *costs* object contains two set constraints. The ‘ \forall ’ symbol has the obvious extension of meaning to its use for items; likewise for the horizontal line and the ‘ \mathbf{O} ’ symbol. Data objects provide a representation of sub-typing.

For example, the *costs* object in its λ -calculus form is:

$$\begin{aligned} & \mathbf{costs}[\lambda P:\mathbf{X}^1 Q:\mathbf{X}^1 \cdot \lambda xy \cdot [S_P(x) \wedge S_Q(y) \wedge \mathbf{costs}(x, y)] \bullet \bullet, \\ & \lambda P:\mathbf{X}^1 Q:\mathbf{X}^1 \cdot \lambda xy \cdot [V_P(x) \wedge V_Q(y) \wedge ((1\ 000 < x < 1\ 999) \rightarrow (y \leq 300))] \bullet \bullet, \\ & \lambda P:\mathbf{X}^1 Q:\mathbf{X}^1 \cdot [C_P \wedge C_Q \wedge (\text{Uni}(P) \wedge \text{Can}(Q, \{P\})) \forall_{(\mathbf{costs}, P, Q)}] \bullet] \end{aligned}$$

and the λ -calculus for the *mark-up-chunk* object is:

$$\begin{aligned} & \mathbf{mark-up-chunk}[\lambda P:\mathbf{X}^2 Q:\mathbf{X}^2 R:\mathbf{X}^2 \cdot \lambda x_1 x_2 y_1 y_2 z_1 z_2 \cdot [S_P(x_1, x_2) \wedge S_Q(y_1, y_2) \\ & \wedge S_R(z_1, z_2) \wedge ((x_1 = y_1 = z_1) \rightarrow (x_2 = z_2 \times y_2))] \bullet \bullet, \\ & \lambda P:\mathbf{X}^2 Q:\mathbf{X}^2 R:\mathbf{X}^2 \cdot \lambda x_1 x_2 y_1 y_2 z_1 z_2 \cdot [V_P(x_1, x_2) \wedge V_Q(y_1, y_2) \wedge V_R(z) \wedge \\ & ((x_1 = y_1 = z_1) \rightarrow (x_2 > y_2))] \bullet \bullet, \\ & \lambda P:\mathbf{X}^2 Q:\mathbf{X}^2 R:\mathbf{X}^2 \cdot [C_P \wedge C_Q \wedge C_R \wedge (\text{Uni}(P) \wedge \text{Uni}(Q) \wedge \text{Can}(P, \{Q, R\}) \\ & \wedge \text{Can}(Q, \{P, R\}) \wedge \text{Can}(R, \{P, Q\})) \forall_{(\mathbf{mark-up-chunk}, P, Q, R)}] \bullet] \end{aligned}$$

The **mark-up-chunk** knowledge object represents the essence of the knowledge within the knowledge chunk [K1] without any reference to the components of that chunk. Objects represent abstract knowledge. Objects also represent value constraints and set constraints in a uniform way. A “join” operation for objects is defined in a similar way [4] to the join operation for items described above. If a set of objects have been decomposed using the object join operator then items generated by those objects will also be in a decomposed form.

Data objects provide a representation of sub-typing. Knowledge is quite clumsy when represented as items; objects are a more compact representation. For example, consider the [*part/sale-price*, *part/cost-price*, *part/mark-up*] knowledge item which represents the chunk of knowledge [K1]. This item can be built by applying a knowledge object **mark-up-chunk** of argument type (I^2 , I^2 , I^2) to the items *part/sale-price*, *part/cost-price* and *part/mark-up*.

References

1. Barr, V. (1999). “Applying Reliability Engineering to Expert Systems” in *proceedings 12th International FLAIRS Conference*, Florida, May 1999, pp494-498.
2. Debenham, J.K. “*Knowledge Engineering*”, Springer-Verlag, 1998.
3. Debenham, J.K. “The Degradation of Knowledge Base Integrity”, in *proceedings 13th International FLAIRS Conference FLAIRS-2000*, Orlando, Florida, May 2000, pp113-117.
4. Debenham, J.K. (1999). “Knowledge Object Decomposition” in *proceedings 12th International FLAIRS Conference*, Florida, May 1999, pp203-207.
5. Debenham, J.K. “Representing Knowledge Normalisation”, in *proceedings Tenth International Conference on Software Engineering and Knowledge Engineering SEKE’98*, San Francisco, US, June 1998.
6. Date, C.J., “*An Introduction to Database Systems*” (4th edition) Addison-Wesley, 1986.
7. Mayol, E. and Teniente, E. (1999). “Addressing Efficiency Issues During the Process of Integrity Maintenance” in *proceedings Tenth International Conference DEXA99*, Florence, September 1999, pp270-281.
8. Ramirez, J. and de Antonio, A. (2000). “Semantic Verification of Rule-Based Systems with Arithmetic Constraints” in *proceedings 11th International Conference DEXA2000*, London, September 2000, pp437-446.
9. 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.
10. Johnson, G. and Santos, E. (2000). “Generalizing Knowledge Representation Rules for Acquiring and Validating Uncertain Knowledge” in *proceedings 13th International FLAIRS Conference*, Florida, May 2000, pp186-2191.
11. Werner, E. “Logical Foundations of Distributed Artificial Intelligence” in O’Hare, G.M.P. & Jennings, N.R. (Eds) “*Foundations of Distributed Artificial Intelligence*”, pp57-118, Wiley, 1996.
12. Darwiche, A. (1999). “Compiling Knowledge into Decomposable Negation Normal Form” in *proceedings International Joint Conference on Artificial Intelligence, IJCAI’99*, Stockholm, Sweden, August 1999, pp 284-289.
13. Jantke, K.P. and Herrmann, J. (1999). “Lattices of Knowledge in Intelligent Systems Validation” in *proceedings 12th International FLAIRS Conference*, Florida, May 1999, pp499-505.