Draft – An updated version of this paper has appeared in the Proceedings of the 7th International Conference on Object Oriented Information Systems

Reconciliation of Object Interaction Models

George Spanoudakis and Hyoseob Kim

Department of Computing,
City University,
Northampton Square, London EC1V 0HB, UK
E–mail: gespan@soi.city.ac.uk, hkim69@soi.city.ac.uk

Abstract: This paper presents Reconciliation+, a tool-supported method which identifies overlaps between models of different object interactions expressed as UML sequence and/or collaboration diagrams, checks whether the overlapping elements of these models satisfy specific consistency rules, and guides developers in handling these inconsistencies. The method also keeps track of the decisions made and the actions taken in the process of managing inconsistencies.

1. Introduction

The design of software systems using the use-case driven approach [23] often results in the generation of multiple object interaction diagrams (i.e. sequence and/or collaboration diagrams). These diagrams are constructed to model specific interactions between the objects in a software system and the actors in its environment, which are required to deliver the functionality described by the particular use case that the diagram describes. These diagrams may be constructed independently by different designers, may advocate specific modelling angles and may reflect disparate perceptions of the system by these designers. As a result, they may have modelling inconsistencies.

Modelling inconsistencies occur when interaction diagrams incorporate messages which overlap – that is they invoke operations with the same implementation – and model these messages in ways that violate specific consistency rules.

As an example, consider an object-oriented design model including the object interaction diagrams I_1 and I_2 in Figures 4 and 5. These diagrams show the interactions between the classes of a library system, which occur when the system is used to search for library items by keywords referring to the author and the title of an item, respectively. Assume also the following consistency rule:

CR1: *If a message m_i overlaps with a message m_j then for every message m_k dispatched by m_i (m_j) there must exist a message m_w dispatched by m_j (m_i) such that m_k and m_w overlap.*

Given the class diagram of Figure 6, it may be reasonably assumed that the messages *7:actionPerformed(ActionEvent)* in I_1 and *10:actionPerformed(ActionEvent)* in I_2 overlap[1]. This is because both of them invoke the operation *actionPerformed(e:ActionEvent)* in the class *DatabaseActionListener*. Given this overlap, these two message violate CR1 since:

- *7:actionPerformed(ActionEvent)* in I_1 dispatches the message *8:getText()* that does not overlap with any of the messages dispatched by *10:actionPerformed(ActionEvent), and*
- *10:actionPerformed(ActionEvent)* in I_2 dispatches the messages *11:getData(),* and *12:formulateQuery()* which do not overlap with any of the messages dispatched by *7:actionPerformed(ActionEvent),*

This paper describes a tool-supported method, called "Reconciliation+", that we have developed to support the detection of overlaps, checking of consistency rules and handling of violations of these rules (i.e., inconsistencies) in models of software interactions expressed as sequence (or collaboration) diagrams in UML [9]. Reconciliation+ has been developed as an extension of a method developed by Spanoudakis and Finkelstein to reconcile object-oriented models of system structures [14]. The newly developed extension:

(i) incorporates a flexible matching algorithm that detects overlaps between messages which may be not identically modelled in diagrams (e.g.

[1] In this paper, we refer to messages by putting a number indicating the order of their dispatch in the interaction diagram they belong to before their signatures. This number uniquely identifies a message in a diagram but is not taken into account in any of the computations described in the paper.

messages with different signatures and messages sent to instances of different classes),

(ii) allows the specification of a wide-range of consistency rules, checks against overlapping messages and allows developers to alter existing consistency rules or add new ones,

(iii) incorporates alternative ways of handling the detected violations of the rules and allows developers to alter them or add new ways of habdling, and

(iv) guides (as opposed to enforcing) developers in selecting which rules to check and how to handle their violations.

The process of reconciling interaction models is driven by the enactment of an explicit process model incorporated by the method.

The rest of the paper is structured as follows. Section 2 presents an overview of Reconciliation+ and the prototype toolkit we have developed to support it. Section 3 describes the specification and enactment of the process model of the method. Section 4 describes the algorithm for detecting overlapping messages in different interaction diagrams. Section 5 describes the specification of consistency rules and the mechanism for detecting violations of these rules. Section 6 describes the scheme for specifying and executing different ways of handling inconsistencies. Section 7 overviews related work and, finally, Section 8 summarises the method and outlines the current and future work on it.

2. Overview of Reconciliation+

Reconciliation+ is supported by a toolkit whose general architecture is shown in Figure 1. This toolkit incorporates a tool that detects overlaps between object interaction models and an engine which enacts a process model that drives the process of reconciling interaction diagrams. This process model specifies:

(i) The tool that may be invoked to identify overlaps in interaction models and the conditions under which this tool may be activated.

(ii) The consistency rules that can be checked against interaction models and the conditions under which these rules may be checked to detect inconsistencies. The consistency rules are expressed as *queries*. In this approach, the queries retrieve the model elements that violate the conditions that the rules require elements to satisfy and therefore violate the rules.

(iii) Different ways of handling inconsistencies (specified by inconsistency handling actions) and the conditions under which each of these ways may be applied.

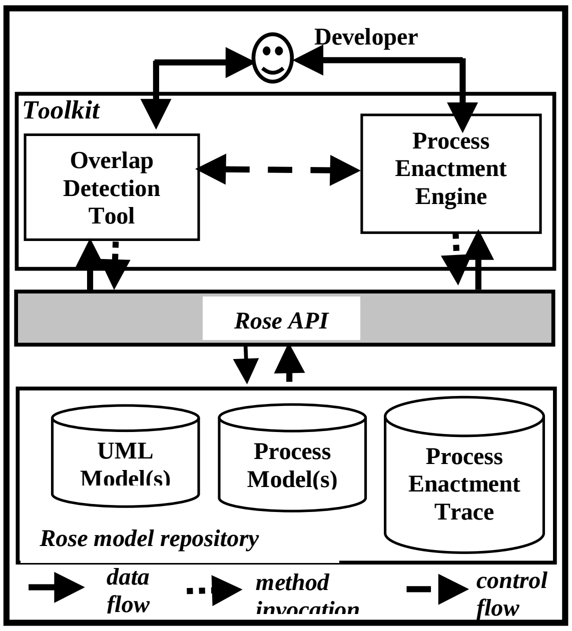


Figure 1: Architecture of the Reconciliation+ toolkit

The process model of the method is enacted by a *process enactment engine.* This engine interprets the process model, presents the available options to the developers, acts according to the option selected (or other instructions given by them), and keeps a trace of the enacted process.

The toolkit of the method has been implemented on the top of *Rational Rose* (a CASE tool supporting UML) using its API. The models to be reconciled are stored as collections of UML class models and sequence diagrams in a model repository managed by Rose. The process model of the method and the traces of the enacted processes are also represented as UML class models in the Rose repository.

3. Specification and enactment of Reconciliation+ processes

3.1 The process meta-model

The process model of Reconciliation+ is specified in UML as an instance of an extended version of the NATURE process meta-model [11,13]. A specification of this meta-model in UML is shown in Figure 2.

According to the Nature meta-model (see non-grey classes in Figure 2) a process is described as a graph of *contexts.* A context represents the decision to pursue a specific goal (*intention*) in a given *situation*. A situation is a condition over the state of the software model being manipulated by the process. Contexts are distinguished into:

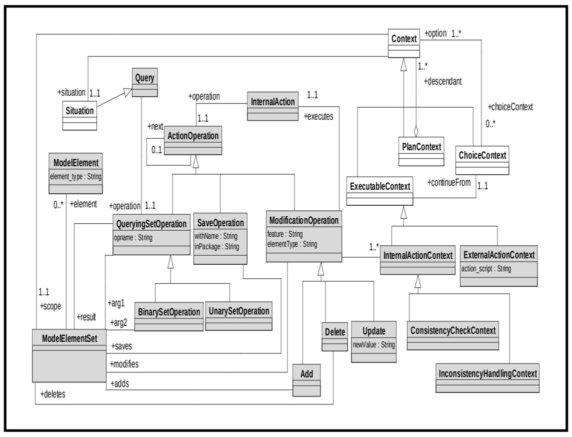


Figure 2: Process Meta-Model in UML

- *executable contexts* – these are contexts whose intentions that can be directly pursued by taking an *action* which changes the state of the software model
- *plan contexts* – these are contexts whose intentions are decomposed into a set of sub-goals which have to be pursued in a specific order
- *choice contexts* – these are contexts whose intentions are refined into one or more alternative goals and can therefore be satisfied by pursuing any of these goals

To be able to specify the process of Reconciliation+, we had to extend the Nature process meta-model in order to establish: (1) a scheme for specifying situations for processes which are enacted upon UML models, and (2) a scheme for specifying the types of actions required for detecting and handling inconsistencies. These extensions were necessary since the Nature meta-model does not determine how to specify context situations and actions. The extensions introduced are described in the following.

3.2 Specification of situations

In the extended meta-model, a situation is a query that is specified as an ordered sequence of *querying set operations*. As shown in Figure 2, a querying set operation can be:

(i) a *UnarySetOperation* – These operations can be used to retrieve: (a) sets of elements (or primitive data values) from a UML model which are associated with a model element *e* via the different kinds of associations or attributes defined by the UML meta-model for the type of *e*, or (b) the elements of a set which satisfy a specific condition (*SelectOperation*). A unary set operation is applied to each of the elements of its *arg1* set and returns the union of the elements retrieved for each of them in its *result* set.

The extended meta-model defines a taxonomy of unary set operations for all the different types of model elements which are defined in the UML meta-model. Figure 3 shows a part of this taxonomy which includes the operations available for retrieving model elements and data values associated with UML classes. As an example, consider the operation *GetAssociations* in this taxonomy. This operation retrieves the associations which relate the classes in its *arg1* set with other classes, creates instances of the process meta-model class *ModelElement* to represent them, and inserts these elements into its *arg2* set (by associating the newly created model elements with this set – see the association end *element* in Figure 2).

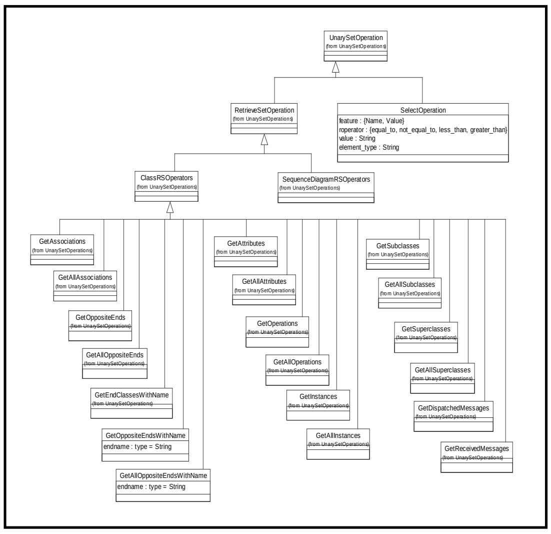


Figure 3: Part of the taxonomy of unary set operations

(ii) a *BinarySetOperation* – These operations realise the operations of set union, intersection and difference. Binary set operations are applied to the sets that constitute their *arg1* and *arg2* sets and generate a *result* set according to the standard definitions of their underlying mathematical operations.

The operations of a situation are executed in the order in which they appear in it. An operation may take as an argument any of the sets generated by an operation appearing before it in a situation. A situation is considered to be satisfied if the result set of the last of its operations is not empty. Examples of situations specified according to this meta-model are given in Section 6 below.

3.3 Specification of actions

Actions are defined for executable contexts and can be specified either as executable script file names (see attribute *action_script* of meta-class *ExternalActionContext* in Figure 2), or as instances of the meta-class *InternalAction*. An executable context may be an *ExternalActionContext* or an *InternalActionContext* depending on whether it includes an action of the former or the latter kind.

External actions are used to invoke tools (e.g. the tool that detects overlaps). Internal actions are used to specify consistency rules (in the operational form that we discuss in Section 5), and ways of handling inconsistencies.

Similarly to situations, internal actions are specified as sequences of *action operations* (see meta-class *ActionOperation* in Figure 2). The taxomomy of the different types of action operations is shown in Figure 2. As shown in this figure, an action operation may be a:

(a) *querying set operation* (see Section 3.2).
(b) *model modification operation* – This is an operation which modifies (i.e., adds, deletes or updates) elements of the retrieved sets (see meta-class *ModificationOperation* in Figure 2).
(c) *save operation* – This is an operation which saves retrieved sets of elements in the process enactment trace (see meta-class *SaveOperation* in Figure 2).

Sequences of internal action operations are interpreted and executed by the process enactment engine similarly to the set querying operations of situations. Examples of internal actions used to detect, and handle inconsistencies are given in Sections 5 and 6, respectively.

3.4 Process enactment

The Reconciliation+ process model is enacted by an engine which functions as a model interpreter [13]. The full algorithm implemented by the process enactment engine of the method is beyond the scope of this paper and may be found in [22]. In this section, we give only an overview of the enactment of process models in Reconciliation+.

The enactment of a model starts from the single root choice context that every process model must have. The situation of this context (and any other context that is encountered as the enactment engine traverses the process model) is evaluated by executing its set querying operations. If the set which results from the execution of the last of these operations (called "situation set") is not empty, the situation of the context is satisfied. If a situation is satisfied, the enactment engine generates N different possible decisions from its context where N is the number of the elements of its evaluated situation set.

A *decision* is a pair
`<context`$_i$`, situation_set_element`$_j$`>`
where
- `context`$_i$ is the context whose situation is satisfied, and
- `situation_set_element`$_j$ is an element of the situation set of `context`$_i$.

The developer may select one of these decisions, ask for tactical guidance or terminate the process. If a decision `<context`$_i$`,situation_set_element`$_j$`>` is selected, the decision is recorded in the process trace (see Figure 1). Subsequently:

- If `context`$_i$ is a choice context the enactment engine: (1) retrieves the option contexts associated with it, (2) inserts the `situation_set_element`$_j$ in the *arg1* set of the initial querying set operations of each of these contexts, (3) evaluates the situation of each of these contexts, (4) generates the possible decisions for each of these contexts, and (5) prompts the developer to make a new selection.

- If `context`$_i$ is an external action context the enactment engine executes the file specified by its attribute *action_script* and continues the enactment of the process model from the context associated with `context`$_i$ via the end *continueFrom* (see Figure 2).

- If `context`$_i$ is an internal action context the enactment engine executes the sequence of the operations in its internal action and continues the enactment of the process model as in the case of external action contexts.

Plan contexts are not used in the current process model of the method and are not supported in the current implementation. In cases where the developer asks for tactical guidance, the enactment engine identifies the decision before the last decision recorded in the process trace and resumes execution from the context of it. The developer may abort the execution of the process model at any point.

4. Detection of overlaps

4.1 Basic algorithmic formulation

The detection of overlaps between sequence diagrams is formulated as an instance of the *weighted bipartite graph matching problem* [10]. Assuming a pair of sequence diagrams modelling two object interactions I_i and I_j, we construct an *interaction overlap graph:*

$$IOG(I_i, I_j) = (V_i \cup V_j, E(V_i,V_j))$$

This graph has two sets of vertices V_i and V_j. Assuming (without loss of generality) that I_i has more messages than I_j, V_i and V_j are defined as:

$$V_i \equiv Messages(I_i) \text{ and } V_j \equiv Messages(I_j) \cup DV_k$$

where DV_k is a set of k special vertices representing dummy messages ($k = |Messages(I_i)| - |Messages(I_j)|$). Vi and Vi are disjoint sets since each message must belong to only one interaction (see [9]). Thus, $IOG(I_i,I_j)$ is a bipartite graph.

The set of the edges $E(V_i,V_j)$ includes only edges which connect the messages of I_i with the messages of I_j:

$$E(V_i,V_j) = \{(n_i ,n_j, b_0(\neg ov(mes(n_i),mes(n_j)))) \mid (n_i \in V_i) \text{ and } (n_j \in V_j)\}$$

An edge $(n_i,n_j,b_0(\neg ov(mes(n_i),mes(n_j))))$ designates the assumption that the messages represented by the nodes n_i and n_j (these are *mes(ni)* and *mes(nj)*) overlap with each other and is weighted by a degree of belief in the falsity of this assumption $b_0(\neg ov(mes(n_i),mes(n_j)))$. The overlap relation is denoted by the predicate *$ov(mes(n_i),mes(n_j))$*.

The degrees of belief b_0 are computed according to the formula:

$$b_0(\neg ov(m_i,m_j)) = \Sigma_{U\subseteq\{1,\ldots,6\}}(-1)^{|U|+1}\{\Pi_{u\in U}\, b_u(\neg ov(m_i,m_j))\} \quad (I)$$

The functions $b_1, \ldots, b_6$ used in (I) are defined in Section 4.2 below. After computing the beliefs b_0 for all the edges of $IOG(I_i,I_j)$, the overlaps between the messages in I_i and I_j are detected in a two-step process. First, the most likely candidate overlaps are identified by selecting a subset $O(V_i,V_j)$ of $E(V_i,V_j)$ which is a total morphism between V_i and V_j and minimises the function:

$$\sum_{(n_u,\, n_w,\, b_0(\neg ov(mes(n_u),mes(n_w)))) \in O(V_i,V_j)} b_0(\neg ov(mes(n_u),mes(n_w))) \quad (II)$$

The morphism $O(V_i,V_j)$ is selected using an algorithm known as the *Hungarian method* [10]. In the second step, $O(V_i,V_j)$ is further restricted to include only the edges $(n_u, n_w, b_0(\neg ov (mes(n_u),mes(n_w)))$ whose non overlap belief does not exceed a particular threshold value b_t, that is:

$$b_0(\neg ov(mes(n_u),mes(n_v))) \leq b_t$$

4.2 Underlying belief functions

The functions $b_1, \ldots, b_6$ used in (I) are computed according to six indicators of the non existence of overlap relations between messages, namely the indicator of *non-equivalent message operations*, and the indicators of *different message senders*, *receivers*, *stereotypes*, *activators* and *activation*s. These functions are defined in the following.

4.2.1 Belief due to non-equivalent operations: b_1

b_1 is the belief in the non existence of an overlap between two messages m_i and m_j computed due the *non equivalent operations indicator*. According to this indicator, two messages do not overlap if they invoke operations which do not override the same most general operation in a class model. b_1 is defined as follows:

Definition 1: The functional form of b_1 is:

$$b_1(\neg ov(m_i,m_j)) = \alpha_1 \times d_1(m_i,m_j)$$
$$b_1(ov(m_i,m_j)) = 0$$

where

- $d_1(m_i,m_j) = (|Ops(o_i) - Ops(o_j)| + |Ops(o_j) - Ops(o_i)|) \;/\; |Ops(o_i) \cup Ops(o_j)|$
- o_i is the operation invoked by m_i and o_j is the operation invoked by m_j
- $Ops(o_i)$ is the set of operations having the same signature with o_i which are defined in the superclasses of the class that defines o_i and which do not override any operation with the same signature as o_i

- α_1 is the expected ratio of messages m_i and m_j with non-equivalent operations which do not overlap ($0 \leq \alpha_1 \leq 1$)

In single inheritance class models, $Ops(o_i)$ and $Ops(o_j)$ are always singleton sets and, therefore, b_1 is always equal to α_1 or 0. In multiple inheritance class models, where there may be ambiguities regarding the most general operation that is overridden by an operation, each of $Ops(o_i)$ and $Ops(o_j)$ may have more than one elements (see for example). In such cases, b_1 can take any value in the range $[0,\ldots, \alpha_1]$.

4.2.2 Belief due to different message senders: b_2

b_2 is the belief in the non existence of an overlap between two messages m_i and m_j computed due to the indicator of *different senders*. According to this indicator, two messages do not overlap if they are sent by objects which are instances of different classes (called "senders" in UML). b_2 is defined as follows:

Definition 2: The functional form of b_2 is:

$b_2(\neg ov(m_i,m_j)) = \alpha_2 \times d_2(s_i,s_j)$

$b_2(ov(m_i,m_j)) = 0$

where

- s_i is the class of the object that sends m_i and s_j is the class of the object that sends m_j
- $d_2(c_i, c_j) = \sum_{x \in NCSij} SD(x)^{-1} / \sum_{y \in ASSij} SD(y)^{-1}$ if c_i and c_i are specified in the models
- $d_2(c_i, c_j) = 1$ if c_i or c_i is not specified in the models
- NCS_{ij} is the set difference of the superclasses of c_i and c_j
- $SD(x)$ is the maximum length (number of links) of the paths connecting a class x with the most general class of its generalisation hierarchy, called *specialisation depth* of x
- α_2 is the expected ratio of messages m_i and m_j with different senders which do not overlap ($0 \leq \alpha_2 \leq 1$)

b_2 measures the belief in the non-identity of two classes based on the number of their non common superclasses and the relative depth of them in the generalisation graph(s) of the models.

4.2.3 Belief due to different message receivers: b_3

b_3 is the belief in the non existence of an overlap between two messages m_i and m_j computed due to the indicator of *different receivers.* According to this indicator, two messages do not overlap if they are received by objects which are instances of different classes (called "receivers" in UML). Similarly to b_2, b_3 is defined as follows:

Definition 3: The functional form of b_3 is:

$b_3(\neg ov(m_i,m_j)) = \alpha_3 \times d_2(r_i,r_j)$

$b_3(ov(m_i,m_j)) = 0$

where

- r_i is the class of the object that receives m_i and r_j is the class of the object that receives m_j
- d_2 is as defined in Definition 3 above
- α_3 is the expected ratio of messages m_i and m_j with different receivers which do not overlap ($0 \leq \alpha_3 \leq 1$)

4.2.4 Belief due to different message stereotypes: b_4

$b_4$is the belief in the non existence of an overlap between two messages m_i and m_j computed due to the indicator of *different stereotypes.* According to this indicator, two messages do not overlap if they are have different stereotypes and, therefore, they belong to semantically different groups of messages. b_4 is defined as follows:

Definition 4: The functional form of b_4 is:

$b_4(\neg ov(m_i,m_j)) = \alpha_4 \times d_4(m_i,m_j)$

$b_4(\neg ov(m_i,m_j)) = 0$

where

- $d_4(m_i,m_j) = (|Stypes(m_i) - Stypes(m_j)| + |Stypes(m_j) - Stypes(m_i)|) / |Stypes(m_i) \cup Stypes(m_j)|$
- $Stypes(m_i)$ and $Stypes(m_j)$ are the sets of the stereotypes of the messages m_i and m_j, respectively
- α_4 is the expected ratio of messages m_i and m_j with different stereotypes which do not overlap ($0 \leq \alpha_4 \leq 1$)

b_4 measures a belief based on the number of the non common stereotypes of two messages.

4.2.5 Belief due to different message activators: b_5

b_5 is the belief in the non existence of an overlap between two messages m_i and m_j computed due to the indicator of *different activators*. According to this indicator, two messages do not overlap if they are dispatched by non overlapping messages (called "activators" in UML) or equivalently (in terms of our definition of message overlaps) in the course of execution of operations with different implementations. b_5 is defined as follows:

Definition 5: The functional form of b_5 is:

$$b_5(\neg ov(m_i,m_j)) = \alpha_4 \times \Sigma_{U \subseteq \{1,\ldots,4\}} (-1)^{|U|+1} \{\Pi_{u \in U} b_u(\neg ov(m_i,m_j))\}$$

$$b_5(ov(m_i,m_j)) = 0$$

where

- b_u (u=1,…,4) are as defined in Definitions 1-4
- α_5 is the expected ratio of messages m_i and m_j with different activators which do not overlap ($0 \leq \alpha_5 \leq 1$)

b_5 is an approximation of b_6 used to avoid a double full analysis of activators in detecting overlaps.

4.2.6 Belief due to different message activations: b_6

b_6 is the belief in the non existence of an overlap between two messages m_i and m_j computed due to the indicator of *different activations*. According to this indicator, two messages do not overlap if they dispatch non overlapping messages (called "activations" in UML) or, equivalently (in terms of our definition of overlap), they invoke operations with different implementations. b_6 is defined as follows:

Definition 6: The functional form of b_6 is:

$$b_6(\neg ov(m_i,m_j)) = \alpha_6 \times d_6(m_i,m_j)$$

$$b_6(ov(m_i,m_j)) = 0$$

where

- $d_6(m_i,m_j) = (\min_{X \in Morphisms(i,j)} (\sum_{(mu,mv) \in X} b_o(\neg ov(m_u,m_v)) + \max(|A_i| - |A_j|, |A_j| - |A_i|)) / \max(|A_i|, |A_j|)$ if $A_i \neq \emptyset$ and $A_j \neq \emptyset$
  $d_6(m_i,m_j) = 1$ if $A_i = \emptyset$ or $A_j = \emptyset$.
- A_i and A_j are the sets of the messages which are dispatched by the messages m_i and m_j, respectively.
- M(i,j) is the set of all the total morphisms from the messages in A_i to the messages in A_j if $|A_i| \leq |A_j|$ or the set of all the onto morphisms from the messages in A_i to the messages in A_j if $|A_j| < |A_i|$.
- $b_o(\neg ov(m_u,m_v))$ is as defined by formula (I).
- α_6 is the expected ratio of messages m_i and m_j with different activators which do not overlap ($0 \leq \alpha_6 \leq 1$)

b_6 measures a belief based on a recursive check for overlaps between the messages which are dispatched by the messages of concern.

4.2.7 Properties of used belief functions

As we prove in [21], the functions $b_1,\ldots,b_6$ are all distance metrics. They also satisfy the axioms that define *Dempster-Shafer basic probability assignments* [19] and, as a consequence, they are legitimately interpreted as probability assignments of this kind. The main reason for this interpretation is that $b_1,\ldots,b_6$ have been defined so as to assign measures of belief in the non existence of an overlap relation between messages and not to the existence of such relations, that is $b_i(ov(m_u,m_v)) = 0$ for all i=1,…,6. This is because of the indicators underlying these functions. More specifically, it has to be appreciated that while the presence of these indicators in the case of two messages does indicate that an overlap relation between them is unlikely, the absence of the indicators does not provide evidence that such an overlap relation exists between the messages.

Consider, for instance, the indicator of different receivers. When two messages have different receivers in interaction diagrams, it is more plausible to assume that they invoke operations with different implementations than to assume that they invoke operations with the same implementation. However, when two messages have the same receiver we cannot conclude that an overlap relation between them exists. Similar arguments apply for the other indicators.

The functions b_i (i=1,…,6) have been defined so as to: $b_i(ov(m_u,m_v)) + b_i(\neg ov(m_u,m_v)) \leq 1$. Belief functions that have this property of assigning a total belief of possibly less than 1 to a proposition and its negation are valid as Dempster-Shafer basic probability assignments but not as classical probability functions for which it must always be that $Prob(E) + Prob(\neg E) = 1$ [16].

As we prove in [21], the functional form of b_0 is derived from the combination of the belief functions b_1, …,b_6 using the rule of the *orthogonal sum* of the Dempster-Shafer theory and measures the belief jointly committed to $\neg ov(m_i,m_j)$ by these functions. b_0 has also been proved to be a distance metric (see [21]). These properties of b_0 guarantee the following intuitively plausible relations between its outputs:

- $b_0(\neg ov(m_i,m_k)) \leq b_0(\neg ov(m_i,m_j)) + b_0(\neg ov(m_j,m_k))$ (due to triangularity of distance metrics)
- $b_0(\neg ov(m_i,m_j)) = b_0(\neg ov(m_j,m_i))$ (due to symmetry of distance metrics)
- $b_0(\neg ov(m_i,m_j) \wedge ov(m_j,m_i)) = 0$ (due to axiomatic foundation of Dempster-Shafer basic probability assignments)

Finally, the measure that results from the minimisation of formula (II) is an upper bound of:

$$b_0\Big(\bigvee_{(n_u,\, n_w,\, b0(mes(n_u),mes(n_w)))\, \in\, O(Vi,Vj)} \neg ov(mes(n_u),mes(n_v))\Big)$$

This is the belief in the proposition that at least one of the overlap relations in $O(V_i,V_j)$ is wrong.

4.3 Example of detection of optimal overlap morphism

As an example of detecting overlaps consider the sequence diagrams I_1 and I_2 in Figures 4 and 5, respectively. These diagrams specify interactions between the objects in a library system which occur when a user uses it to search for a library item by keywords referring to the title (I_1) or to the author of an item (I_2).

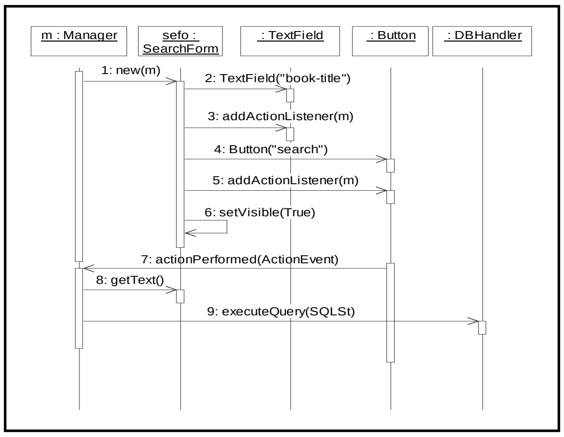


Figure 4: Interaction diagram I_1- SearchByTitle

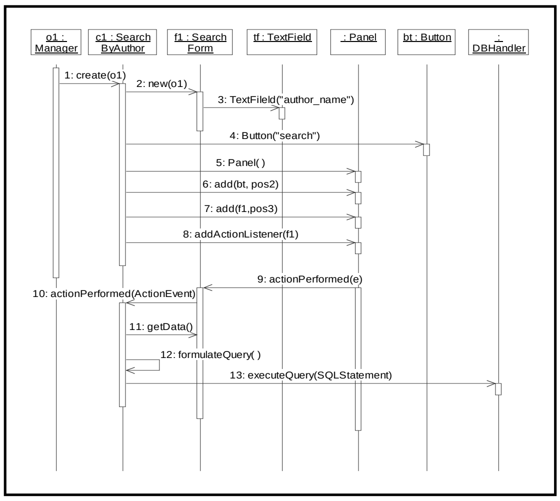


Figure 5: Interaction diagram I_2- SearchByAuthor

I_1 and I_2 assume the event-driven GUI implementation model of the Java 1.1 Abstract Window Toolkit [20]. More specifically, in I_1 the class *SearchForm* has been declared as a subclass of the Java AWT component *Panel* (see Figure 6) and has a text field to let the user to type in the keywords that should be searched within the titles of items. In I_2, the class *SearchByAuthor* incorporates a panel composed of a text field and a button that enable the user type in the author keywords and initiate the search.

When a search is initiated the system gets the search keywords from the appropriate UI component (see messages *8:getText()* in I_1 and *11:getData()* in I_2). formulates a query (see message *12:formulateQuery()* in I_2), and forwards this query to a database driver to execute it (see messages *9:executeQuery(SQLSt)* in I_1 and *13:executeQuery(SQLStatement)* in I_2).

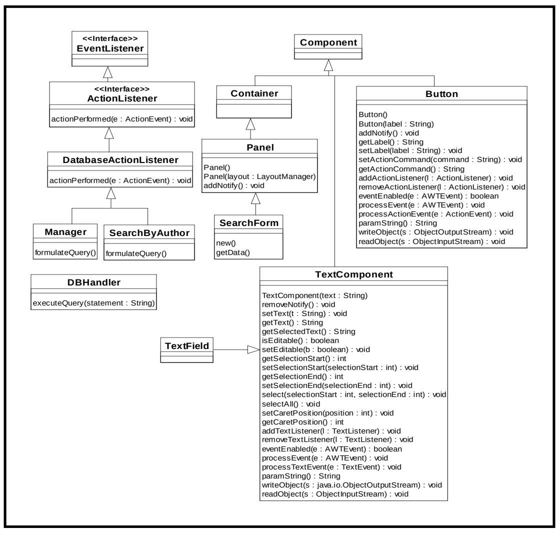

<<Interface>> EventListener
Component
<<Interface>> ActionListener
actionPerformed(e : ActionEvent) : void
Container
Button
Button()
Button(label : String)
addNotify() : void
getLabel() : String
setLabel(label : String) : void
setActionCommand(command : String) : void
getActionCommand() : String
addActionListener(l : ActionListener) : void
removeActionListener(l : ActionListener) : void
eventEnabled(e : AWTEvent) : boolean
processEvent(e : AWTEvent) : void
processActionEvent(e : ActionEvent) : void
paramString() : String
writeObject(s : ObjectOutputStream) : void
readObject(s : ObjectInputStream) : void
Panel
Panel()
Panel(layout : LayoutManager)
addNotify() : void
DatabaseActionListener
actionPerformed(e : ActionEvent) : void
Manager
formulateQuery()
SearchByAuthor
formulateQuery()
SearchForm
new()
getData()
DBHandler
executeQuery(statement : String)
TextComponent
TextComponent(text : String)
removeNotify() : void
setText(t : String) : void
getText() : String
getSelectedText() : String
isEditable() : boolean
setEditable(b : boolean) : void
getSelectionStart() : int
setSelectionStart(selectionStart : int) : void
getSelectionEnd() : int
setSelectionEnd(selectionEnd : int) : void
select(selectionStart : int, selectionEnd : int) : void
selectAll() : void
setCaretPosition(position : int) : void
getCaretPosition() : int
addTextListener(l : TextListener) : void
removeTextListener(l : TextListener) : void
eventEnabled(e : AWTEvent) : boolean
processEvent(e : AWTEvent) : void
processTextEvent(e : TextEvent) : void
paramString() : String
writeObject(s : java.io.ObjectOutputStream) : void
readObject(s : ObjectInputStream) : void
TextField

Figure 6: Generalisation graph of the classes in I_1 and I_2

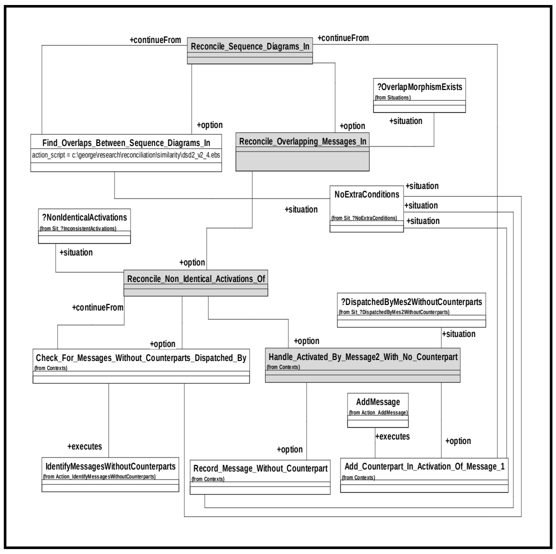
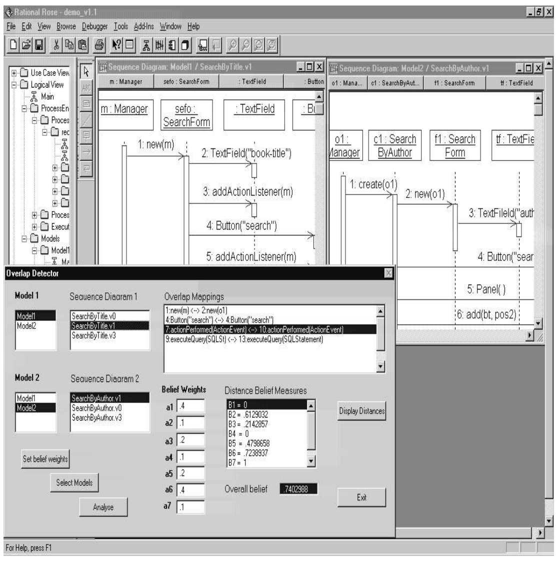

+continueFrom
Reconcile_Sequence_Diagrams_In
+continueFrom
?OverlapMorphismExists
(from Situations)
+situation
+option
+option
Find_Overlaps_Between_Sequence_Diagrams_In
action_script = c:\george\research\reconciliation\similarity\dsd2_v2_4.ebs
Reconcile_Overlapping_Messages_In
NoExtraConditions
(from Sit_?NoExtraConditions)
+situation
+situation
+situation
+situation
?NonIdenticalActivations
(from Sit_?InconsistentActivations)
+situation
+option
Reconcile_Non_Identical_Activations_Of
+continueFrom
?DispatchedByMes2WithoutCounterparts
(from Sit_?DispatchedByMes2WithoutCounterparts)
+situation
+option
+option
Check_For_Messages_Without_Counterparts_Dispatched_By
(from Contexts)
Handle_Activated_By_Message2_With_No_Counterpart
(from Contexts)
AddMessage
(from Action_AddMessage)
+executes
+executes
+option
+option
IdentifyMessagesWithoutCounterparts
(from Action_IdentifyMessagesWithoutCounterparts)
Record_Message_Without_Counterpart
(from Contexts)
Add_Counterpart_In_Activation_Of_Message_1
(from Contexts)

Figure 7: Part of the Reconciliation+ process model

The detection of overlaps can be activated during the enactment of the Reconciliation+ process model by selecting the executable context *Find_Overlaps_Between_Sequence_Diagrams_In* shown in Figure 7. This figure shows part of the Reconciliation+ process model (the choice contexts are grey and the executable contexts are white).


Overlap Detector

Model 1: Model1 (selected), Model2
Sequence Diagram 1: SearchByTitle.v0, SearchByTitle.v1 (selected), SearchByTitle.v3
Overlap Mappings:
1:new(m) <-> 2:new(o1)
4:Button("search") <-> 4:Button("search")
7:actionPerformed(ActionEvent) <-> 10:actionPerformed(ActionEvent)
9:executeQuery(SQLSt) <-> 13:executeQuery(SQLStatement)

Model 2: Model1, Model2 (selected)
Sequence Diagram 2: SearchByAuthor.v1 (selected), SearchByAuthor.v0, SearchByAuthor.v3

Belief Weights: a1 .4, a2 .1, a3 .2, a4 .1, a5 .2, a6 .4, a7 .1
Distance Belief Measures: B1 = 0, B2 = .6129032, B3 = .2142857, B4 = 0, B5 = .4798658, B6 = .7238937, B7 = 1
Overall belief 7402988

Figure 8: Screenshot of Overlap Detector

Find_Overlaps_Between_Sequence_Diagrams_In is instantiated for a default package of the underlying tool repository that includes the models to be reconciled and is an external action executable context that invokes the overlap detector.

As shown in Figure 8, the overlaps detected between the messages of I_1 and I_2 are:

- *ov(1:new(m), 2:new(o1))* – $b_1 = 0$, $b_2 = 0.21$, $b_3 = 0$, $b_4 = 0$, $b_5 = 1$, $b_6 = 0.93$, and $b_0 = 0.559$
- *ov(4:Button("search"), 4:Button("search"))* – $b_1 = 0$, $b_2 = 1$, $b_3 = 0$, $b_4 = 0$, $b_5 = 0.52$, $b_6 = 1$, and $b_0 = 0.564$
- *ov(7:actionPerformed(ActionEvent), 10: actionPerformed(ActionEvent))* – $b_1 = 0$, $b_2 = 0.61$, $b_3 = 0.21$, $b_4 = 0$, $b_5 = 0.48$, $b_6 = 0.72$, and $b_0 = 0.48$
- *ov(9:executeQuery(SQLSt), 13: executeQuery(SQLStatement))* – $b_1 = 0$, $b_2 = 0.21$, $b_3 = 0$, $b_4 = 0$, $b_5 = 0.1$, $b_6 = 1$, and $b_0 = 0.48$

The above relations were detected using a threshold value of 0.6 for b_t and the α_i values shown in Figure 8.

The overlaps detected by the Overlap Detector are represented in the process enactment trace as shown in Figure 9. Each of the detected overlap relations is represented by an object which is associated with the messages involved in the relation. This object also stores the measures generated by the functions $b_1, \ldots, b_6$ as values of specific attributes of its own.

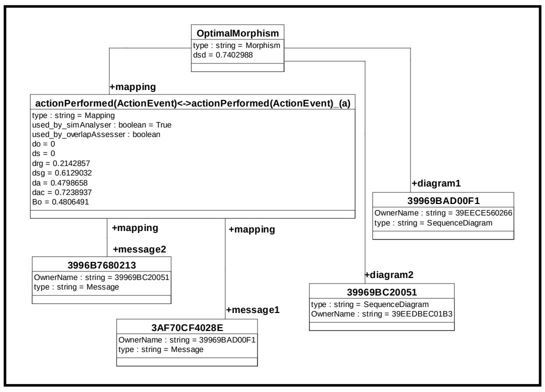

OptimalMorphism
type : string = Morphism
dsd = 0.7402988
+mapping
actionPerformed(ActionEvent)<->actionPerformed(ActionEvent)_(a)
type : string = Mapping
used_by_simAnalyser : boolean = True
used_by_overlapAssesser : boolean
do = 0
ds = 0
drg = 0.2142857
dsg = 0.6129032
da = 0.4798658
dac = 0.7238937
Bo = 0.4806491
+diagram1
39969BAD00F1
OwnerName : string = 39EECE560266
type : string = SequenceDiagram
+mapping
+message2
+mapping
3996B7680213
OwnerName : string = 39969BC20051
type : string = Message
+diagram2
39969BC20051
type : string = SequenceDiagram
OwnerName : string = 39EEDBEC01B3
+message1
3AF70CF4028E
OwnerName : string = 39969BAD00F1
type : string = Message

Figure 9: Optimal morphism in the process execution trace

For example, the object "actionPerformed(ActionEvent)<-> actionPerformed(ActionEvent)" represents the overlap relation between the so called messages of I_1 and I_2, and is associated with objects that represent these two messages (they appear with their internal identifiers in Figure 8). It also stores their d_3 distance (as the value of its attribute *drg*) and is associated with an object that represents the selected optimal morphism between I_1 and I_2 (called "OptimalMorphism"). All these objects are stored in a special package of the tool repository called "ExecutionClasses" which is used to store information generated during the enactment of the process model of the method.

5. Specification of consistency rules and detection of inconsistencies

In our method, consistency rules are specified as internal actions of a special kind of executable internal action contexts, called "consistency check contexts". These contexts are instances of the meta-class *ConsistencyCheckContext* in Figure 2.

As we discussed in Section 3.1, an internal action can generally be specified as a sequence of set, save, or modification operations. In the case of consistency check contexts, however, the internal actions are restricted not to include any modification operations. This restriction guarantees that the execution of a consistency check will not modify the contents of the underlying models.

Furthermore, the last operation in the sequence of operations that specify a consistency rule must be a *SaveOperation*. Save operations save a set in the process execution trace (i.e. the special package called "ExecutionClasses"). In the case of an action that specifies a consistency rule, this set includes the model elements (messages) which breach the rule. This restriction realises a well-formedness rule for the Reconciliation+ process models which guarantees that the results of a consistency check will always be made temporarily available to the enactment of the process[2].

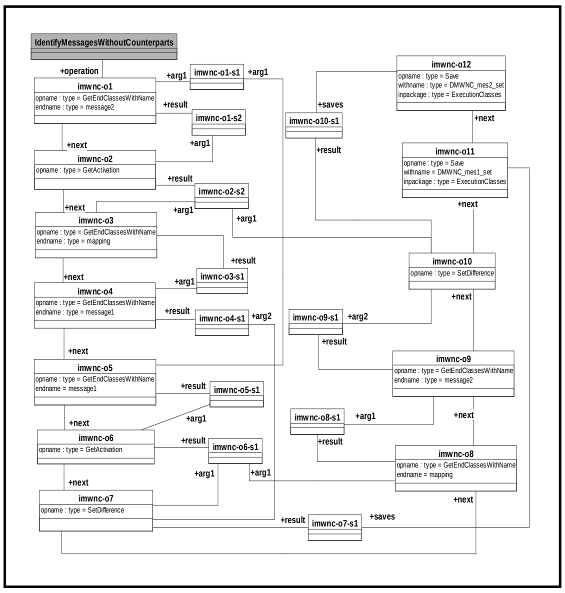

IdentifyMessagesWithoutCounterparts
+operation
imwnc-o1
opname : type = GetEndClassesWithName
endname : type = message2
+arg1
imwnc-o1-s1
+arg1
+result
imwnc-o1-s2
+next
+arg1
imwnc-o2
opname : type = GetActivation
+result
imwnc-o2-s2
+next
+arg1
+arg1
imwnc-o3
opname : type = GetEndClassesWithName
endname : type = mapping
+result
inwnc-o3-s1
+next
+arg1
imwnc-o4
opname : type = GetEndClassesWithName
endname : type = message1
+result
imwnc-o4-s1
+arg2
+next
imwnc-o5
opname : type = GetEndClassesWithName
endname = message1
+result
imwnc-o5-s1
+next
+arg1
imwnc-o6
opname : type = GetActivation
+result
imwnc-o6-s1
+arg1
+arg1
+next
imwnc-o7
opname : type = SetDifference
+result
imwnc-o7-s1
+saves
imwnc-o12
opname : type = Save
withname : type = DMWNC_mes2_set
inpackage : type = ExecutionClasses
+saves
imwnc-o10-s1
+result
+next
imwnc-o11
opname : type = Save
withname = DMWNC_mes1_set
inpackage : type = ExecutionClasses
+next
imwnc-o10
opname : type = SetDifference
+next
imwnc-o9-s1
+arg2
+result
imwnc-o9
opname : type = GetEndClassesWithName
endname : type = message2
imwnc-o8-s1
+arg1
+next
+result
imwnc-o8
opname : type = GetEndClassesWithName
endname = mapping
+next

Figure 10: Specification of action *IdentifyMessagesWithoutCounterparts*

Next, we give an example of a consistency rule specified according to the scheme outlined above and show when exactly it may be checked in the overall process of Reconciliation+ and the results of its checking.

5.1 Specification of consistency rules: an example

As an example of a consistency rule specified according to the scheme outlined above, consider again the rule CR1. As discussed in Section 1, CR1 requires that for each message x activated by a message m_i which overlaps with a message m_j there must be a message y activated by m_j that overlaps with x and vice versa.

[2] They can also become permanently available through an inconsistency handling action that saves an inconsistency recorded in the process trace in a special package called *Inconsistencies* whose contents persist the completion of a process enactment session.

In the process model of Reconciliation+, CR1 is specified as the action *IdentifyMessagesWithoutCounterparts* of the consistency check context *Check_For_Messages_Without_Counterparts_Dispatched_By* shown in Figure 7. To check this rule against a pair of overlapping messages, the developer can select a decision that applies *Check_For_Messages_Without_Counterparts_Dispatched_By* to the object that represents the overlap relation detected these messages (see Figure 9). This overlap relation will be referred to as the "selected overlap object" in the following.

As shown in Figure 10, *IdentifyMessagesWithoutCounterparts* is specified as a sequence of the following operations:

1. *imwnc-o1 – imwnc-o1* is an instance of the unary set operation *GetEndClassesWithName* which returns in its *result* set the classes which are attached to the opposite association ends of the classes in its *arg1* set having the same name as the value of its attribute *endname*. The *arg1* set of *imwnc-o1* always includes the selected overlap object. Thus, *imwnc-o1* retrieves the *message2* of the overlap relation represented by it (for example the object "3996B7680213" in Figure 9).
2. *imwnc-o2 – imwnc-o2* is an instance of the unary set operation *GetActivation*. *GetActivation* returns in its *result* set the union of the sets of the messages which are activated by the messages in its *arg1* set. Thus, *imwnc-o2* retrieves the messages which are dispatched by the *message2* of the selected overlap object.
3. *imwnc-o3 – imwnc-o3* is an instance of the unary set operation *GetEndClassesWithName* and retrieves the objects that represent the overlap relations detected for the messages dispatched by *message2* of the selected overlap object.
4. *imwnc-o4 – imwnc-o4* is an instance of the unary set operation *GetEndClassesWithName* and retrieves the messages activated by *message1* of the selected overlap object and which overlap with the messages activated by the *message2* of this object.
5. *imwnc-o5 – imwnc-o5* is an instance of the unary set operation *GetEndClassesWithName* and retrieves the *message1* of the selected overlap object.
6. *imwnc-o6 – imwnc-o6* is an instance of the operation *GetActivation* and retrieves the set of

messages which are activated by *message1* of the selected overlap object.

7. *imwnc-o7* – *imwnc-o7* is an instance of the binary set operation *SetDifference* which returns the set difference between its *arg1* and *arg2* set in its *result* set. Thus, *imwnc-o7* returns the messages which have are directly dispatched by the *message1* of the selected overlap object and do not overlap with any of the messages which are directly dispatched by the *message2* of this object.
8. *imwnc-o8* – Similarly to *imwnc-o4, imwnc-o8* retrieves the overlap relations detected for the messages which are directly dispatched by the *message1* of the selected overlap object.
9. *imwnc-o9* – Similarly to *imwnc-o3, imwnc-o9* retrieves the messages which overlap with the messages dispatched by the *message1* of the selected overlap object.
10. *imwnc-o10* – Similarly to *imwnc-o7, imwnc-o10* returns the messages which are directly dispatched by the *message2* of the selected overlap object and which do not overlap with any of the messages dispatched by the *message1* of this object.
11. *imwnc-o11* – *imwnc-o11* is an instance of the *SaveOperation* which saves with the name indicated by the value of its attribute *withname* and in the package indicated by the value of its attribute *inpackage* its *saves* set. Thus, *imwnc-o11* saves in the package *ExecutionClasses* the set of the messages which are dispatched by *message1* of the selected overlap object and do not overlap with any of the messages dispatched by the *message2* of this object. The name of the saved set is *DMWNC_mes1_set.*
12. *imwnc-o12* – Similarly to *imwnc-o11*, *imwnc-o12* saves in the package *ExecutionClasses* the set of the messages which are dispatched by *message2* of the selected overlap object and do not overlap with any of the messages which are dispatched by the *message1* of this object. The name of the saved set is *DMWNC_mes2_set.*

5.2 Execution of consistency rules

A consistency check context becomes available for selection only in certain parts of the reconciliation process (as determined by the process model) and when the situation associated with it is satisfied.

Following the process model of Figure 7, the consistency check context *Check_For_Messages_Without_Counterparts_Dispatched_By*, for example, may be selected for an overlapping pair of messages only after a designer has selected:

1) the executable context *Find_Overlaps_Between_Sequence_Diagrams_In* to detect overlaps relations in the sequence diagrams to be reconciled
2) the choice context *Reconcile_Overlapping_Messages_In* to start the reconciliation of the overlapping messages detected in these sequence diagrams, and
3) the choice context *Reconcile_Non_Identical_Activations_Of* to start the reconciliation of the overlapping messages in these sequence diagrams with potentially non identical activations

The selection of *Check_For_Messages_Without_Counterparts_Dispatched_By* for the object representing the overlap between the messages *7:actionPerformed(ActionEvent)* in I_1 and *10:actionPerformed(ActionEvent))* in I_2 generates the following sets of non overlapping messages in the process enactment trace:

- DMWNC_mes1_set = {getText()}
- DMWNC_mes2_set = {getData(), formulateQuery()}

6. Handling of Inconsistencies

Similarly to the consistency checks, the ways of handling inconsistencies are specified as internal actions of a special kind of executable contexts, called *inconsistency handling contexts* (see Figure 2). In this section, we discuss how the situations which determine when these contexts become available and the actions which determine how the inconsistencies can be dealt with are specified.

The process model of Reconciliation+ incorporates contexts for handling inconsistencies which arise as violations of different consistency rules. And in cases where there are more than one possible ways of handling the violations of a particular rule, it includes contexts for each of these ways. These contexts are options of choice contexts in the process model which exist to group the different ways of handling a particular kind of inconsistencies. The situations of these choice contexts are defined in a way that makes them selectable only if there have been recorded violations of the relevant rule in the process enactment trace.

As an example, consider the inconsistencies recorded as violations of the rule specified by the internal action of *Check_For_Messages_Without_Counterparts_Dispatched_By*. As we discussed in Section 5.1, the internal action of this context detects and keeps a record of messages which violate the rule CR1. The process model defines four contexts for handling these violations, including the contexts:

1) *Record_Message_Without_Counterpart*, and
2) *Add_Counterpart_In_Activation_Of_Message_1*.

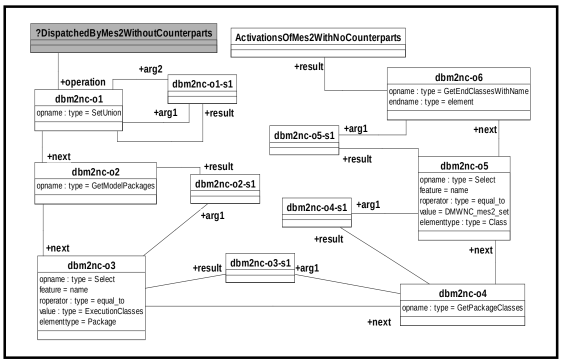
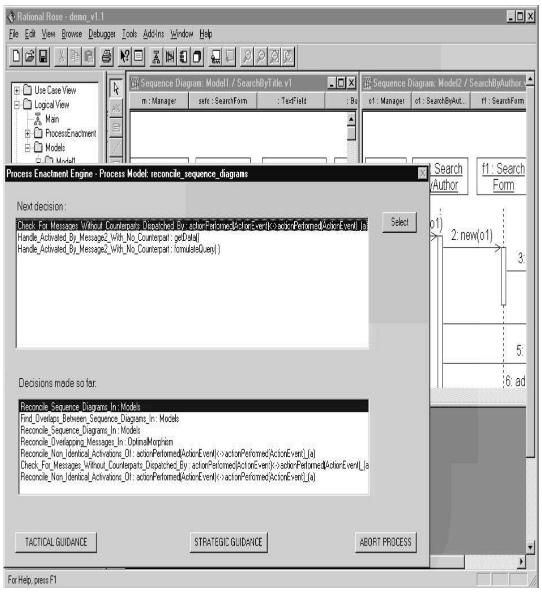


Figure 11: Specification of situation *?DispatchedByMes2WithoutCounterparts*

As shown in Figure 7, these contexts are options available from the choice context *Handle_Activated_By_Message2_With_No_Counterpart* which becomes available only if there are messages that violate CR1. The satisfiability of this condition is checked when the process enactment engine executes the sequence of the operations of the situation *?DispatchedByMes2WithoutCounterparts* of this context shown in Figure 11. As shown in this figure, *?DispatchedByMes2WithoutCounterparts* is satisfied only if there is a class called "DMWNC_mes2_set" in the package "ExecutionClasses" of the repository of the Reconciliation+ toolkit representing a non empty set of messages that violate CR1. Note that, as we discussed in Section 5.1, this set is generated by the action of the context *Check_For_Messages_Without_Counterparts_Dispatched_By*.

As we discussed in Section 3.4, *Handle_Activated_By_Message2_With_No_Counterpart* can be applied to any of the messages in the set *DMWNC_mes2_set*. These alternative applications are generated as decisions and are proposed to the developer by the process enactment engine. For example, the options generated from *Handle_Activated_By_Message2_With_No_Counterpart* in the case of the overlap relation between the messages *7:actionPerformed(ActionPerformed)* and *11:actionPerformed(ActionEvent)* of the sequence diagrams shown in Figures 4 and 5 are shown in Figure 12. This figure shows a snapshot of the process enactment engine.



Process Enactment Engine - Process Model: reconcile_sequence_diagrams

Next decision :

Check_For_Messages_Without_Counterparts_Dispatched_By : actionPerformed(ActionEvent)<->actionPerformed(ActionEvent)_(a)
Handle_Activated_By_Message2_With_No_Counterpart : getData()
Handle_Activated_By_Message2_With_No_Counterpart : formulateQuery( )

Decisions made so far:

Reconcile_Sequence_Diagrams_In : Models
Find_Overlaps_Between_Sequence_Diagrams_In : Models
Reconcile_Sequence_Diagrams_In : Models
Reconcile_Overlapping_Messages_In : OptimalMorphism
Reconcile_Non_Identical_Activations_Of : actionPerformed(ActionEvent)<->actionPerformed(ActionEvent)_(a)
Check_For_Messages_Without_Counterparts_Dispatched_By : actionPerformed(ActionEvent)<->actionPerformed(ActionEvent)_(a
Reconcile_Non_Identical_Activations_Of : actionPerformed(ActionEvent)<->actionPerformed(ActionEvent)_(a)

Figure 12: Snapshot of process enactment engine

When one of these decisions is selected, the two alternative contexts for handling this kind of inconsistencies *Record_Message_Without_Counterpart* and *Add_Counterpart_In_Activation_Of_Message_1* also become options available for selection. This is because none of these contexts has any extra conditions in its situation.

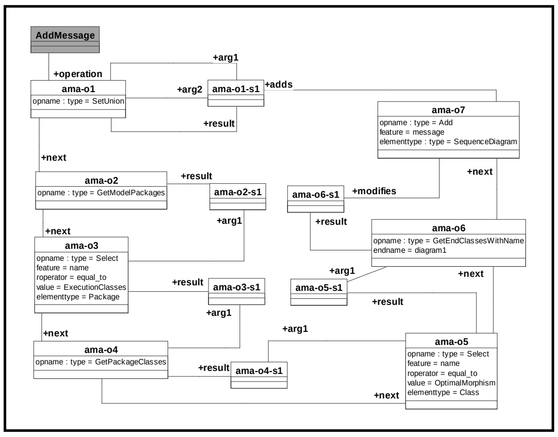


Figure 13: Specification of action *AddMessage*

Add_Counterpart_In_Activation_Of_Message_1 can be selected to create a copy of a message in the activation of a message m_j that does not have an overlapping counterpart in the activation of a message m_i that overlaps with it and add it to the activation of m_i.

Record_Message_Without_Counterpart can be activated to make a persistent record (i.e. a record saved after the end of the on-going process enactment trace) of the inconsistency. In the following, we explain the specification of the action of the former context that realises the former way of handling the inconsistency.

The specification of the action *AddMessage* of *Add_Counterpart_In_Activation_Of_Message_1* is shown in Figure 13. According to this specification, in executing *AddMessage*, the process enactment engine first locates the sequence diagram of message m_i in the overlap relation (m_i,m_j) (see operations *ama-o2* to *ama-o6)*, and then adds (see operation *ama-o7*) to the set of the messages of this diagram a copy of the message without the counterpart (that is the element of the set *ama-o1-s1*). *ama-o7* is an instance of the operation type *AddMessage* and is used to add copies of the message in its *arg1* set to each of the sequence diagrams which belong to its *modifies* set.

7. Related Work

A considerable body of research has been concerned with the problem of detecting inconsistencies in software models and documentation. This work has generated techniques for detecting inconsistencies in structured and text-based [1,3,4,12], object-oriented [2,6,18], state-based [7,8], and formal software models [5,17].

Some of the proposed techniques focus on object-oriented models. Glinz [6], for example, has developed a technique that checks behavioural software models expressed as statecharts for deadlocks, reachability and mutual exclusiveness of states. Cheung et al [2] have developed a technique that checks whether the sequence of the execution of operations that is implied by a UML statechart diagram is compliant with the sequence of the executions of operations implied by a UML sequence diagram. Zisman et al [18] have developed a consistency link generator which checks whether UML software models satisfy specific consistency rules. These rules are expressed in XML [2] and the consistency checking is performed using a tool developed using and XML development platform.

A critical survey of all the above techniques may be found in [15].

8. Conclusions and Further Work

In this paper, we presented Reconciliation+, a tool supported method developed to guide developers in reconciling models of the behaviour of software systems expressed as UML object interaction diagrams. The method provides a mechanism for detecting overlaps between messages in interaction models, a scheme for specifying consistency rules that overlapping messages must satisfy, and a scheme for specifying ways of handling violations of these rules. The rules and the ways of handling inconsistencies are specified as parts of a process model that is enacted by the method to drive the reconciliation of interaction diagrams. The method is extensible and configurable. This is because developers may add both consistency check and inconsistency handling contexts to its process model. The can also alter the situations of existing contexts.

Currently, we are evaluating the prototype developed for the method in an industrial case study. We are also studying properties of the schemes for expressing the consistency rules and the inconsistency handling contexts in the method. Future work will focus on expanding the built-in process models of the method to enable the capture of rationale for the selection of specific inconsistency handling actions.

Acknowledgements

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

References

1. Boehm, B, In, H, "Identifying Quality Requirements Conflicts", IEEE Software, (1996), 25-35.
2. Cheung K., Chow K., Cheung T., Consistency Analysis on Lifecycle Model and Interaction Model, Proc. of the Int. Conference on Object-Oriented Information Systems (OOIS '98), 1998, 427-441.
3. Easterbrook, S., "Handling Conflict between Domain Descriptions with Computer-Supported Negotiation", Knowledge Acquisition, Vol. 3, (1991), 255-289.
4. Emmerich W., Finkelstein, F. Montangero, C., Antonelli, S., Armitage, S., "Managing Standards Compliance". IEEE Transactions on Software Engineering, Vol. 25, No. 6, 1999.
5. Finkelstein, A., Gabbay, D., Hunter, A., Kramer, J., and Nuseibeh, B., "Inconsistency Handling In

Multi-Perspective Specifications", IEEE Transactions on Software Engineering, 20, 8, (1994), 569-578.
6. Glinz M., "An Integrated Formal Model of Scenarios Based on Statecharts" Proc. of the 5th European Software Engineering Conference, LNCS 989, Springer-Verlag, 1995, 254-271.
7. Heimdahl M.P.E, Leveson N., "Completeness and Consistency in Hierarchical State-Based Requirements", IEEE Transactions in Software Engineering, Vol. 22, No. 6, 1996, 363-377
8. Heitmeyer C., Jeffords R., Kiskis D., "Automated Consistency Checking Requirements Specifications", ACM Transactions on Software Engineering and Methodology, Vol 5, No 3, 1996, 231-261.
9. OMG, Unified Modeling Language Specification, V. 1.3a. Available from : ftp://ftp.omg.org/pub/docs/ad/99-06-08.pdf.
10. Papadimitriou C., Steiglitz K., Combinatorial Optimisation: Algorithms and Complexity, Prentice-Hall Inc., 1982.
11. Pohl K. "Process-Centred Requirements Engineering", Advanced Software Development Series, J. Kramer (ed), Research Studies Press Ltd., ISBN 0-86380-193-5, (1996), London
12. Robinson, W. and Fickas S. "Supporting Multi-Perspective Requirements Engineering", In Proc. of the IEEE Conference on Requirements Engineering, IEEE Computer Society Press, Los Alamitos, CA, (1994), 206-215.
13. Si-Said S, Rolland C, Grosz G, "MENTOR: A Computer Aided Requirements Engineering Environment", In Proc. of the 8th International Conference on Advanced Information Systems Engineering, Heraklion, Springer, (1996), 22-43
14. Spanoudakis G., and Finkelstein A. "Reconciling requirements: a method for managing interference, inconsistency and conflict", Annals of Software Engineering, Special Issue on Software Requirements Engineering, 3, (1997), 459-475.
15. Spanoudakis G., Zisman A. "Inconsistency Management in Software Engineering: Survey and Open Research Issues", Handbook of Software Engineering and Knowledge Engineering, (ed) Chang S. K., World Scientific Publishing Co., 2001 (to appear).
16. Kingman J., Taylor S., "Introduction to Measure and Probability", Cambridge Uni. Press, 1996.
17. Lamsweerde A., Darimont R., & Letier E., "Managing Conflicts in Goal-Driven Requirements Engineering", IEEE Transactions on Software Engineering, Special Issue on Managing Inconsistency in Software Development, 1999.
18. Zisman A., Emmerich W., Finkelstein A., "Using XML to Specify Consistency Rules for Distributed Documents", 10th Int. Workshop on Software Specification and Design, 2000.
19. Shafer G., "A Mathematical Theory of Evidence", Princeton University Press, 1976.
20. http://java.sun.com/j2se/1.3/docs/guide/awt/
21. Spanoudakis G.: "An Algorithm for Detecting Overlaps between Models of Object Interactions", Technical Report Series, TR-2000/03, ISSN 1364-4009, Department of Computing, City University, November 2000.
22. Spanoudakis G., Kim H.: "Evidential Management of Inconsistencies in Object Interaction Models", Technical Report Series, TR-2000/04, ISSN 1364-4009, Department of Computing, City University, December 2000.
23. Jacobson I., Christerson M., Jonsson P., and Overgaard G..: "Object Oriented Software Engineering: A Use Case Driven Approach". Addison-Wesley, Reading, Massachusetts, 1992