Quantitative Assessment of the Significance of Inconsistencies in Object-Oriented Designs

George Spanoudakis, Hyoseob Kim

Department of Computing, City University

Northampton Square, London, EC1V 0HB

Tel: + 44 20 74778413, Fax: + 44 20 74778587

E-mail: {gespan | hkim69} @soi.city.ac.uk

Abstract: This paper presents a framework for assessing the significance of inconsistencies which arise in object-oriented design models that describe systems from multiple perspectives. The framework allows the definition of significance criteria and measures the significance of inconsistencies as beliefs for the satisfiability of these criteria.

1. Introduction

The need to describe complex software systems from different design perspectives, such as those of the static structure and the interactions of the components of a system, may result in the construction of many partial system design models (or simply "models" henceforth). These models may be constructed independently by different designers, may advocate specific modelling angles and may reflect disparate perceptions of these designers. As a result, they may be inconsistent with each other.

Inconsistencies occur when partial models refer to common aspects of the system under development and make assertions which violate consistency rules applicable to these aspects [4,11]. As an example consider an object-oriented design model that consists of an object interaction diagram and a class diagram. Assume also a consistency rule requiring that for any message received by an object in the interaction diagram, an operation with the same signature as the message must have been defined for one of the classes of the object in the class diagram. In this model an inconsistency would arise if there was a message with no counterpart operation, thus

violating the above consistency rule.

Inconsistencies are inevitable in software development [13]. And, although they will have to be settled eventually, they may need to be tolerated temporarily to give designers a chance to work independently developing their own parts of a model without the need for continual reconciliation [4,11]. In settings providing freedom for groupwork, it is important to be able to *diagnose* the significance of an inconsistency in order to decide when and with what degree of priority it has to be settled [4,11]. In one of the experiments reported in [10], we detected 278 violations of the consistency rule mentioned above. In such cases having a mechanism to assess the significance of inconsistencies and order them by this significance would be undoubtedly useful. This paper introduces a framework that we have developed to support this assessment.

The main premise of our framework is that the significance of an inconsistency depends on the significance of the model elements that give rise to it for the model. Our framework assumes models expressed in the Unified Modelling Language (UML [7]) and defines a set of *characteristics* which indicate the significance of the main kinds of elements in such models.

The assessment of whether or not an element has a particular characteristic in a model is approximate; the framework incorporates *belief functions* measuring the extent to which it may be believed from its modelling that an element has the characteristic. The need for approximate reasoning arises because it cannot always be guaranteed that the model provides a consistent, complete and accurate description of the system it describes at the different stages of its evolution. Also it cannot be guaranteed that the element will retain the characteristic in the next version of the model.

In the rest of this paper, we introduce the characteristics which indicate the significance of model elements and the belief functions associated with them (§2), establish a scheme for expressing consistency rules and significance criteria which determine the characteristics that the elements violating them must have for the violations to be significant (§3), give an example of how to

use these criteria to evaluate the significance of inconsistencies and rank them (§4), overview related work (§5), and conclude with a summary and directions for further work (§6).

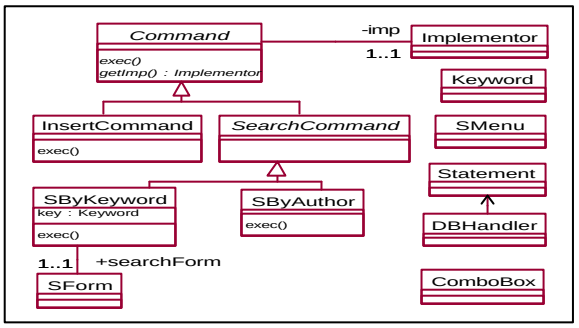
2. Characteristics of significant model elements

The UML models assumed by our framework can be composed of any number of *class* and *sequence diagrams*. Class diagrams specify the static structure of, and the relationships between the classes of a system. Classes can have *attributes*, *operations*, and be related by *associations* and *generalisation* (*Is-a*) relations. Sequence diagrams specify *interactions* between the instances of these classes (the terms "sequence diagram" and "interaction" are used synonymously in the rest of the paper). An interaction consists of a set of messages exchanged between objects to deliver part of the functionality of a system. A complete description of the semantics of these kinds of UML model elements is beyond the scope of this paper and may be found in [7].

In our framework, the significance of the above kinds of UML model elements is indicated by six characteristics: the *genericity* and *coordination capacity* of classes, the *functional essentiality* of attributes and association ends, the *characticity* of operations, and the *functional dominance* and *coordinating capacity* of messages. These characteristics are described below.

2.1 Class genericity

In software models, classes with numerous



subclasses normally specify interfaces (i.e. sets of operation signatures) for groups of services which are provided by their subclasses and the internal state of the instances of these subclasses which is required to realise the services. In effect, such generic classes provide a basis for specifying clients capable of using the services without knowing the exact class which provides them. An inconsistency involving the specification of a generic class is significant since it may

affect both its subclasses and the clients of the services specified by it.

The belief to the genericity of a class in our framework is measured as the likelihood of an arbitrary class on the longest generalisation path that involves it in a model being a subclass of it:

Definition 1: The belief to the genericity of a class *c* in a model M (denoted by the predicate *gen-c(c)*) is defined as:

$m_1(\text{gen-c(c)}) = L_{sub}/(L_{sub}+L_{sup})$ *if c.Sub* ¹ Æ*

$m_1(\text{gen-c(c)}) = 0$ *otherwise*

$m_1(\neg\text{gen-c(c)}) = 1 - m_1(\text{gen-c(c)})$

where

- L_{sub} is the length of the longest path of Is-a relations in *M* ending at *c*
- L_{sup} is the length of the longest path of Is-a relations in *M* starting from *c*
- *c.Sub** is transitive closure of the subclasses of *c*

Figure 1: UML class diagram for a library system

Given this definition, the beliefs to the genericity of the command classes (in the sense of [3]) *Command, SearchCommand* and *SByKeyword* shown in the class diagram of Figure 1 (class diagram of a library system, cf. Section 2.2 below) are 1, 0.5 and 0, respectively.

The reason why we are considering only the longest generalisation path involving a class in a model is that this path gives the most accurate indication of the number of the successive layers at which the class may be specialised or generalised [8].

2.2 Coordination capacity of classes

Some classes in the design of a system may have a coordination capacity, that is they may exist to coordinate interactions between other classes. Coordinating classes are very important in a design since they encapsulate protocols of interactions between the classes they coordinate and, thus, they appear in numerous design patterns (e.g. *mediator*, *observer*, *facade* [3]). An inconsistency involving a coordinating class is important since it is likely to affect all the classes and the interactions it coordinates.

A common characteristic of coordinating classes across all the different coordination patterns they may realise is that they send

messages to or receive messages from all the classes that they coordinate. Drawing upon this observation, we measure the belief to the coordination capacity of a class *c* in a set of interactions *S* as the likelihood of an arbitrary class in *S* be communicating with it:

Definition 2: The belief to the coordination capacity of a class *c* in a subset *S* of the interactions of a model (denoted by the predicate *coord-c(c,S)*) is defined as:

$m_2(\text{coord-c}(c,S)) = |\text{Com}(c,S)|/|\text{Classes}(S) - \{c\}|$

$m_2(\neg\text{coord-c}(c,S)) = 1 - m_2(\text{coord-c}(c, S))$

where

- *Com(c,S)* is the set of the classes whose instances send messages to or receive messages from the instances of c in the interactions of the set *S* excluding *c*
- *Classes(S)* is the set of the classes which appear as receivers or senders of messages in the interactions of *S*.

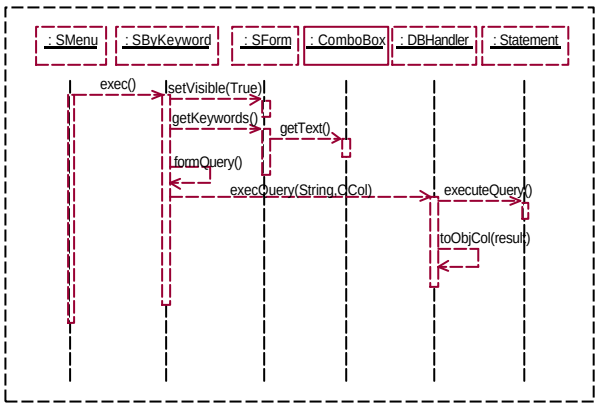


Figure 2: I_1 - Sequence diagram of a library system

The sequence diagram of Figure 2 shows an interaction between the classes of a library system. The interaction takes place when the system is used to search for library items by keywords. As shown in the diagram, a search menu (*SMenu*) is used to activate the option of searching for library items by keywords. This option is modelled by the command class *SByKeyword*. When activated to execute the operation *exec()*, *SByKeyword* displays a search form (*setVisible(True)*), gets some keywords (*getKeyword()*), constructs a string representing an SQL query (*formQuery()*) and invokes the operation *execQuery(String,OCol)* in the class *DBHandler* (i.e., a database driver) to execute this query.

According to Definition 2, the beliefs to the coordination capacity of the classes *SByKeyword*, *DBHandler*, *SMenu* in the diagram are 0.6, 0.4 and 0.2, respectively. These beliefs reflect the strong coordination capacity of *SByKeyword* in the entire interaction, the less strong coordination capacity of *DBHandler* for a part of the interaction and the almost negligible coordination capacity of *SMenu*.

2.3 Functional essentiality of attributes and association ends

Attributes and association ends may provide the only channels for sending messages between the instances of the classes connected to them. Consider, for instance, an interaction where an instance of a class c_i sends a message to an instance of another class c_j. Unless c_i has an attribute or an association end whose type is the class c_j (and therefore its instances have a means of holding references to the instance of c_j) or the message has an argument of type c_j, the instance of c_i will not be able to identify and send the message to the instance of c_j.

Note also that in cases where c_i has more than one attributes or navigable association ends of type c_j it is impossible to identify from the model which of these attributes or association ends is used by the sender of the message[1]. Nevertheless, it is plausible to assume that the more the messages sent by the instances of c_i (or its subclasses) to instances of the type of an attribute or association end *a* and the fewer the other attributes or association ends of c_i having the same type as *a*, the higher the chance that at least one of these messages is dispatched through *a* and thus the higher the functional essentiality of *a* for the class c_i. Drawing upon this observation, we define the belief to the functional essentiality of attributes and association end as follows:

Definition 3: The belief to the functional essentiality of an attribute or association end *a* for a class *c* in a model *M* (denoted by the predicate *fessen-a(a,c)*) is defined as:

$m_3(\text{fessen-a}(a,c)) =$

$1 - (1-1/(|\text{Rel}(a,c)| +1))^{|\text{Mes}(a, c, M)|}$

$m_3(\neg\text{fessen-a}(a,c)) = 1 - m_3(\text{fessen-a}(a,c))$

where

- *Mes(a,c,M)* is the set of messages sent by the instances of *c* (or its subclasses) to instances of the type of the attribute or the association end *a* which do not have an argument of the same type as *a*
- *Rel(a,c)* is the set of the attributes and navigable association ends defined in or inherited by the class *c* that have the same type as *a*

[1] The graphical syntax of UML for sequence diagrams does not allow the specification of the exact attribute or association end whose value is used as the receiver of a message in an interaction.

m_3 measures the likelihood of the instances of *c* sending messages to objects that constitute the value of the attribute or association end *a*. In Definition 3, the cardinality of *Rel(a,c)* is increased by one to account for the possibility of sending the message to an instance of *c* that is created within the method that implements the operation invoked by the message. This is necessary since this creation might not be evident from the interaction itself.

According to Definition 3, the beliefs to the functional essentiality of the association end *searchForm* and the attribute *keyword* for the class *SForm* in Figure 1 – given the sequence diagram of Figure 2 – are 0.75 and 0, respectively. These beliefs reflect the fact that *searchForm* is likely to be the association end used to identify the receivers of at least one of the messages in the diagram sent to instances of *SForm*. Unlike it, the attribute *keyword* does not appear to have any functional role for *SForm* since no messages are sent to instances of its type (that is the class *Keyword*).

An inconsistency involving a functionally essential attribute or association end is significant because it may affect the ability of the objects to request the execution of operations.

2.4 Operation characticity

An operation overridden by most of the classes in its scope, that is the set of the classes which introduce or inherit it in a model, is significant for the design of a system because it constitutes a basic kind of behaviour which must be provided by objects of different types (even if realised in different ways by these objects). We refer to this characteristic of operations as "operation characticity" and define the belief to it as follows:

Definition 4: The belief to the characticity of an operation *o* in a model *M* (denoted by the predicate *char-o(o)*) is defined as

$m_4(\text{char-o(o)}) =$

$\Pi_{c \,\varepsilon\, Oclasses(o)} |Ov(o,c) \cup \{c\}| \,/\, |\, c.Sub^* \cup \{c\}|$

$m_4(\neg \text{char-o(o)}) = 1 - m_4(\text{char-o(o)})$

where

- *Oclasses(o)* is the set of the most general superclasses of the class of o which define an operation with the same signature as o

- *Ov(o,c)* is the set of the subclasses of *c* which override *o*

m_4 measures the likelihood of an arbitrary class in each of the possible scopes of an operation overriding it.

According to Definition 4, the beliefs to the characticity of the operations *exec()* and *getImp()* in the class diagram of Figure 1 are 0.8 and 0.2, respectively. The former belief measure reflects the fact that *exec()* is an operation that has to be defined in every command class (since it is used to trigger the execution of these commands [4]) but implemented differently by each of these command classes. Unlike it, the operation *getImp()*, which returns the object that implements a command, has a single implementation in the abstract command class *Command*. The fact that *getImp()* is not overridden by any of the different command classes in the Is-a hierarchy of Figure 1 indicates the relatively insignificant functional role of it for these classes.

2.5 Coordination capacity of messages

Messages in interactions are exchanged between objects to invoke operations in these objects. These operations may: (a) provide part of the internal functionality of the object, or (b) coordinate the interaction of a group of other objects by invoking other operations in them, combining the data that the latter operations may generate, and eventually notifying the combined outcome of the interaction to the object that invoked them.

The operations of the latter kind (and therefore the messages invoking them) are more critical for the design of the system than those of the former kind. This is because they realise the protocols of the required coordinations between objects. Note, however, that in a UML design model, the only evidence about the operations invoked when a specific operation is executed comes from the messages dispatched by the message that invokes the operation. Also, depending on the elaboration stage of a model, the messages which appear in sequence diagrams may not have counterpart operations defined for the classes of their receivers (or their superclasses) in the class diagrams. To cope with these phenomena, we have defined the

coordination capacity as a characteristic of messages:

Definition 5: The belief to the coordination capacity of a message *m* in a subset *S* of the interactions of a model *M* (denoted by the predicate *coord-m(m,S)*) is defined as:

m_5(coord-m(m,S)) = |Dsig(m,S)|/ |Asig(m,S)|

if Asig(m, S) ¹ Æ

m_5(coord-m(m, S)) = 0 *if Asig(m, S) = Æ*

m_5(¬coord-m(m,S)) = 1 – m_5(coord-m(m,S))

where

- *Dsig(m, S)* is the set of the signatures of the messages directly dispatched by *m* in the interactions of *S*
- *Asig(m,S)* is the set of the signatures of the messages which are directly or indirectly dispatched by *m* in the interactions of *S*

m_5 measures the likelihood of an arbitrary message *x* in the transitive closure of the messages dispatched by a message *m* being directly (as opposed to indirectly) dispatched by *m*.

According to Definition 5, the beliefs to the coordination capacity of the messages *execQuery(String,OCol)*, *exec()*,and *setVisible(True)* in Figure 2 are 1, 0.57 and 0, respectively. These beliefs indicate that *execQuery(String,OCol)* has a co-ordination capacity in the part of the interaction which deals with the retrieval of data from the database of the library system, *exec()* has some co-ordination capacity for the entire interaction, and *setVisible(True)* has no coordination capacity.

2.6 Functional dominance of messages

We consider messages that invoke operations triggering a substantial part of the behaviour of objects in an interaction being functionally dominant in it. In our framework, the basic belief to the functional dominance of a message *m* in an interaction is defined as the likelihood of an arbitrary message in it being dispatched by *m* as shown below:

Definition 6: The belief to the functional dominance of a message *m* in an interaction *I* of a model *M* (denoted by the predicate *fdom-m(m,S)*) is defined as:

m_6(fdom-m(m,I))=(|Asig(m,{I})|+1)/|Sg(I,m)|

m_6(¬fdom-m(m,I)) = 1 – m_6(fdom-m(m,I))

where *Sg(I,m)* is the set of the signatures of

the messages in I excluding the signature of m.

According to Definition 6, the beliefs to the functional dominance of the messages *exec()* and *execQuery(String,OCol)* are 1 and 0.28, respectively. These belief measures reflect the fact that the former message triggers the entire interaction while the latter message triggers only a small part of it.

3. Specification of consistency rules and significance criteria

As we discussed in Section 1, we define an inconsistency as a violation of a specific consistency rule. To assess the significance of inconsistencies, our framework introduces a scheme for specifying significance criteria and associating them with consistency rules. These criteria define the characteristics that the elements involved in the violation of a rule should have for the violation to be significant.

We express consistency rules using the Object Constraint Language (OCL is defined as part of [7]) and significance criteria using a subset of OCL and the predicates introduced in Section 2, and wrap them in UML objects related as indicated in the extension of the UML meta-model that we have made and is shown in Figure 3.

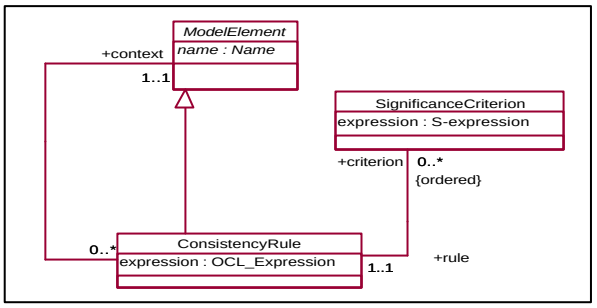


Figure 3: Consistency rules and significance criteria

As shown in Figure 3, each consistency rule is associated with a specific UML model element, called the "context" of the rule. Consequently, the OCL expression that specifies the rule can make references to all the named structural and behavioural features of its context as well as to the associations and generalisations which may relate it to other model elements. The classes of a UML model along with built-in OCL types which represent primitive data types and collections of values/objects (for example *Set* [7]) are the legitimate types for the OCL expressions written for it.

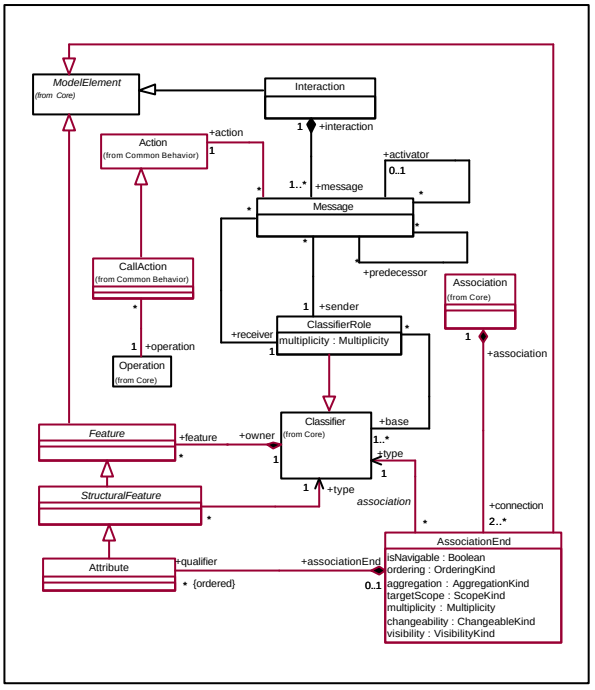


Figure 4: UML model elements

(adopted from [7])

An OCL expression specifies conditions over the values of the features it references using the standard logical operators "and", "or", "implies" and "not" and the set operators "forall" and "exists". The semantics of these set operators are the same as the semantics of the universal and existential quantifier of predicate calculus. Thus, an expression of the form *set->forall(x | OCL-condition-over-x)* and *set->exists(x | OCL-condition-over-x)* becomes true if *OCL-condition-over-x* is true for all or at least one of the elements of *set*, respectively.

As an example of specifying consistency rules using OCL consider a rule requiring that for every message in a sequence diagram there must be either an association or an attribute between its sender and its receiver navigable from the former to the latter class. This rule can be defined in the context of the UML meta-class *Message* (i.e., the class of all the messages which appear in the interactions of a model, see Figure 4) using OCL as follows[2]:

Rule-1

context: Message

expression:

self.*action*.**oclIsTypeOf**(*CallAction*)

implies self.*sender*.*feature*->**exists**(a |

a.**oclIsTypeOf**(*Attribute*) **and**

(a.*type* = **self**.*receiver*) **or**

Association.**allInstances**–>**exists**(r |

r.*connection*->**exists**(e_1, e_2 | (e_1 <> e_2) **and**

(e_1.*type* = **self**.*sender*) **and** (e_2.*type* =

self.*receiver*) **and** (e_2.*isNavigable* = **True**)))

A significance criterion in our framework is specified by a significance expression (*S*-

[2] In OCL and S-expressions strings in boldface and Italics are reserved OCL keywords and names established in the UML meta-model, respectively. **self** in these expressions refers to an instance of the class that constitutes the context of the consistency rule and consequently the context of the S-expression that defines a criterion associated with it.

expression) and must be associated with a consistency rule (see Figure 3). The S-expression specifies a logical combination of the characteristics which the model elements giving rise to the violation of the rule (or other model elements connected to them) are required to have for the inconsistency to be significant. These characteristics are specified by using the special predicates defined in Section 2. An S-expression has the same context as the consistency rule associated with the criterion it defines and, therefore, it can reference any named feature in the closure of the features of the model elements which are reachable from this context.

Tables 1 and 2 present the syntactic forms of the S-expressions definable in our framework and the typing conditions that these expressions have to satisfy in order to be valid. More specifically, Table 1 presents the syntactic forms of, and the type validity conditions for the so-called "atomic S-expressions" (these are expressions consisting of only one of the predicates introduced in Section 2). The type validity condition determines the valid type(s) for the element(s) that the predicate of an expression refers to. Table 2 presents the syntactic forms of, and the validity conditions for "non atomic S-expressions" (these are logical combinations of atomic S-expressions). Thus, for instance, according to Table 1 the S-expression *gen-c(elem)* is valid only if the type of the model element denoted by *elem* is the UML meta-class *Class*. The complete grammar for S-expressions is given in [10].

Atomic S-expression	**Belief**	**Type validity condition**
gen-c(elem)	Bel(gen-c(elem)) = m_1(gen-c(elem))	elem.*type* = *Class*
fessen-a(elem$_1$,elem$_2$)	Bel(fessen-a(elem$_1$,elem$_2$)) = m_3(fessen-a(elem$_1$,elem$_2$))	elem$_1$.*type* = *Attribute* OR elem$_1$.*type* = *AssociationEnd* AND elem$_2$.*type* = *Class*
char-o(elem)	Bel(char-o(elem)) = m_4(char-o(elem))	elem.*type* = *Operation*
coord-c(elem$_1$,elem$_2$)	Bel(coord-c(elem$_1$,elem$_2$)) = m_2(coord-c(elem$_1$,elem$_2$))	elem$_1$.*type* = *Class* AND elem$_2$.*type* = **Set** (*Interaction)*
coord-m(elem$_1$,elem$_2$)	Bel(coord-m(elem$_1$,elem$_2$)) = m_5(coord-m(elem$_1$,elem$_2$))	elem$_1$.*type* = *Message* AND elem$_2$.*type* = **Set** (*Interaction)*
fdom-m(elem$_1$,elem$_2$)	Bel(fdom-m(elem$_1$, elem$_2$)) = m_6(fdom-m(elem$_1$, elem$_2$))	elem$_1$.*type* = *Message* AND elem$_2$.*type* = **Set** (*Interaction)*

Table 1: Syntactic forms, typing conditions and beliefs for valid atomic S-Expressions

Non atomic S-Expression		**Belief**	**Validity condition**
Non quantified expressions	p_1 **and** …**and** p_n	$Bel(and_{i=1,\ldots,n}\ p_i) = \Pi_{\ i=1,\ldots,n} Bel\ (p_i)$	p_i : valid atomic S-expression (forall i=1,…,n)
	p_1 **or** …**or** p_n	$Bel(or_{i=1,\ldots,n}\ p_i) = \Sigma_{J\subseteq\{1,\ldots,n\}}(-1)^{\lvert J\rvert+1} Bel(and_{i\varepsilon J} p_i)$	p_i : valid atomic S-expression (forall i=1,…,n)
Quantified expressions	elem->**exists**(x | OCL-exp-over-x **and** se(x))	$\Sigma_{J\subseteq S}(-1)^{\lvert J\rvert+1} Bel(and_{x\varepsilon J}\ se(x))$ where S = {x | (x ε elem) and OCL-exp-over-x = True}	elem.*type* = **Set**(*ModelElement*) AND se(x): is a valid non quantified S-expression over x
	elem->**forall**(x | OCL-exp-over-x **and** se(x))	$\Pi_{x\ \varepsilon\ elem} Bel(se(x))$ *If elem->**forall**(x | OCL-exp-over-x) = True)* 0, *Otherwise*	elem.*type* = **Set**(*ModelElement*) AND se(x): is a valid non quantified S-expression over x

Table 2: Syntactic forms of and validity conditions for non atomic S-Expressions

As an example of specifying a significance criterion consider the case where the violations of *Rule-1* above should be considered significant only if they are caused by messages which are functionally dominant and have coordinating capacity in their interactions. This criterion of significance can be specified as follows:

Criterion-1

Rule: Rule-1

S-expression:

fdom-m(**self**, **self.***interaction*)

and coord-m(**self**, **self.***interaction*)

In the S-expression of this criterion, "self" refers to the instances of the context of *Rule-1*, that is the UML meta-class *Message*. By using the special predicates *fdom-m* and *coord-m*, this S-expression specifies that the message that violates the rule must be functionally dominant and have a coordinating capacity in the interaction (sequence diagram) it belongs to (that is the value of the feature: **self.***interaction*).

To assess the significance of the violations of a specific consistency rule, we compute degrees of belief for the satisfiability of the S-expression of the criterion associated with the rule by the elements of the model which this expression refers to. These elements are related to the model elements that gave rise to

the violation of the rule as specified by the S-expression. Subsequently, the violations the rule are ranked in descending order of these degrees of belief.

Tables 1 and 2 show the formulas used to compute the degrees of belief for the different forms of atomic and non-atomic S-expressions. These formulas are derived using the axioms of the Dempster-Shafer theory of evidence [9] as we prove in [10]. Their derivation is based on the fact that – as we have also proven in [10] – the belief functions introduced in Section 2 satisfy the axiomatic foundation of Dempster-Shafer *basic probability assignments* [9].

In the following section, we give an example of computing degrees of belief for the satisfiability of significance criteria and ranking inconsistencies according to them.

4. Example

As an example of detecting and assessing the significance of inconsistencies in our framework, consider the UML model consisting of the class and sequence diagrams, shown in Figures 1 and 2. These diagrams are inconsistent with respect to *Rule-1* in Section 3 since there are no attributes and/or associations between the sender and the receiver of the following messages: *exec()*, *getText()*, *execQuery(String,OCol)*.

If the significance of these inconsistencies is assessed according to *Criterion-1* in Section 3, the inconsistency caused by the message *exec()* becomes the one with the highest significance, followed by the inconsistencies caused by the messages *execQuery(String,OCol)*, and *getText()*. This is because the degrees of belief about the satisfiability of *Criterion-1* by each of these messages are (according to the belief functions of Tables 1 and 2):

1) $Bel(\text{fdom-m}(\text{exec}(),I_1)$ **and**
   $\text{coord-m}(\text{exec}(),\{I_1\})) =$
   $m_6(\text{fdom-m}(\text{exec}(),I1)) \times$
   $m_5(\text{coord-m}(\text{exec}(),\{I_1\})) = 1 \times 0.57 = 0.57$
2) $Bel(\text{fdom-m}(\text{execQuery}(\text{String},\text{OCol}),I_1)$
   and
   $\text{coord-m}(\text{execQuery}(\text{String},\text{OCol}),\{I_1\})) =$
   $m_6(\text{fdom-m}(\text{execQuery}(\text{String},\text{OCol}),I1)) \times$
   $m_5(\text{coord-m}(\text{execQuery}(\text{String},\text{OCol}),\{I_1\})) =$

$0.286 \times 1 = 0.286$

3) Bel(fdom-m(getText(),I_1) **and** coord-m(getText(),{I_1})) = m_6(fdom-m(getText(),I1)) × m_5(coord-m(getText(),{I_1})) = $0 \times 0 = 0$

5. Related work

A substantial body of research has been concerned with the problem of detecting and resolving inconsistencies between software system specifications [2,4,11,13,14,15,16] but only two strands of work [2,4] have been concerned with the diagnosis of inconsistencies. Emmerich et al [2] have developed a framework for managing the compliance of software documentation artefacts with consistency rules which realise document representation standards. In their framework, software designers can implement diagnostic checks to assess the importance and the difficulty of making a document compliant with the rule it violates. In [4], diagnosis has been realised as the identification of parts of formal specifications which are not affected by an inconsistency and, therefore, they are safe to reason from.

Software metrics similar to some of the metrics defined in our framework have been proposed in the literature but have not been used to assess the significance of inconsistencies in software models.

More specifically, the *depth of inheritance tree (DIT)* [1] and the *class hierarchy nesting level* [6] are similar to m_1. Note, however, that unlike m_1, DIT does not take into account the length of the longest path to the most specific subclass of the class of concern and therefore treats as generic classes which have no subclasses. Such classes are not as important as classes that m_1 would spot as generic since they have no subclasses that could also be affected by inconsistencies involving them. Also *class coupling (CBO)* [1] and the *number of collaborating classes (NCC)* [5] are similar to m_2. The difference between m_2 and CBO and NCC is that m_2 provides a relative measure of inter-class collaboration in a specific set of system interactions.

Also it has to be appreciated that, what clearly differentiates the metrics used in our framework from the above software metrics is their common underlying axiomatic interpretation as D–S beliefs. This, as

discussed in [10], provides a sound basis for deriving the beliefs for the significance criteria presented in Section 3.

6. Summary and future work

This paper presented a framework for assessing the significance of inconsistencies in design models of software systems expressed in UML based on criteria that software designers can specify to establish the characteristics that the model elements involved in an inconsistency should have for the inconsistency to be significant.

On going work focuses on the experimental evaluation of the framework, investigation of the possibility of expanding it with more characteristics of model elements. We are also extending the current implementation of the framework on the top of the CASE tool Rational Rose [12].

Acknowledgements

The work presented in this paper has been partially funded by the British Engineering and Physical Sciences Research Council (IMOOSD project, EPSRC grant No. GR/M57422).

References

1. Chidamber S., Kemerer C., 1994. A Metrics Suite for Object Oriented Design, IEEE Transactions on Software Engineering, 20(6), pp. 476-493.
2. Emmerich W., et al., 1999. Managing Standards Compliance. IEEE Transactions on Software Engineering 25(6).
3. Gamma E., et al., 1995. Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley.
4. Hunter A. and Nuseibeh B., 1998. Managing Inconsistent Specifications: Reasoning, Analysis and Action, ACM Transactions in Software Engineering and Methodology, 7(4), pp. 335-367
5. Jacobson I., et al., 1995. Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley.
6. Lorenz M., 1993. Object-Oriented Software Development: A Practical Guide, Prentice Hall.
7. OMG, 1999. OMG Unified Modelling Language Specification, V. 1.3a. Available from: ftp://ftp.omg.org/pub/docs/ad/99-06-08.pdf.
8. Rosch E. et al., 1976. Basic Objects in Natural Categories, Academic Press.
9. Shafer G., 1975 A Mathematical Theory of Evidence, Princeton University Press.
10. Spanoudakis G., Towards an Evidential Significance Diagnosis Framework for Elements of UML Software Models, Technical Report, Technical Report Series, City University, Department of Computing, 1999
11. Spanoudakis G., Finkelstein A. Managing Interference, Proc. of the SIGSOFT '96 workshops, ACM Publications, pp. 172-174
12. Rational Software Corporation, 1998.

Rational Rose '98: Extensibility Reference Manual. See also: http://www.rational.com/products/rose/index.jtmpl

13. Schwanke W., Kaiser E., Living with Inconsistency in Large Systems, Proc. of the Int. Workshop on Software Version and Configuration Control, pp. 98-118
14. van Lamsweerde A., Darimont A., Letier E., 1998. Managing Conflicts in Goal-Driven Requirements Engineering, IEEE Transactions on Software Engineering, Special Issue on Inconsistency Management, November 1998
15. Heitmeyer C., Labaw B. and Kiskis D., 1995. Consistency Checking of SCR-Style Requirements Specifications, Proc. of the 2nd Int. Symposium on Requirements Engineering, IEEE CS Press, pp. 56-63.
16. Robinson W., Fickas S., 1994. Supporting Multiple Perspective Requirements Engineering, Proc. of the 1st Int. Conference on Requirements Engineering, IEEE CS Press, pp. 206-215