Supporting the Reconciliation of Models of Object Behaviour[1]

GEORGE SPANOUDAKIS AND HYOSEOB KIM[2]

Department of Computing,
City University,
Northampton Square, London EC1V 0HB, UK

E–mail: gespan@soi.city.ac.uk

Abstract: This paper presents Reconciliation+, a method which identifies overlaps between models of software systems behaviour expressed as UML object interaction diagrams (i.e., sequence and/or collaboration diagrams), checks whether the overlapping elements of these models satisfy specific consistency rules and, in cases where they violate these rules, guides software designers in handling the detected inconsistencies. The method detects overlaps between object interaction diagrams by using a probabilistic message matching algorithm that has been developed for this purpose. The guidance to software designers on when to check for inconsistencies and how to deal with them is delivered by enacting a built-in process model that specifies the consistency rules that can be checked against overlapping models and different ways of handling violations of these rules. Reconciliation+ is supported by a toolkit. It has also been evaluated in a case study. This case study has produced positive results which are discussed in the paper.

Keywords: consistency management, software design models, object interaction diagrams

1 Introduction

The specification of software system behaviour using multiple object interaction diagrams (i.e., sequence and/or collaboration diagrams) creates the potential of conflicting specifications of messages, objects and operations in these models. This is because different object interaction diagrams may, by virtue of the exchanges of messages that they specify and other elements in the specifications of these messages, imply different behaviours for the same objects and operations.

Consider, for example, an object model for a library system that includes the object interaction diagrams I_1 and I_2 of Figure 1 and the class diagram of Figure 2. The diagrams I_1 and I_2 specify interactions, which occur when the library system is used to search for items in the library either by keywords which refer to the author of an item (I_1) or by keywords which refer to the title of an item (I_2). The class diagram of Figure

[1] This article is an extended version of the article "Reconciliation of Object Interaction Models" that appeared in the proceedings of the 7th International Conference on Object Oriented Information Systems.

[2] This article reports on research that was carried out while the second author was affiliated with the Department of Computing of City University.

2 specifies the classes of the objects that participate in the interactions of I_1 and I_2. According to I_1 and I_2**,** the library system: (i) gets search keywords from a UI component (see messages `11:getText()` in I_1 and `8:getText()` in I_2); (ii) formulates a database query (see message `9:formulateQuery()` in I_2); and (iii) executes the query (see messages `12:executeQuery(SQLStatement)` in I_1 and `10:executeQuery(SQLSt)` in I_2).

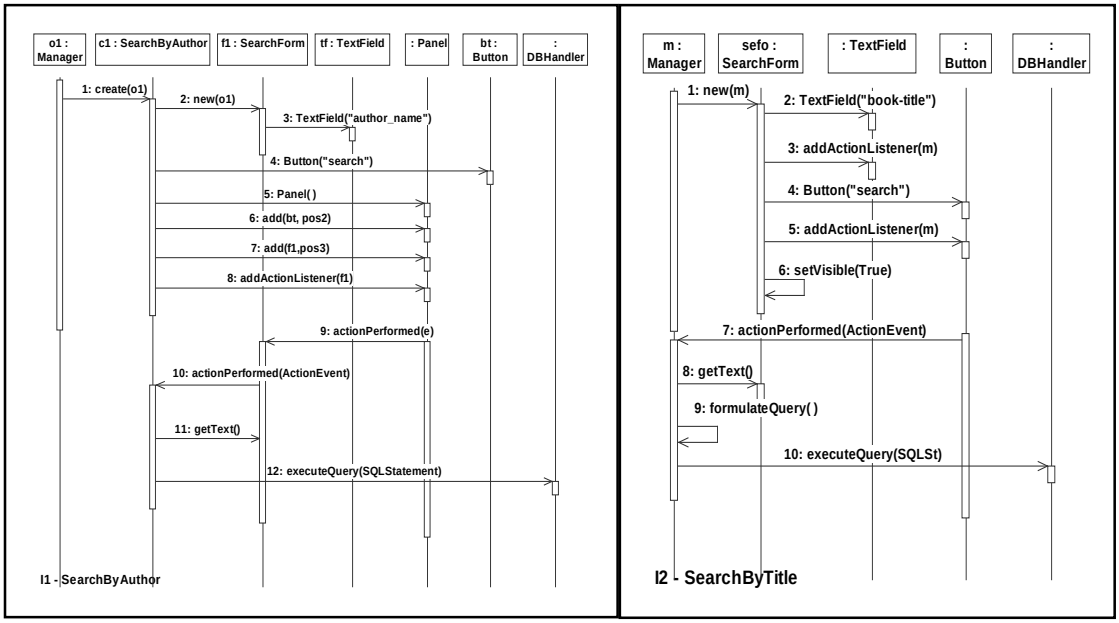

Figure 1: Object interaction diagrams `SearchByAuthor` (I_1) and `SearchByTitle` (I_2)

In this example, it is plausible to assume that the messages `10:actionPerformed(ActionEvent)` in I_1 and `7:actionPerformed(ActionEvent)` in I_2 overlap since (in the current state of the models) they both appear to invoke the operation `actionPerformed(e:ActionEvent)` of the class `DatabaseActionListener` in Figure 2. If, however, this assumption is correct then the specifications of the behaviour of the operation `actionPerformed(e:ActionEvent)` which are implied by I_1 and I_2 are conflicting. This is because according to I_2 (but not I_1) the execution of the operation `actionPerformed(e:ActionEvent)` leads to the dispatch of the message `9:formulateQuery()` and therefore the execution of the operation `formulateQuery()` of the class `Manager.`

Conflicts of this form need to be detected and reconciled in the design phase of a system development project in order to eliminate ambiguities that could be more expensive to resolve at the implementation phase.

In this paper, we describe a tool-supported method, called *Reconciliation+*, that we have developed to support software designers to detect and handle conflicts in behavioural specifications in models of object interactions which are expressed as sequence (or collaboration) diagrams in UML [9]. This method is a newly developed

extension of a method for managing modelling discrepancies in structural object models that is discussed in [15].

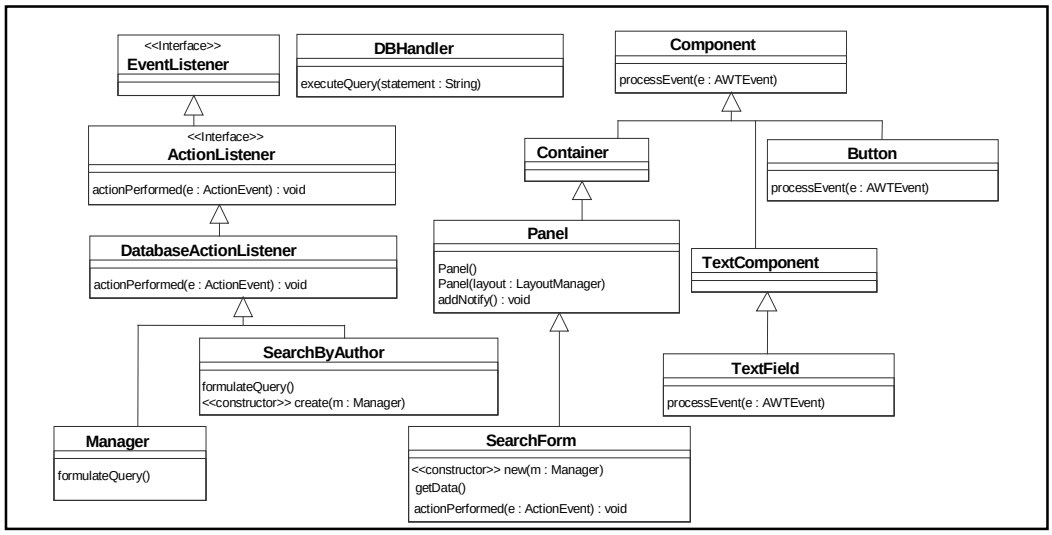

Figure 2: Model of classes of object interaction diagrams of Figure 1

Conflicts in Reconciliation+ are detected as violations of *consistency rules*. A consistency rule sets the conditions that should be satisfied by overlapping messages in different object interaction diagrams. In the above example of the library system, for instance, the conflict in the specifications of the behaviour of the operation `actionPerformed(e:ActionEvent)` implied by I_1 and I_2 could be detected as a violation of the following consistency rule:

CR1: If a message m_i overlaps with a message m_j then for every message m_k that is 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.

CR1 in this case would be violated by the message `9:formulateQuery()` that is dispatched by the message `7:actionPerformed(ActionEvent)` in I_2 and has no overlapping counterpart in the messages which are dispatched by the message `10:actionPerformed(ActionEvent)` in I_1 that overlaps with it.

Overlaps in Reconciliation+ are defined as relations between messages which are likely to signify the invocation of operations with the same implementation and are detected by a *probabilistic message matching algorithm*. This algorithm can detect overlaps between messages which are not identically modelled if certain parts of their specifications provide strong evidence to believe that the messages signify the invocation of operations with the same implementation. Thus, for example, it may detect an overlap relation between two messages m_i and m_j that have different signatures if they are sent to objects which are instances of the same class and dispatch the same messages (e.g. message `12:executeQuery(SQLStatement)` in I_1 and message `10:executeQuery(SQLSt)` in I_2).

Reconciliation+ incorporates a set of pre-defined consistency rules that should be satisfied by overlapping messages and alternative ways of handling violations of these rules. The method guides software designers in selecting which of these consistency rules to check and how to handle their violations. This guidance is delivered by enacting a built-in *process model* that specifies the consistency rules that can be checked against overlapping messages, and different ways of handling the violations of these rules. The method is extensible as software designers can extend its built-in process model by specifying new consistency rules, and new ways of handling violations of new or existing rules.

The rest of this paper is structured as follows. In Section 2, we describe the algorithm that Reconciliation+ deploys for detecting overlapping messages in different interaction diagrams. In Section 3, we describe the specification and enactment of the process model of the method that is used for guiding designers in reconciling object interaction diagrams. In Section 4, we describe the specification of consistency rules using the process model of the method and the mechanism for detecting violations of these rules. In Section 5, we describe the scheme for specifying and executing different ways of handling inconsistencies. In Section 6, we present an overview of the prototype toolkit we have developed to support Reconciliation+. In Section 7, we present the results of a case study that we have conducted to evaluate the method. Finally, in Section 8 we overview related work and, in Section 9, we summarise the method and outline ongoing work on it.

2 Detection of overlaps

2.1 Basic algorithmic formulation

Overlaps in Reconciliation+ are defined as relations between messages which are likely to signify the invocation of operations with the same implementation. The detection of these relations is formulated as an instance of the *weighted bipartite graph matching problem* [10]. More specifically, assuming a pair of interaction diagrams I_i and I_j, we construct a *weighted interaction overlap graph:* $IOG(I_i, I_j) = (V_i \cup V_j, E(V_i,V_j))$. This graph has two sets of disjoint vertices, V_i and V_j, which assuming that I_i has more messages than I_j, are defined as:

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

where

- *Messages(I_i)* is the set of messages of the interaction diagram I_i;
- *Messages(I_j)* is the set of messages of the interaction diagram I_j; and
- DV_k is a set of k special vertices representing dummy messages ($k = |Messages(I_i)| - |Messages(I_j)|$).

The set of the edges $E(V_i,V_j)$ includes all the possible edges between the messages of I_i and the messages of I_j, or formally:

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

An edge $(n_i, n_j, b_0(\neg ov(n_i,n_j)))$ in $E(V_i,V_j)$ designates the assumption that the messages represented by the nodes n_i and n_j overlap and is weighted by the measure $b_0(\neg ov(n_i,n_j))$. This measure is defined as the degree of belief in the falsity of the overlap assumption expressed by the edge, and is computed according to the following function:

$$b_0(\neg ov(n_i,n_j)) = \Sigma_{U \subseteq \{1,\ldots,6\}}(-1)^{|U|+1}\{\Pi_{u \in U}\, b_u(\neg ov(n_i,n_j))\}$$
$$\text{if } n_i \in \text{Messages}(I_i) \text{ and } n_j \in \text{Messages}(I_j) \qquad \text{(I)}$$
$$b_0(\neg ov(n_i,n_j)) = 1 \text{ if } n_j \in DV_k$$

The functions $b_1,\ldots,b_6$ used in (I) compute partial beliefs in the existence/absence of an overlap between two messages. The computation of these partial beliefs is based on heuristic criteria for assessing the equivalence of the *functional roles* and *implementations* of the operations invoked by the messages, and the *functional contexts* in which these operations are invoked. These belief functions and the criteria underpinning them are discussed in detail in Section 2.2.

After computing the beliefs b_0 for all the edges of $IOG(I_i, I_j)$, the most likely overlaps between the messages in I_i and I_j are detected in two steps. In the first step, 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[3]:

$$\Sigma_{(nu,\, nw,\, b0(\neg ov(nu,nw))) \in O(Vi,Vj)}\; b_0(\neg ov(n_u,n_w)) \qquad \text{(II)}$$

In the second step, $O(V_i,V_j)$ is restricted to include only the edges $(n_u,n_w,b_0(\neg ov(n_u,n_w))$ whose belief does not exceed a threshold value b_t, (i.e., edges for which $b_0(\neg ov(n_u,n_v)) \leq b_t$).

2.2 Criteria of overlap and partial belief functions

Beliefs in favour of, or against a hypothesis that two messages overlap are computed based on six criteria. These criteria indicate the equivalence of the *functional roles* and *implementations* of the operations which are invoked by the messages, and the equivalence of the *contexts* in which these operations are invoked.

2.2.1 Equivalence of functional roles of invoked operations

Beliefs in the equivalence of the functional roles of the operations which are invoked by two messages are computed using two criteria, namely the criterion of the *most generic overridden operation* and the criterion of the *operation stereotypes*.

The criterion of most generic overridden operation

Two operations are assumed to have equivalent functional roles if they override the same most general operation in an object model.

According to this criterion, the operations `processEvent(e: AWTEvent)` which are defined in the classes `Button` and `TextField` in Figure 2, for example, are

[3] The morphism $O(V_i,V_j)$ is selected using the *Hungarian method* [10].

considered to have equivalent functional roles. This is because both of them override the same most generic operation in the class model of the figure, namely the operation `processEvent(e:AWTEvent)` of `Component`. In this example, the classes `Button` and `TextField` override the latter operation that they inherit from `Component` in order to introduce the different functionality which is required for processing events of two different types: action events by `Button` and text events by `TextField` [20]. Despite of the differences in their exact functionality, however, the functional role of the operations `processEvent(AWTEvent e)` in `Button` and `TextField` is the same, that is to enable the instances of these classes to handle events related to them.

Note that, while in single inheritance hierarchies an operation always overrides a single most generic operation (which may trivially be itself[4]), in multiple inheritance graphs there is a potential for ambiguity. Such an ambiguity arises in cases where two superclasses of a class C, which are not directly or implicitly related by a generalisation relation themselves, define operations that have the same signature as an operation that is defined in C.

To cope with ambiguities of this form, we introduce the following function that measures the degree of belief in the existence (absence) of an overlap relation between two messages based on the criterion of the most generic overridden operations:

Definition 1: The degree of belief in the existence (absence) of an overlap relation between two messages m_i and m_j based on the criterion of the most generic operations overridden by the operations invoked by m_i and m_j is computed according to the function:

$$b_1(\neg ov(m_i,m_j)) = \alpha_1 \times d_1(m_i,m_j) \quad \text{and} \quad b_1(ov(m_i,m_j)) = 0$$

where

- $d_1(m_i,m_j) = (|Os(o_i)-Os(o_j)|+|Os(o_j)-Os(o_i)|)/|Os(o_i)\cup Os(o_j)|$
  if $Os(o_i) \neq \emptyset$ and $Os(o_j) \neq \emptyset$
- $d_1(m_i,m_j)=1$
  if $Os(o_i) = \emptyset$ or $Os(o_j) = \emptyset$
- o_i is the operation invoked by m_i and o_j is the operation invoked by m_j
- $Os(o_i)$ ($Os(o_j)$) is a set of operations which are defined in the superclasses of the class that defines o_i (o_j), have the same signature with it, and do not override any other operation with the same signature.
- α_1 is the expected ratio of messages that invoke operations which do not override the same most generic operation and do not overlap ($0 \leq \alpha_1 \leq 1$)

Examples

The belief produced by b_1 in the absence of an overlap relation between the messages `7:actionPerformed(ActionEvent)` and `10:actionPerformed(ActionEvent))` in the interaction diagrams I_1 and I_2 of Figure 1 is 0. This is because, according to the class model of Figure 2, both these messages invoke the same operation, that is `DatabaseActionListener.actionPerformed(e:ActionEvent)`, and

[4] As in the case of the operation `formulateQuery()` of the class `Manager` in Figure 2.

therefore the same most general overridden operation (i.e., `ActionListener.actionPerformed(e:ActionEvent)`). Thus, the d_1 distance between these messages is 0.

Note, however, that the belief in the absence of an overlap relation between the messages `9:actionPerformed(e)` in I_1 and `7:actionPerformed(ActionEvent)` in I_2 that is generated by b_1 is 0.4 (assuming that $\alpha_1 = 0.4$). This is because the former message invokes the operation `SearchForm.actionPerformed(e: ActionEvent)` and the latter message invokes the operation `DatabaseActionListener.actionPerformed(e: ActionEvent)` in Figure 2. And as these two operations override different most general operations in the relevant class model (i.e., the operation `SearchForm.actionPerformed(e: ActionEvent)` and the operation `ActionListener.actionPerformed(e:ActionEvent)`, respectively), the distance between the above messages is 1. As a result, b_1 assumes that the functional roles of `SearchForm.actionPerformed(e: ActionEvent)` and `DatabaseActionListener.actionPerformed(e: ActionEvent)` are different and, therefore, it generates the maximum possible belief against the existence of an overlap relation between `9:actionPerformed(e)` and `7:actionPerformed(ActionEvent)`.

The criterion of operation stereotypes

The second criterion for assessing the equivalence of the functional roles of two operations is based on operation stereotypes. According to this criterion, the functional roles of two operations are considered equivalent if the operations have the same stereotype(s). This criterion is used since, in UML, operation stereotypes are used to designate groups of functionally similar operations. The belief function associated with this criterion is defined as follows:

Definition 2: The degree of belief in the existence (absence) of an overlap relation between two messages m_i and m_j based on the criterion of the stereotypes of the operations that m_i and m_j invoke is computed according to the function:

$$b_4(\neg ov(m_i,m_j)) = \alpha_4 \times d_4(m_i,m_j) \quad \text{and} \quad b_4(ov(m_i,m_j)) = 0$$

where

- $d_4(m_i,m_j) = (|St(o_i) - St(o_j)| + |St(o_j) - St(o_i)|)/|St(o_i) \cup St(o_j)|$ if $St(o_i) \neq \varnothing$ and $St(o_j) \neq \varnothing$
- $d_4(m_i,m_j) = 1$ if $St(o_i) = \varnothing$ or $St(o_j) = \varnothing$
- o_i is the operation invoked by m_i and o_j is the operation invoked by m_j
- $St(o_i)$ and $St(o_j)$ are the sets of the stereotypes of o_i and o_j
- α_4 is the expected ratio of operations with different stereotypes which do not overlap ($0 \leq \alpha_4 \leq 1$)

The functional form of b_4 covers cases where an operation may belong to different stereotype groups. In these cases, d_4 measures the likelihood of the operations invoked by two messages not having a common stereotype.

Examples
The d_4 distance between the messages `1:create(o1)` in I_1 and `1:new(m)` in I_2 is 0. This is because the operations `SearchByAuthor.create(o:Manager)` and `SearchForm.new(o:SearchByAuthor)` which are invoked by these messages, are both stereotyped as *constructor-operations* in the class diagram of Figure 2. Thus, b_4 is also 0 in this case. For any other pair of messages in Figure 1, however, since the stereotypes of the operations invoked by them are not defined (see Figure 2) the d_4 distance is 1 and thus it generates a b_4 belief equal to α_4.

2.2.2 Equivalence of the functional contexts of operations

The assessment of the equivalence of the functional contexts of two messages is based on three criteria. The first of these criteria is whether the messages are sent by instances of the same class or, equivalently in terms of UML, they have the same *senders*. The second criterion is whether the messages are received by instances of the same class or, in terms of UML, they have the same *receivers*. The third criterion is whether the messages are dispatched by the same message or, in terms of UML, have the same *activator*.

The criterion of message senders

According to this criterion, the functional contexts in which two messages are dispatched are considered to be different if the messages are sent by objects which are instances of different classes. The belief function that is associated with this criterion is defined as:

Definition 3: The degree of belief in the existence (absence) of an overlap relation between two messages m_i and m_j based on the criterion of message senders is computed according to the function:

$$b_2(\neg ov(m_i,m_j)) = \alpha_2 \times d_2(s_i, s_j) \quad \text{and} \quad b_2(ov(m_i,m_j)) = 0$$

where

- s_i and s_j are the classes of the objects that send m_i and m_j, respectively
- $d_2(s_i, s_j)$ is a function measuring the generalisation distance between s_i and s_j defined as:
  - $d_2(s_i, s_j) = \sum_{x \in NCSij} SD(x)^{-1} / \sum_{y \in ASSij} SD(y)^{-1}$ if s_i and s_i are specified
    $d_2(s_i, s_j) = 1$ if s_i or s_i is not specified
  - $NCS_{ij} = ((s_i.Isa^* \cup \{s_i\}) - (s_j.Isa^* \cup \{s_j\})) \cup ((s_j.Isa^* \cup \{s_j\}) - (s_i.Isa^* \cup \{s_i\}))$
  - $ASS_{ij} = ((s_i.Isa^* \cup \{s_i\}) \cup (s_j.Isa^* \cup \{s_j\}))$
  - SD(x) is the length of the longest path connecting a class x with its most general superclass, called *specialisation depth* of x
  - $s_i.Isa^*$ ($s_j.Isa^*$) is the transitive closure of the superclasses of class s_i(s_j)
- α_2 is the expected ratio of non overlapping messages with different senders ($0 \leq \alpha_2 \leq 1$)

The assumption underpinning the definition of the belief function b_2 is that the identity of class names is not necessarily an accurate indicator of the identity of classes, especially if classes are specified in two independently constructed models of the same

system. Thus, b_2 computes a belief in class identity based not only on the classes themselves but also on their superclasses. According to its definition, the more the non-common superclasses of two classes c_i and c_j the stronger the belief that c_i and c_j are not identical.

Each of the non-common superclasses of two classes c_i and c_j produces evidence of different strength for the assumption that c_i and c_j are not identical. The strength of this evidence is measured as the inverse of the *specialisation depth* (i.e., the length of the longest path connecting a class with its most general superclasses) of a superclass in the generalisation graph of the model(s). According to this measure, non common superclasses which appear in relatively low levels of generalisation graphs and, by virtue of their position, introduce fine-grain specialisations of more general classes provide weaker evidence than classes which appear in higher positions [14].

Examples

Given the class model of Figure 2, the d_2 distance between the classes `Manager` and `SearchByAuthor` that send the messages `2:new(o1)` and `1:new(m)` in the interaction diagrams I_1 and I_2 is 0.21. Thus, assuming that $\alpha_2 = 0.1$, the belief in the absence of an overlap relation between these messages that b_2 generates is 0.021.

The criterion of message receivers

According to this criterion, the functional contexts of two messages are not considered to be equivalent if the messages are received by objects which are instances of different classes or, equivalently in terms of UML, if they have different *receivers*. The belief function that is associated with this criterion is defined as:

Definition 4: The degree of belief in the existence (absence) of an overlap relation between two messages m_i and m_j based on the criterion of message receivers is computed according to the function:

$$b_3(\neg ov(m_i,m_j)) = \alpha_3 \times d_2(r_i, r_j) \quad \text{and} \quad b_3(ov(m_i,m_j)) = 0$$

where

- r_i and r_j are the classes of the objects that receive m_i and m_j, respectively
- d_2 is as defined in Definition 3
- α_3 is the expected ratio of messages with different receivers which do not overlap ($0 \leq \alpha_3 \leq 1$)

The rational for using the above function for measuring belief in the absence/existence of an overlap relation between the receivers of two messages is the same as that used in the case of the function b_2.

The criterion of message activators

The third criterion for calculating belief in the equivalence of the functional contexts of two messages m_i and m_j is whether these messages are sent by messages which overlap themselves (these messages are called *activators* in UML). The belief function associated with this criterion of message activators is defined as follows:

Definition 5: The degree of belief in the existence (absence) of an overlap relation between two messages m_i and m_j based on the criterion of their message activators is computed according to the function:

$b_5(\neg ov(m_i,m_j)) = \alpha_5 \times \Sigma_{U \subseteq \{1,\ldots,4\}}(-1)^{|U|+1}\{\Pi_{u \in U} b_u(\neg ov(m_k,m_l))\}$ if $m_k \neq nil$ & $m_l \neq nil$

$b_5(\neg ov(m_i,m_j)) = \alpha_5 \times 1$ if $m_k = nil$ or $m_l = nil$

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

where

- m_k and m_l are the activators of the messages m_i and m_j, respectively
- $b_1,\ldots, b_4$ are the belief functions defined definitions 1-4
- α_5 is the expected ratio of messages with different activators which do not overlap ($0 \leq \alpha_5 \leq 1$)

According to Definition 5, the criteria used for computing beliefs in the absence of an overlap relation between the activators of two messages include the criteria of most generic overridden operation, operation stereotypes, message senders and message receivers but excludes the criteria of message activators and message activations (see Section 2.2.3). The reason for not using the criterion of message activators in the computation of b_5 beliefs is to avoid recursive computations in the transitive closure of the activators of the messages. Similarly, the reason for excluding the criterion of message activations when calculating beliefs in the existence and absence of an overlap relation between the activators of two messages m_i and m_j is that their inclusion would lead to a non terminating recursion in the computations, since it would require the computation of the belief in the absence of an overlap between m_i and m_j again (see definition of belief function b_6 below).

Examples

Assuming that the parameters $\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6$ take the values .4, .1, .2, .1, .2, and .4, respectively, the b_5 belief in the absence of an overlap relation between the messages `3:TextField("author_name")` in I_1 and `2: TextField("book_title")` in I_2 is 0.02. This is because there is weak evidence from the model against the existence of an overlap relation between the activators of these messages (these are the messages `2:new(o1)` and `1:new(m)`, respectively) . More specifically, in this case we have that:

$\Sigma_{U \subseteq \{1,\ldots,4\}}(-1)^{|U|+1}\{\Pi_{u \in U} b_u(\neg ov($`2:new(o1), 1:new(m)`$))\} = 0.1$

2.2.3 Equivalence of the implementations of operations

The criterion of message activations

The final criterion for computing a partial belief in the existence/absence of an overlap relation between two messages m_i and m_j is whether the messages that m_i and m_j dispatch messages which overlap themselves or, in terms of UML, they have the same activations. This criterion is used since the dispatch of non-overlapping messages by two messages m_i and m_j implies that the operations which are invoked by m_i and m_j have different implementations. The belief function associated with this criterion is defined as follows:

Definition 6: The degree of belief in the existence (absence) of an overlap relation between two messages m_i and m_j based on the criterion of message activations is computed according to the function:

$$b_6(\neg ov(m_i,m_j)) = \alpha_6 \times d_6(m_i,m_j) \quad \text{and} \quad b_6(ov(m_i,m_j)) = 0$$

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 $A_i \neq \varnothing$ and $A_j \neq \varnothing$
  $d_6(m_i,m_j) = 1$ if $A_i = \varnothing$ or $A_j = \varnothing$
- A_i and A_j are the sets of messages which are dispatched by m_i and m_j, respectively.
- *Morphisms(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 *onto* morphisms from the messages in A_i to the messages in A_j if $|A_j| < |A_i|$.
- $b_0(\neg ov(m_u,m_v))$ is computed as defined by formula (I).
- α_6 is the expected ratio of messages with different activations which do not overlap ($0 \leq \alpha_6 \leq 1$)

According to Definition 6, the computation of b_6 beliefs for two messages m_i and m_j leads recursively to the identification of the most likely overlaps between all the messages which are directly or transitively dispatched by them (this is because d_6 is defined in terms of b_0 which, according to formula (I), is defined in terms of b_6). This recursive computation is terminated whenever the messages under comparison in the transitive closures of the activations of m_i and m_j dispatch no further messages. In cases where any of the messages under comparison dispatches no messages, d_6 returns a belief equal to one. This belief reflects the hypothesis that in the absence of any evidence about the additional operations that two operations o_1 and o_2 invoke o_1 and o_2 can be assumed to have different implementations.

Examples

The b_6 beliefs in the absence of an overlap relation between the messages `1:new(m)` and `2:new(o1)` and between the messages `7:actionPerformed(ActionEvent)` and `10:actionPerformed(ActionEvent)` in the interaction diagrams of Figure 1 are .88 and .61, respectively. These beliefs are computed assuming that the parameters $\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6$ take the values .4, .1, .2, .1, .2, and .4, respectively.

The above b_6 beliefs reflect the facts that

(a) The activations of `1:new(m)` and `2:new(o1)` have only one pair of overlapping messages, that is the pair formed by the messages `2:TextField("book-title")` and `3:TextField("author-name")`, and four messages without overlapping counterparts, namely the messages `3:addActionListener(m)`, `4:Button("search")`, `5:addActionListener(m)`, and `6:setVisible(True)` in the activation of `1:new(m)`.

(b) The activations of `7:actionPerformed(ActionEvent)` and `10:actionPerformed(ActionEvent)` have two pairs of overlapping messages (these are pairs formed by the messages `8:getText()` and

`11:getText()`, and the messages `10:executeQuery(SQLSt)` and `12:executeQuery(SQLStatement)`), and one message without an overlapping counterpart, namely the message `9:formulateQuery()`.

2.3 Example of detecting an overlap morphism

The algorithm specified in Sections 2.1 and 2.2 detects the following overlapping messages in the interaction diagrams of Figure 1:

(i) message `2:new(o1)` in I_1 and message `1:new(m)` in I_2
(ii) message `3:TextField("author_name")` in I_1 and message `2:TextField("book-title")` in I_2
(iii) message `4:Button("search")` in I_1 and message `4:Button("search")` in I_2
(iv) message `10:actionPerformed(ActionEvent)` in I_1 and message `7:actionPerformed(ActionEvent)` in I_2
(v) message `11:getText()` in I_1 and message `8:getText()` in I_2
(vi) message `12:executeQuery(SQLStatement)` in I_1 and message `10:executeQuery(SQLSt)`in I_2

Messages	**Beliefs**						
	b_1	b_2	b_3	b_4	b_5	b_6	b_o
(2, 1)	0	0.214	0	0	1	0.928	0.507
(3, 2)	0	1	0	0	0.52	1	0.516
(4, 4)	0	1	0	0	0.52	1	0.516
(10, 7)	0	0.612	0.214	0	0.479	0.616	0.388
(11, 8)	0	0.214	0	0	0.1	1	0.424
(12, 10)	0	0.214	0	0	0.1	1	0.424

Table 1: Beliefs against the overlaps detected between the messages of I_1 and I_2

The beliefs in the absence of overlaps between the above messages are shown in Table 1 and were computed after setting the parameters α_1, α_2, α_3, α_4, α_5, α_6 and b_t to 0.4, 0.1, 0.2, 0.1, 0.2, 0.4, and 0.65, respectively. The rows of the table designate the above pairs of messages by using the numbers that indicate the order of their dispatch in each diagram. For example, the overlapping messages `11:getText()`and `8:getText()`are signified in Table 1 as the pair (11, 8).

The values selected for the parameters α_1–α_6 in this example were empirically determined after considering models that incorporated generalisation graphs and object interactions specified at varying degrees of completeness and elaboration. Reconciliation+ assumes that designers should decide which are the appropriate values for the parameters α_1–α_6 and b_t. These decisions should be based on an assessment of how accurate is as an indicator of overlaps each of the criteria that underpin the belief functions b_1–b_6. This assessment can be formed based on the form and level of elaboration of different parts of the involved design model(s). If a model uses no

stereotypes, for instance, α_4 should be set to 0. Similarly, for models that do not incorporate elaborate class generalisation graphs α_2 should be set to a low value.

2.4 Properties of the belief functions

As proved in [21], the functions b_1–b_6 are all distance metrics and satisfy the axioms of *Dempster-Shafer basic probability assignments* [19]. 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 that is jointly committed to $\neg ov(m_i,m_j)$ by b_1–b_6. b_0 is also a distance metric (see [21] for a proof). These characteristics of b_0 guarantee the following intuitive properties for its outputs:

- for any three messages m_i, m_j and m_k we have, due to the *triangularity* of distance metrics, that: $b_0(\neg ov(m_i,m_k)) \leq b_0(\neg ov(m_i,m_j)) + b_0(\neg ov\ (m_j,m_k))$
- for any two messages m_i and m_j we have, due to the *symmetry* of distance metrics, that: $b_0(\neg ov(m_i,m_j)) = b_0(\neg ov(m_j,m_i))$
- for any two messages m_i and m_j we have, due to axiomatic foundation of Dempster-Shafer basic probability assignments [19], that: $b_0(\neg ov(m_i,m_j) \wedge ov(m_j,m_i)) = 0$

3 The Reconciliation+ process: specification and enactment

As we discussed in Section 1, Reconciliation+ guides software designers through the activity of reconciling their models by enacting a built-in process model. This model specifies consistency rules that may be checked against overlapping messages and alternative ways of handling violations of these rules. In this section, we introduce the scheme that is used by the method to specify this process model, and the mechanism that is used to enact it.

3.1 A UML profile for specifying reconciliation processes

The process of Reconciliation+ is specified as a graph of *contexts* following a decision-oriented approach to software process modelling [11]. A context represents a *decision* that may be taken in a given *situation*. This situation is specified as a condition over the state of the software models which are being manipulated by the process (i.e., the interaction diagrams which are being reconciled in the case of Reconciliation+). Contexts are distinguished into:

(1) *executable contexts* – these are contexts which represent decisions to take *actions* that change the state of the software model;
(2) *plan contexts* – these are contexts which represent decisions that can be realised by a set of sub-decisions which must be made in a specific order; and

(3) *choice contexts* – these are contexts which represent decisions that may be realised by two or more other alternative decisions.

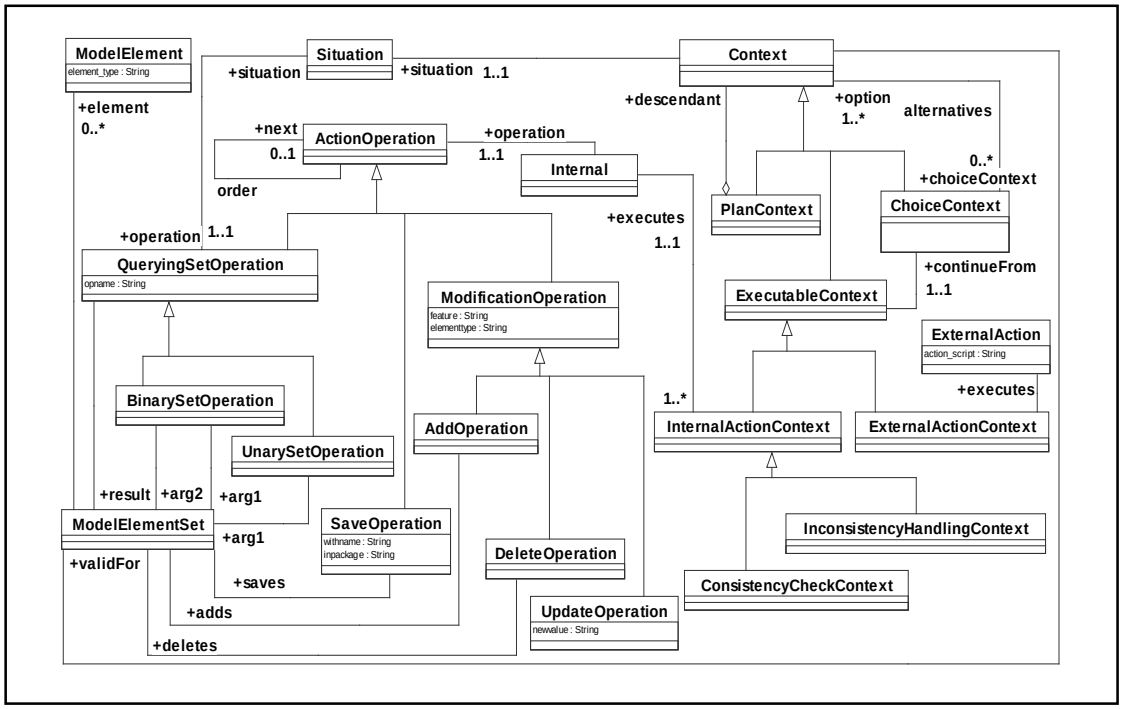

Figure 3: A UML profile for specifying Reconciliation+ processes

The process models that can be formulated using this process modelling approach are essentially AND/OR graphs of decisions which are guarded by specific conditions. Figure 3 presents a *UML profile* that we have defined for specifying model reconciliation processes in UML using the contextual process modelling approach described above[5].

Profiles in UML are user-defined extensions of the meta-model of the language which are introduced to support the construction of models for specific purposes and domains. Typically, a UML profile consists of: (i) a set of *stereotypes* that define additional kinds of modelling elements which are required to construct models for the particular domain, (ii) *tag* definitions that introduce additional meta-attributes for the model elements that may be classified under the stereotypes of the profile, and (iii) *constraints* which are associated with the stereotypes of the profile and apply to all the elements classified under them [9].

The Reconciliation+ process specification profile that we have developed is represented by the UML class diagram shown in Figure 3. This diagram has been constructed according to the following representation conventions:

[5] The introduction of a UML profile for the specification of the reconciliation processes of our method was motivated by the need to have both software models and the processes that reconcile them expressed in a single representational framework as this would facilitate the implementation of tool support for the method (see Section 6).

(i) Classes represent the stereotypes of the profile. The class `ChoiceContext`, for example, designates the stereotype that represents choice contexts in the process modelling approach outlined above.

(ii) A named association end in Figure 3 designates a tag defined for the stereotype that is represented by the class that is attached to the opposite end of the relevant association. For example, `situation` is a tag defined for the stereotype `Context`. The type of this tag is the stereotype `Situation` and its multiplicity is 1..1.

(iii) An attribute in Figure 3 designates a tag defined for the stereotype that is represented by the class incorporating it. Attributes represent tags whose type is a data type. For example, `feature` is a tag defined for the stereotype `ModificationOperation` whose type is `String`.

(iv) A generalisation relation in Figure 3 designates a generalisation relation between the stereotypes represented by the classes that it connects. For example, `ChoiceContext` is a special kind of `Context`.

As shown in Figure 3, our profile, includes stereotypes that represent the basic constructs for specifying reconciliation processes, including contexts, situations and actions.

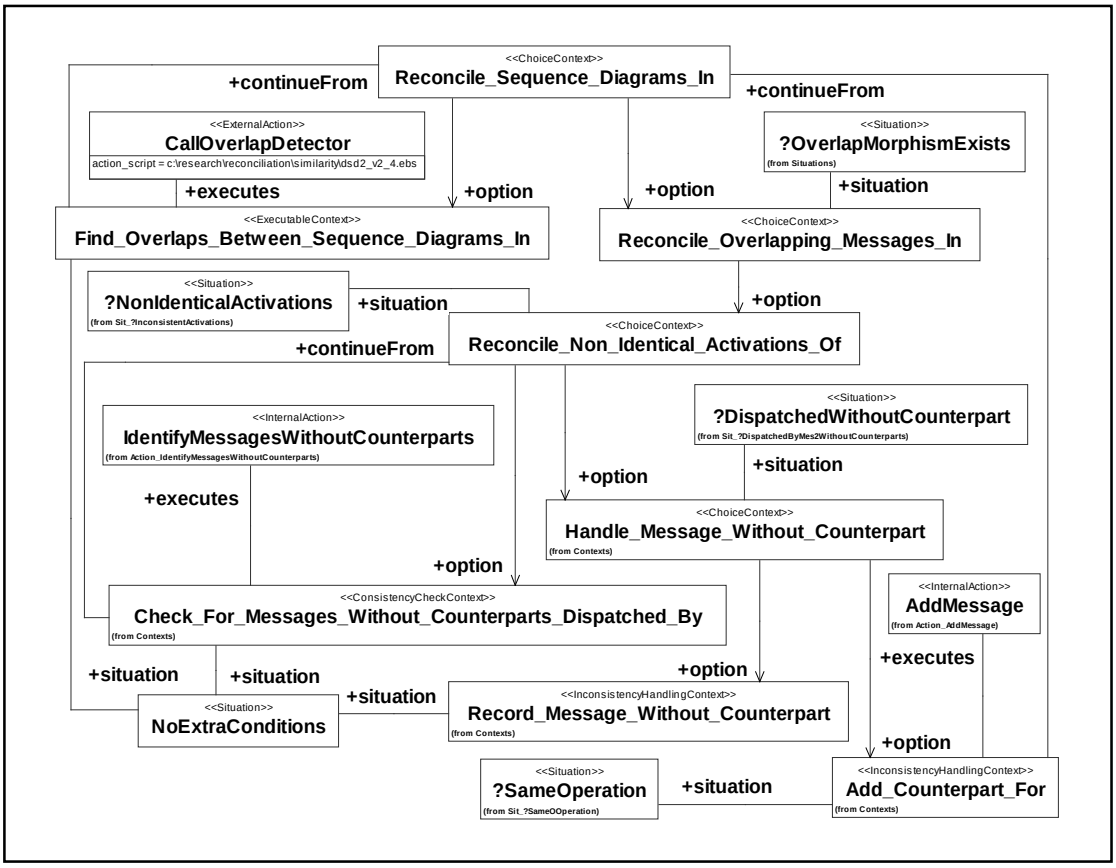

Figure 4: Part of the Reconciliation+ process model

A part of the Reconciliation+ process model that is specified according to the above profile is shown in Figure 4. This part includes, for example, the choice context `Reconcile_Overlapping_Messages_In` which represents the decision to start the process of reconciling overlapping messages. The situation of this context, `?OverlapMorphismExists`, checks whether an overlap morphism between the interaction diagrams to be reconciled has been identified. Thus, the context `Reconcile_Overlapping_Messages_In` can be selected only if the overlaps between two interaction models have been identified.

3.2 Specification of situations

In Reconciliation+, a situation is specified as a *query* defined by an ordered sequence of *querying set operations*. A querying set operation is either a `UnarySetOperation` or a `BinarySetOperation`.

Unary set operations are used to retrieve elements of a UML model which are associated with a given element *e* of it via any of the different kinds of associations or attributes defined for the type of *e*. A unary set operation is associated with two sets: the `arg1` set and the `result` set. The former set includes all the elements of a UML model that the operation should be applied to. The latter set is used to store the union of the elements which are retrieved for each of the elements of `arg1` following the application of the operation. An example of a unary set operation is the operation `imwnc-o2` in Figure 6. This operation is stereotyped as a `GetActivation` operation, that is a unary set operation which retrieves the messages which are directly dispatched by the messages which belong to its `arg1` set. Additional examples of unary set operations are given in Section 4.

Binary set operations are associated with three sets: the `arg1`, `arg2` and `result` set. There are three different types of such operations in our process specification profile for computing the union, intersection and set-difference of given sets. Similarly to unary operations, the result of a binary set operation is saved in its `result` set. An example of a binary set operation is the operation `imwnc-o7` in Figure 6. This operation is stereotyped as a `SetDifference` operation and, therefore, it computes the set difference of the sets of messages `imwnc-o6-s1` and `imwnc-o4-s1` (i.e., `imwnc-o6-s1` − `imwnc-o4-s1`) and inserts the elements of this set difference in `imwnc-o7-s1`.

The operations that define the query of a situation *s* are ordered through the association `order`[6] (see Figure 3) and may take as an argument any of the sets which are generated by operations preceding them in *s*. A situation is satisfied if the `result` set of the last of its operations is not empty. Examples of specifications of situations are given in Section 5.

[6] The association end `next` of this association denotes the next operation in a local operation sequence.

3.3 Specification of actions

In our process specification profile, each executable context must have an action that can be either an external or internal action (see the action stereotypes `ExternalAction` and `InternalAction` in Figure 3).

External actions are used to specify the invocation of external tools during the enactment of a Reconciliation+ process. An example of an external action is the action `CallOverlapDetector` in the process model of Figure 4. This action is associated with the context `Find_Overlaps_Between_Sequence_Diagrams_In` and is executed when this context is selected. The execution of this action invokes the overlap detection tool of Reconciliation+ (a pointer to the executable file of this tool is specified as the value of the attribute `action_script` of the action as shown in Figure 4).

Internal actions are used to specify consistency rules and ways of handling inconsistencies. These actions are specified as sequences of *action operations*. An action operation may be a *querying*, *model modification*, or *save* operation. Querying operations are the same as those used in the specification of situations. Modification operations are operations which are used to modify the state of the software models being manipulated by the process. Save operations are used to store the results of querying operations in the trace of the enactment of a process model so as to make them available in subsequent stages of this enactment. Action operations are ordered in internal actions and executed similarly to sequences of querying operations in situations. Examples of specifications of internal actions and the different types of operations that may be used in them are given in Sections 4 and 5.

3.4 Process enactment

The Reconciliation+ process model is enacted by an engine which functions as a model interpreter [13]. The algorithm underpinning the operation of this engine is specified in Figure 5. According to this algorithm, the enactment of a process model starts from the *root context* of the process model (a process model must have a single root context that should be a choice context). The situation of the root context of a process model (and any other context that is encountered as the enactment engine traverses it) is evaluated by executing the set querying operations that define it. 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 considered to have been satisfied. In this case, the enactment engine generates different possible *decisions* from the context, one for each of the elements in the situation set.

More specifically, a *decision* is defined as a pair:

$$\langle context_i, situation\_set\_element_j \rangle$$

where

- $context_i$ is the context whose situation is satisfied, and

- `situation_set_element`$_j$ is an element of the situation set of `context`$_i$.

```
Algorithm:   EnactProcess (CurContext, Argument, Trace)
In:          CurContext
             Argument      // model element that the input Context was instantiated for
In/Out:      Trace         // list of pairs <context, argument>

  If CurContext ≠ nil Then
    EvaluateSituation(CurContext.Situation, Argument, ResultSet);
    For each model element e in ResultSet Do
        Options = Options ∪ {< CurContext, e >}
    End For
    Options = Options ∪ {< Tactical Guidance, nil >} ∪ {< Abort Process, nil >};
    // Options is a set of pairs of the form <context, argument>
    If Options is not empty Then
      SelectedOption = User's selection from available options;
      If SelectedOption.context = Tactical Guidance Then
        LastChoice =  last <context, argument> pair in Trace before CurContext whose context
                      is a choice context; // if no such context exists LastChoice becomes <nil, nil>
        NextContext = LastChoice.context;
        NextArgument = LastChoice.argument;
        EnactProcess(NextContext, NextArgument, Trace);
      Else If SelectedOption.context = Abort Process Then
        NextContext = nil;
        NextArgument = nil;
        EnactProcess(NextContext, NextArgument, Trace);
      Else
        insert(SelectedOption.context, SelectedOption.argument, Trace);
        If SelectedOption.context is an executable context      Then
          Execute(SelectedOption.context);
          NextContext = SelectedOption.context.continueFrom;
          NextContextAncestor =     last <context, argument> pair in Trace before NextContext;
          NextArgument = NextContextAncestor.argument;
          EnactProcess(NextContext, NextArgument, Trace);
        Else // SelectedOption.context is a choice context
          Alternatives = CurContext.alternatives
          For each context c in Alternatives Do
            EvaluateSituation(c, SelectedOption.argument, ResultSet);
            For each model element e in ResultSet Do
              Options = Options ∪ {< c, e >}
            End For
          End For
          Options = Options ∪ {<TacticalGuidance,nil>} ∪ {<AbortProcess,nil>};
          SelectedOption = User's selection from available options
          EnactProcess(SelectedOption.context, SelectedOption.argument, Trace);
        End If
      End If
    End If
End If
```

Figure 5: Process enactment algorithm

The situation `?NonIdenticalActivations` in the process model of Figure 4, for example, retrieves all the pairs of overlapping messages of two interaction diagrams that have non identical activations. In the case of the interaction diagrams I_1 and I_2, these pairs of messages are:
(`2:new(o1)`, `1:new(m)`)
(`10:actionPerformed(ActionEvent)`, `7:actionPerformed(ActionEvent)`)

Thus, the possible decisions that may be generated from the context `Reconcile_Non_Identical_Activations_Of` of this situation when reconciling I_1 and I_2 are:

(1) <Reconcile_Non_Identical_Activations_Of, (2:new(o1), 1:new(m))>
(2) <Reconcile_Non_Identical_Activations_Of, (10:actionPerformed(ActionEvent), 7:actionPerformed(ActionEvent)) >

A designer may select one of the different possible decisions which are generated from a context, ask for tactical guidance or terminate the process. If a decision <$context_i$, $situation\_set\_element_j$> is selected, it is recorded in the trace of the enacted process model and 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 operation 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 designer to make a new selection.
- If $context_i$ is an external action context, the enactment engine executes the file specified by the attribute action_script of it and continues the enactment of the process model from the context associated with $context_i$ via the association end continueFrom (see Figure 3).
- 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.[7]

In cases where the designer 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 designer may also abort the execution of the process model at any point.

In the following, we describe how the process specification profile of our method can be used to specify consistency rules and actions to handle their violations.

4 Detection of inconsistencies

In Reconciliation+, a consistency rule is defined as an internal action of a *consistency check context* (i.e., a special kind of executable contexts as shown in Figure 3). This action is essentially a query which retrieves the model elements that violate the conditions required by the rule. Thus, the specification of consistency rules is procedural. The consistency check context which incorporates the internal action that defines a consistency rule represents the decision to check the rule. It also specifies the conditions under which this decision may be made. These conditions are specified by the *situation* of the context.

[7] Plan contexts are not used in the current process model of Reconciliation+ and therefore the description of their enactment is beyond the scope of this paper.

Furthermore, the internal actions of consistency check contexts are restricted not to include any modification operations. This restriction guarantees that the execution of a consistency rule will not modify the contents of the underlying models. Also, the last operation of such actions must be a *save operation* that records the model elements which violate the rules (see stereotype `SaveOperation` in Figure 3) in order to make them available to subsequent stages of the enactment of the reconciliation process.

Figure 6 shows the internal action `IdentifyMessagesWithoutCounterparts` that defines the consistency rule CR1. As discussed in Section 1, CR1 requires that if a message m_i overlaps with a message m_j then for each message x activated by m_i there must be a message y activated by m_j that overlaps with x and vice versa. The specification of this internal action assumes that message overlap relations are represented by overlap objects in the trace of the Reconcilication+ process which point to the overlapping messages and store the beliefs in their overlap (see the object `10:actionPerformed(ActionEvent)`↔`7:actionPerformed(ActionEvent)` in Figure 7).

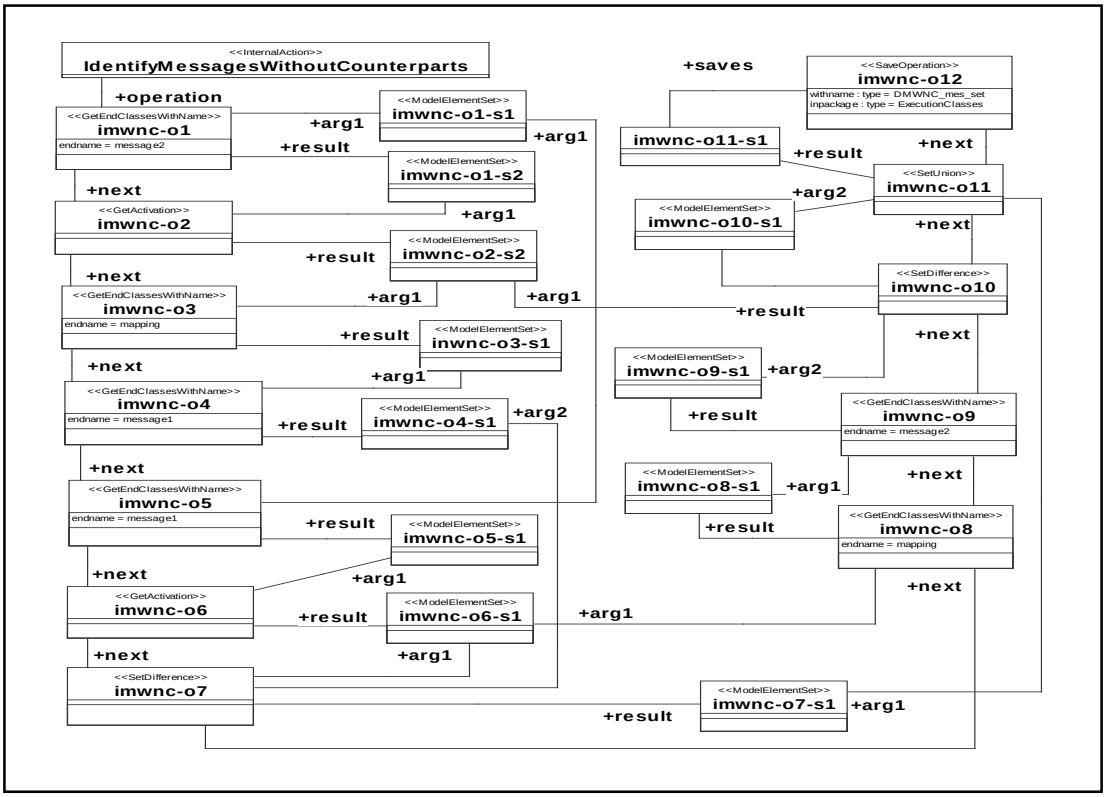

Figure 6: Specification of the internal action `IdentifyMessagesWithoutCounterparts`

To implement CR1, `IdentifyMessagesWithoutCounterparts` retrieves the messages in the activations of two overlapping messages which do not have overlapping counterparts through the execution of the following operations:

(1) `imwnc-o1`[8] – this operation retrieves one of the messages that is pointed to by the selected overlap object (`message2`).
(2) `imwnc-o2` – this is an operation which, by virtue of its stereotype (i.e., a `GetActivation` operation), finds the messages which are directly dispatched by a message (`message2` of the selected overlap object in this case).
(3) `imwnc-o3` and `imwnc-o4` – these operations find the messages that overlap with the messages dispatched by `message2` and inserts them in set `imwnc-o4-s1`.
(4) `imwnc-o5` and `imwnc-o6` – these operations find the messages which are directly dispatched by the other message that is pointed to by the selected overlap object (i.e., `message1`) and insert them in set `imwnc-o6-s1`.
(5) `imwnc-o7` – this operation finds the messages which are dispatched by `message1` and have no counterparts in the set of messages dispatched by `message2` by computing the difference between set `imwnc-o6-s1` and set `imwnc-o4-s1`.
(6) `imwnc-o8, imwnc-o9,` and `imwnc-o10` – similarly to steps 4-5 these operations identify the messages which are dispatched by `message2` and have no counterparts in the set of messages dispatched by `message1`.
(7) `imwnc-o11` – this operation takes the union of the messages which are dispatched by `message1` and `message2` and have no overlapping counterparts.
(8) `imwnc-o12` – this operation saves the messages which are dispatched by `message1` and `message2` and have no overlapping counterparts as elements of the set `DMWNC_mes_set`.

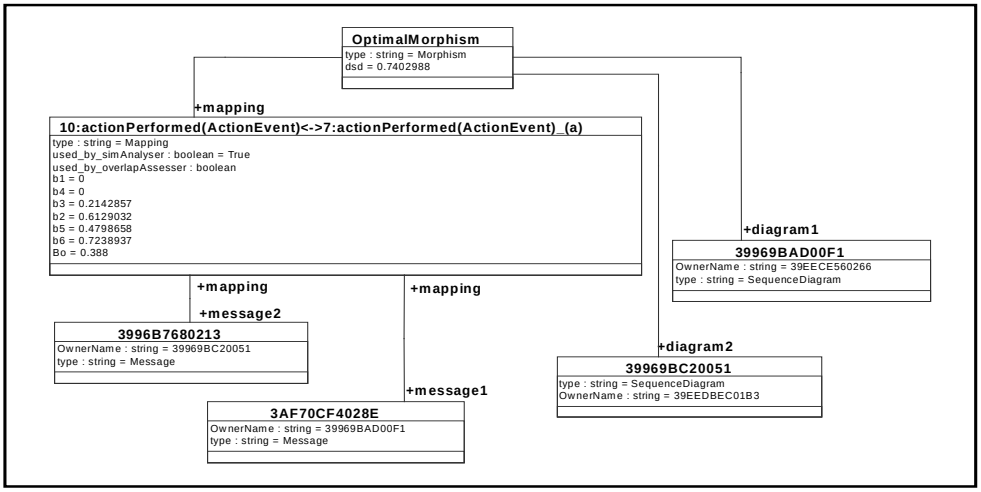

Figure 7: Overlap objects - objects that represent overlap relations

`IdentifyMessagesWithoutCounterparts` is specified as the internal action of the consistency check context

[8] `imwnc-o1` is an operation stereotyped as `GetEndClassesWithName`. Operations of this stereotype retrieve the objects which are related to the elements in their `arg1` set via the association end named by their attribute `endname`.

`Check_For_Messages_Without_Counterparts_Dispatched_By` (see Figure 4). Thus, to check the consistency rule CR1 against a pair of overlapping messages, a designer has to decide to apply this context to this pair. Note, however, that `Check_For_Messages_Without_Counterparts_Dispatched_By` may be applied only in certain parts of the reconciliation process and if the situation associated with it is satisfied. More specifically, according to the process model of Figure 4, this context can be applied to a pair of overlapping messages only after a designer has selected:

1) the executable context `Find_Overlaps_Between_Sequence_Diagrams_In` to detect overlaps in the interaction diagrams to be reconciled;
2) the choice context `Reconcile_Overlapping_Messages_In` to start the reconciliation of the overlapping messages detected in these interaction diagrams; and
3) the choice context `Reconcile_Non_Identical_Activations_Of` to start the reconciliation of the overlapping messages with the non-identical activations.

The selection and application of the consistency check context `Check_For_Messages_Without_Counterparts_Dispatched_By` to the overlap object of Figure 7 (that is the object `10:actionPerformed(ActionEvent)`↔`7:actionPerformed(ActionEvent)`) leads to the execution of the internal action `IdentifyMessagesWithoutCounterparts` which retrieves and saves in the process trace the following set of messages without overlapping counterparts:
`DMWNC_mes_set` = {`9:formulateQuery()`}

5 Handling inconsistencies

The ways of handling inconsistencies in Reconciliation+ are specified as internal actions of a special kind of executable contexts, called *inconsistency handling contexts* (see stereotype `InconsistencyHandlingContext` in Figure 3).

A consistency rule is associated with one or more inconsistency handling contexts which specify alternative ways of handling its violations. These contexts are grouped as options of a choice context which, by virtue of the definition of its situation, becomes selectable only if there is a record of violations of the particular consistency rule in the trace of the reconciliation process. The alternative inconsistency handling contexts which are available as options of this context are associated with situations which define the particular conditions under which the alternative inconsistency handling options may be applied.

In the following, we discuss how the situations and the actions of inconsistency handling contexts can be specified using the process modelling profile of

Reconciliation+. Our discussion is based on inconsistency handling contexts that the process model of Reconciliation+ incorporates to deal with inconsistencies which arise as violations of the rule CR1.

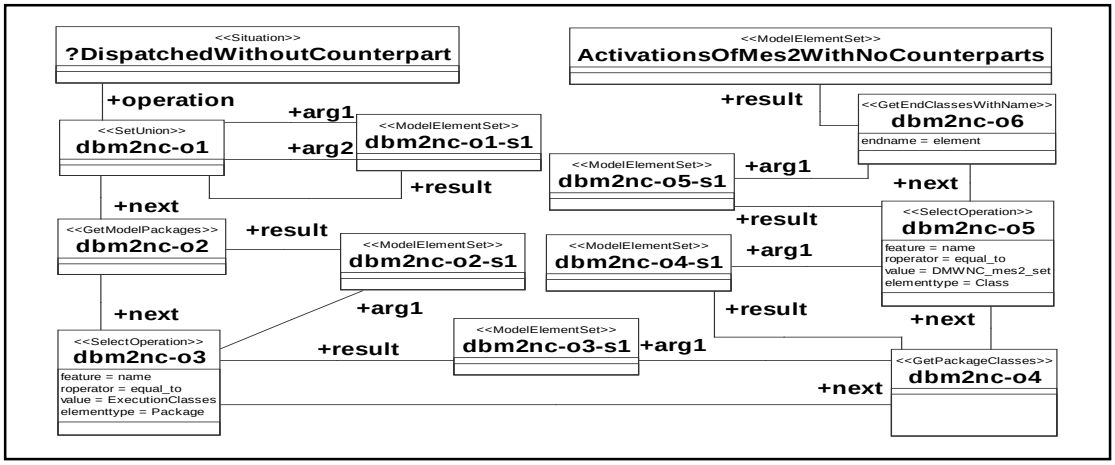

Figure 8: Specification of situation `?DispatchedWithoutCounterparts`

A violation of CR1 can be handled in different ways. One possibility, for instance, is to delete the messages without counterparts from the relevant message activation. A second possibility is to add the missing messages in the relevant activation. A third possibility is to modify the software models so that overlapping messages that dispatch messages without counterparts will no longer be considered as overlapping messages. A fourth possibility is to record the inconsistency and postpone its resolution.

The process model shown in Figure 4 includes the contexts `Add_Counterpart_For` and `Record_Message_Without_Counterpart`, which correspond to the 2nd and 4th of the above options. As shown in Figure 4, these contexts are grouped as options of the choice context `Handle_Message_With_No_Counterpart`. The latter context, due to the definition of its situation `?DispatchedWithoutCounterparts` (see Figure 8), becomes available only if there are messages that violate CR1. This is because, according to Figure 8, `?DispatchedWithoutCounterparts` is satisfied only if there is a non empty set called `DMWNC_mes_set` that has as elements the messages in the activations of two overlapping messages that violate CR1[9] (recall from Section 4 that `DMWNC_mes_set` is generated by the action of the context `Check_For_Messages_Without_Counterparts_Dispatched_By`).

`Handle_Message_Without_Counterpart` can be applied to any of the messages in `DMWNC_mes_set`. These alternative applications are generated as decisions by the process enactment engine as described in Section 3.4 and are proposed to the designer. In the case of the overlap relation between the messages `10:actionPerformed(ActionPerformed)` and

[9] The set *DMWNC_mes_set* is represented as an object that is associated with all its elements and exists in a special package called *ExecutionClasses* of the repository of the Reconciliation+ toolkit.

7:actionPerformed(ActionEvent) of the interaction diagrams I_1 and I_2 the only decision generated from Handle_ Message_Without_Counterpart is: <Handle_Message_Without_Counterpart, 9:formulateQuery()>

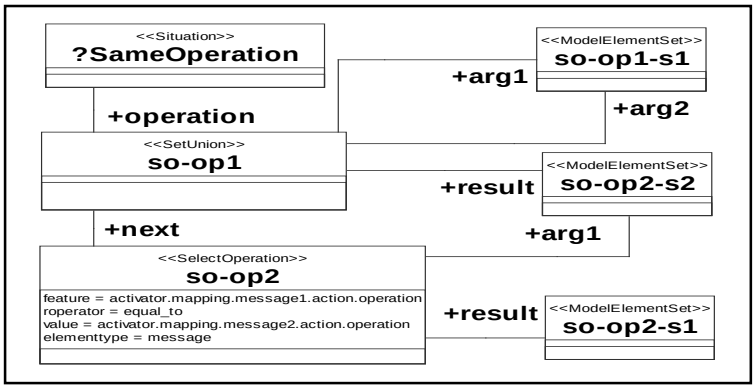

Figure 9: Specification of situation ?SameOperation

If this decision is selected, the process enactment engine checks the situations of the two alternative contexts for handling this particular kind of inconsistencies, namely Record_Message_Without_Counterpart and Add_Counterpart_For to establish if they can be applied in the case of 9:formulateQuery() .

If applicable, Record_Message_Without_Counterpart can be activated to make a persistent record of the inconsistency (i.e., a record saved after the end of the reconciliation process). This context can be selected under any circumstances as its situation contains no conditions in addition to those set by its ancestor context in the process model.

Add_Counterpart_For can be selected to create a copy of a message that does not have an overlapping counterpart and add it to the activation of the message that overlaps with its activator. Note, however, that this way of resolving the inconsistency makes sense only if the overlapping messages that gave rise to it indeed invoke the same operation in the object model. Thus, the situation of Add_Counterpart_For (i.e., the situation ?SameOperation shown in Figure 9) is specified so as to check whether this is the case. More specifically, the unary set operation so-op2 in ?SameOperation retrieves the activator *m1* of the message that caused the violation of CR1 (i.e., 9:formulateQuery()in our example) and the message *m2* that *m1* overlaps with and checks if *m1* and *m2* have been declared in the models to invoke the same operation (the operations invoked by *m1* and *m2* are identified through the evaluation of the path expressions activator.mapping.message1.action.operation and activator.mapping.message2.action.operation[10]). If that is the case,

[10] The evaluation of the sub-paths activator.mapping.message1(2)in these path expressions locate the overlapping messages in the activations of which the message without the counterpart was encountered. These sub-paths assume the representation of overlap morphisms and relations by the tool that we have built to support Reconciliation+ (shown in Figure 7). The remaining sub-paths

?SameOperation is satisfied and, therefore, Add_Counterpart_For can be selected for the message 9:formulateQuery().

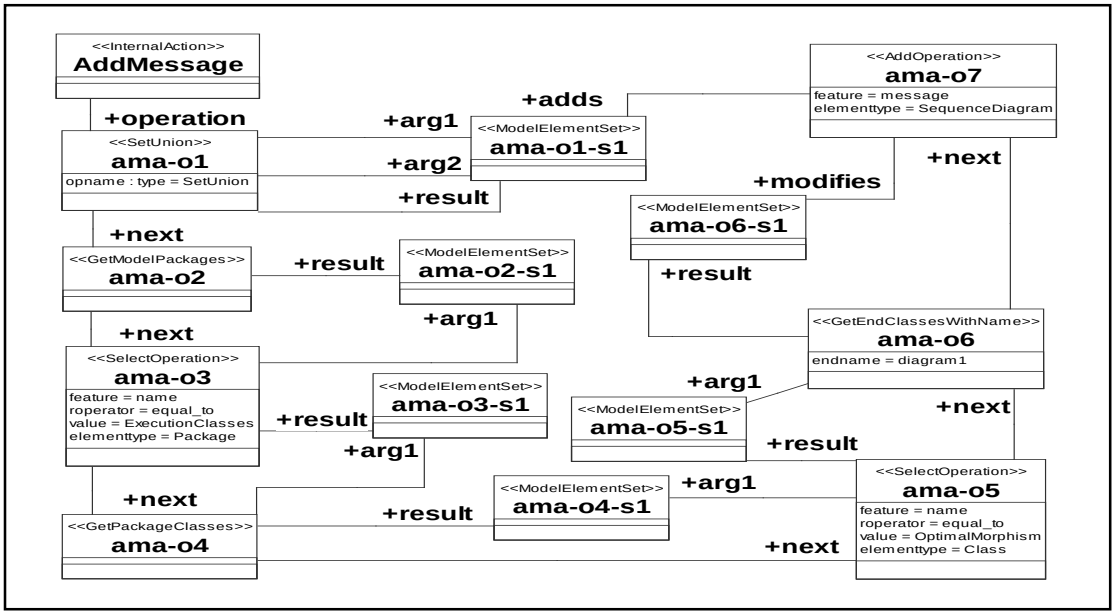

Figure 10: Specification of action AddMessage

The creation of a copy of this message is the result of executing the internal action AddMessage of Add_Counterpart_For. The specification of this action is shown in Figure 10. According to this specification, to execute AddMessage, the process enactment engine first locates the interaction diagram of the overlapping message of the activator of the message that Add_Counterpart_For was selected for (i.e., the diagram I_1 in our example) by executing the operations ama-o2 to ama-o6, and then adds to the set of the messages of this diagram a copy of this message by executing the operation ama-o7.

6 Tool support

Reconciliation+ is supported by a toolkit which incorporates: (a) a tool that detects overlaps between object interaction models, and (b) an engine which enacts the process model of the method to drive the activity of reconciling interaction diagrams. This toolkit has been implemented as an add-on utility for *Rational Rose* (a CASE tool supporting UML) using the API of this tool [21].

The architecture of the Reconciliation+ toolkit is shown in Figure 11. As shown in this figure, the toolkit stores the models to be reconciled as collections of UML class models and sequence diagrams in a model repository that is accessible through the API of Rose. The overlap morphisms which are detected by the overlap detection tool,

(*.action.operation) assume the standard representation for UML models that is established by the UML meta-model [9].

the process model of the method, and the trace of the enactment of this model are also represented and stored as UML object models in the same repository.

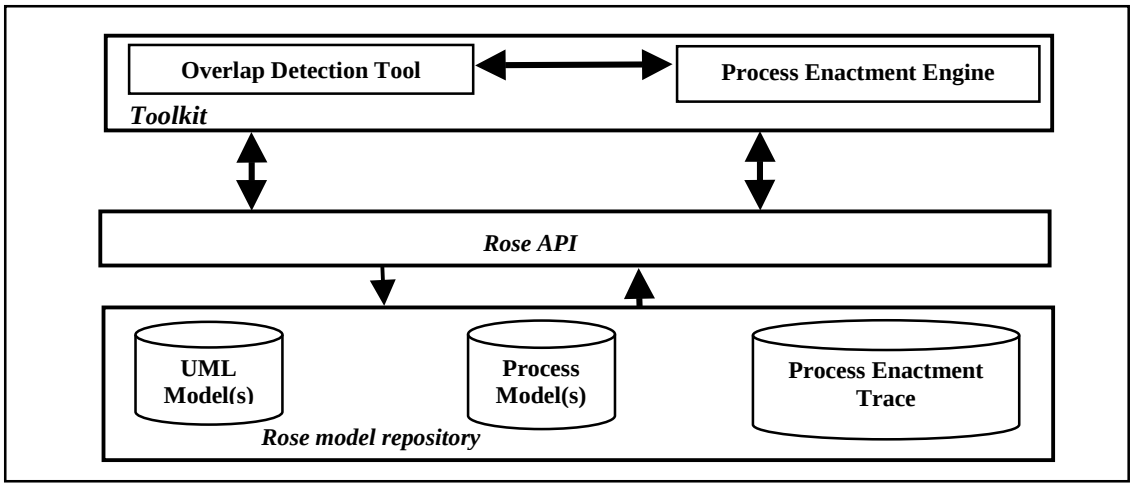

Figure 11: Architecture of the Reconciliation+ toolkit

Figure 12 shows a snapshot of the overlap detection tool following its invocation to identify overlaps between the interaction diagrams of Figure 1.

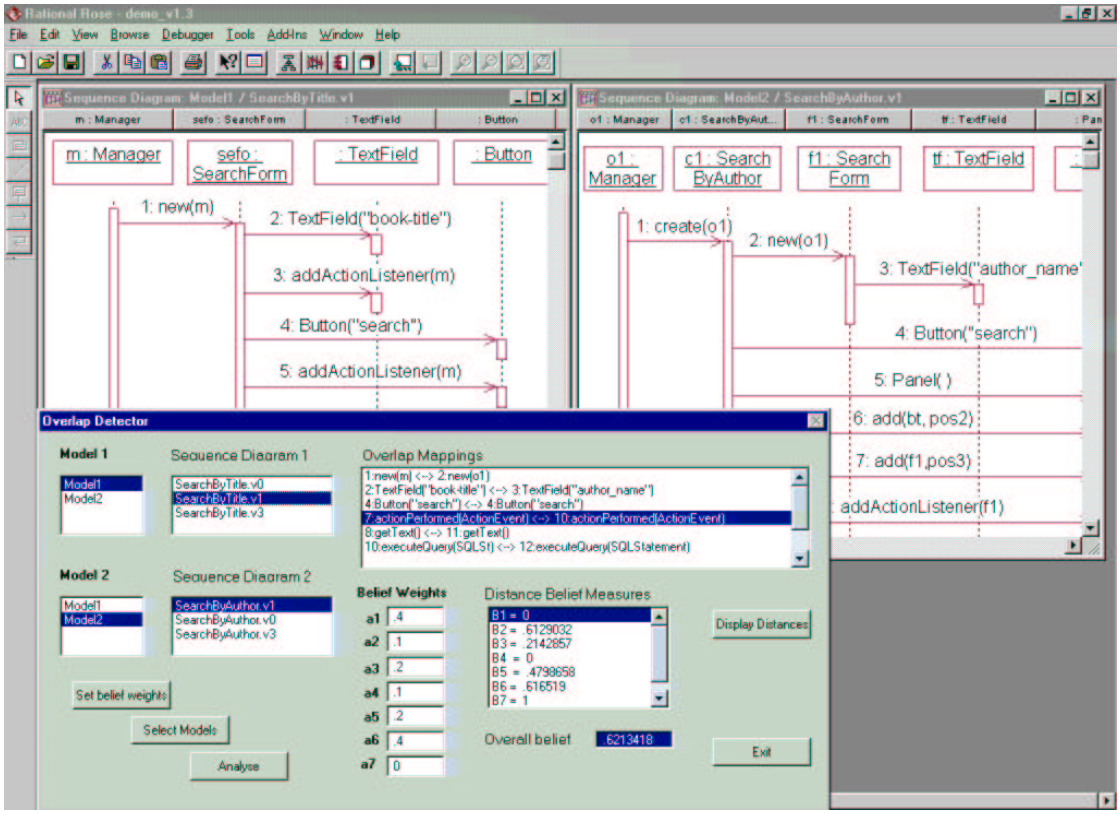

Figure 12: Overlap Detector – detection of overlap morphism between I_1 and I_2

Figure 13 shows a snapshot of the process enactment engine of the toolkit. This snapshot shows a point in the enactment of the Reconciliation + process model where the consistency check context

`Check_For_Messages_Without_Counterparts_Dispatched_By` may be selected to check for violations of the rule CR1 by messages in the activations of the overlapping pair of messages: `10:actionPerformed(ActionEvent)`, and `7:actionPerformed(ActionEvent)` of the diagrams of Figure 1. As shown in Figure 13, the process enactment engine gives a designer the options of: (a) applying any of the contexts which become available at the current point in the enactment of the process model (see list *Next Decision*), and (b) asking for tactical guidance or equivalently go back to the previous decision point in the enactment of the process model. Note also that the enactment engine keeps a record of the decisions that have been made up to the current point in the enactment of a process (process trace) and presents them to the designer (see the list *Decisions made so far*).

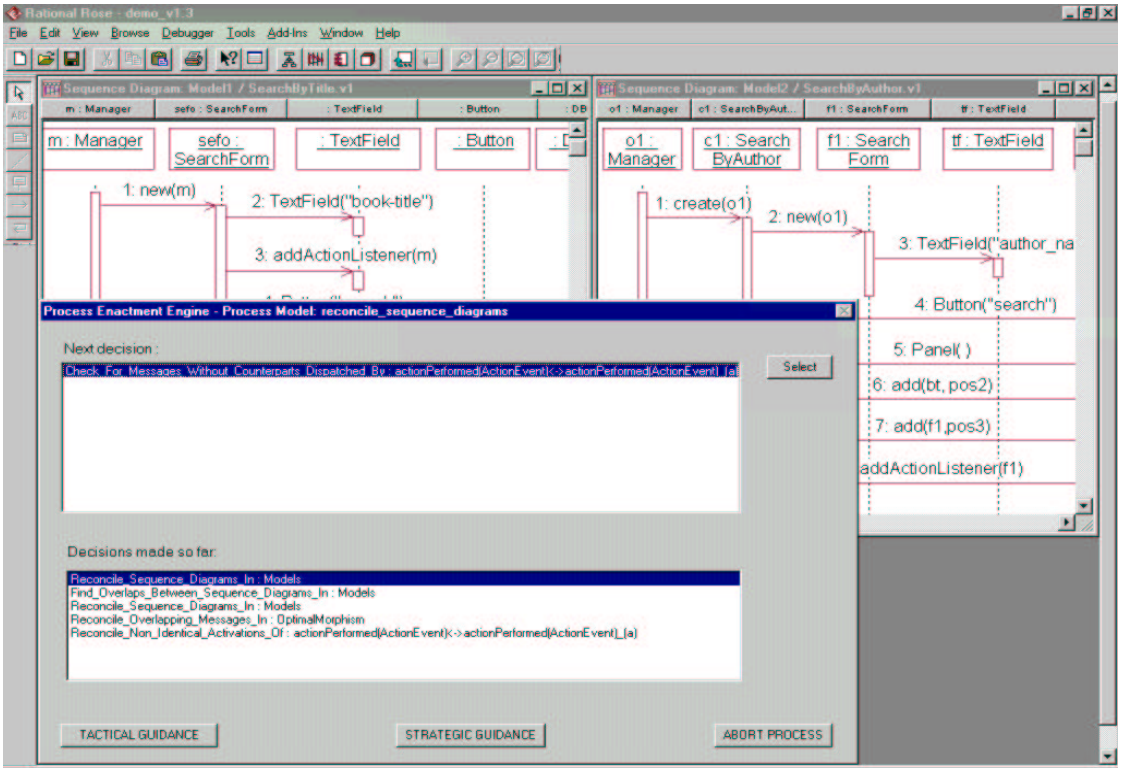

Figure 13: Process Enactment Engine

7 Case study

To evaluate Reconciliation+ we have carried out a case study. The objectives of this study were to:

(i) measure the performance of the overlap detection algorithm that is deployed by the method in terms of *recall* (i.e., the ability to detect overlap relations that humans would identify), and *precision* (i.e., the ability to detect correct overlap relations);

(ii) assess the sensitivity of the overlap detection algorithm against the granularity of the models it is applied to; and

(iii) investigate the diversity of the ways that may be used to handle inconsistencies, and assess whether the process model of the method could support the specification of inconsistency handling contexts to implement these ways.

In this case study, we used 5 UML models built by MSc students at City University. These models had been constructed to specify a system supporting banking transactions through the use of ATM machines.

7.1 Recall and precision of overlap detection algorithm

To evaluate the recall and precision of the overlap detection algorithm of Reconciliation+, we performed 50 comparisons of pairs of UML interaction diagrams drawn from the models of our case study.

Following a manual identification of overlap relations between these interaction diagrams, we detected overlap relations between them using the algorithm specified in Section 2, and subsequently measured the recall and the precision of the algorithm according to the following formulas:

$$\text{Precision} = \frac{|AO^{ij} \cap MO^{ij}|}{|AO^{ij}|} \quad \text{and} \quad \text{Recall} = \frac{|AO^{ij} \cap MO^{ij}|}{|MO^{ij}|}$$

where

- AO^{ij} is the set of the overlap relations that were detected by the overlap detection algorithm between the messages of two interaction diagrams *i* and *j*; and
- MO^{ij} is the set of the manually identified overlap relations between the messages of two interaction diagrams *i* and *j*.

Model	**| MO ∪ AO |**	**#ID-Pairs**	**Recall**		**Precision**	
			Mean	*St. Dev.*	*Mean*	*St. Dev.*
1	77	10	1.00	0	1.00	0
2	60	11	0.86	0.20	0.99	0.03
3	74	6	0.92	0.20	0.90	0.2
4	231	15	0.87	0.08	0.98	0.05
5	87	8	0.91	0.06	1.00	0
All	529	50	0.91	0.13	0.98	0.08

Table 2: Recall and precision of overlap detection algorithm

Table 2 shows the average and standard deviation of the recall and precision measures that we obtained for pairs of diagrams drawn from each of the five object models used in our experiments, and from all of the models (these results were obtained for the values of parameters α_1–α_6 and b_t used in Section 2.3). It also shows the number of the pairs of interaction diagrams that were compared in each model (i.e., *#ID-Pairs*) and the total number of the overlap relations detected by the overlap algorithm and the overlap relations that were manually identified between them in each case (i.e., $|MO \cup AO|$). The object models that we used in our experiments and the overlap relations identified manually by the experts and the overlap detection algorithm are available from: http://www.soi.city.ac.uk/~gespan/imoosd_case_studies.html

As shown in Table 2, the overlap detection algorithm had very high precision (0.98 on average across all models) and relatively high recall (0.91 on average across models). Also there was a low variation in these measures across different pairs of interaction diagrams and models (the standard deviation of the recall and precision measures across all models were 0.13 and 0.07, respectively). Although preliminary, the above results indicate that the overlap detection algorithm has a very low probability of producing false overlaps and is capable of detecting a high proportion of the overlaps indicated by human designers.

7.2 Effect of model granularity on overlap detection

To explore the effect of model granularity on our overlap detection algorithm, we also carried out a correlation analysis of the obtained recall and precision measures against the following measures of model granularity:

(i) the number of classes in a model (#CL) – this measure was expected to affect the beliefs b_3 and b_4,
(ii) the average number of superclasses of a class in a model (#Isa*) – this measure was expected to affect directly the beliefs b_2 and b_3 and implicitly b_5,
(iii) the average degree of operation overriding in a model (i.e., the ratio of classes which inherit an operation but override it – #OO) – this measure was expected to affect directly the beliefs b_1 and implicitly b_5, and
(iv) the average number of dispatched messages in message activations (#AC) – this measure was expected to affect the beliefs b_6.

Model	**#Cl**	**#Isa***	**#OO**	**#AC**
1	697	0.014	1	3.685
2	200	0.420	0.305	1.288
3	55	0.309	0.584	2.722
4	702	0.017	0.772	1.030
5	696	0.012	0.696	1.911
Correlation Coefficients				
Recall	0.29	–0.28	0.76	0.96
Precision	0.74	–0.74	0.24	–0.16

Table 3: Model granularity measures and correlation with recall and precision

Table 3 presents the above granularity measures for the five models of our case study and their correlation with the recall and precision measures obtained for these models. As shown in the table, recall had strong positive correlations with #OO and #AC. This was expected as the weights of the d_1 and d_6 distances, which were directly affected by #OO and #AC, in establishing beliefs in overlaps were relatively higher than the weights of the other four distances of our algorithm (see values of a_1 and a_6 in Section 2.3). The observed negative correlation of recall with the average number of superclasses (#Isa*) is likely to have been the result of the small number of superclasses that the senders and receivers of messaged had in the considered models. It should also be noted that only the positive correlation of #AC with recall (i.e., 0.96) was statistically significant (at α=0.10).

In the case of precision, positive correlations were detected only for #CL and #OO and none of the obtained correlations was statistically significant at α=0.10. This may be attributed to the fact that precision was very high across all models and therefore the differences in model granularity did not have any significant effect on it.

The results of the above correlation analysis indicate that the overlap detection algorithm is not over-sensitive to the degree of completeness and elaboration of design models. This was expected due to the use of six different criteria that focus on different parts of software design models for detecting overlaps. Clearly, however, our results are only preliminary and need to be confirmed by additional experiments.

7.3 Diversity of inconsistency handling options

The third objective of our case study was to investigate the diversity of the ways that may be used to handle inconsistencies, and to assess whether the process model of the method could support the specification of inconsistency handling contexts to implement these ways. In this part of the study, we selected 4 consistency rules to check against the overlapping messages that were detected by the overlap detection algorithm in the first part of the study. The selected rules were:

- CR1 – this was the rule specified in Sections 1 and 4;
- CR2 – this was a rule that required the operations invoked by two overlapping messages to have the same name;
- CR3 – this was a rule that required the operations invoked by two overlapping messages to be defined in the same class; and
- CR4 – this was a rule that required the operations invoked by two overlapping messages to have the same number and types of parameters.

The above rules were selected since they capture the main forms of possible discrepancies in the specifications of operations which are invoked by overlapping messages.

	CR1	CR2	CR3	CR4
Violations	40	3	2	23
No need to be resolved	10	0	1	9
Need to be resolved	29	3	1	14
Alternative resolutions	6	2	1	4

Table 4: Inconsistencies and alternative ways of handling them

Table 4 shows the number of the detected violations of each of these rules, the number of the cases where we found that it was necessary to resolve the detected inconsistency, and the number of the different alternative ways that we advocated for resolving the inconsistencies of the same rule. As it may be seen from the figures of this table, some diversity was indeed observed in the ways of handling violations of the same rule. However, due to the size of our experiments these results cannot be generalised. Nevertheless, the case study has given rise to some interesting observations that we discuss in the following.

More specifically,

- In some cases, following the violation of a consistency rule, it was realised that the overlap relation detected by the method and checked against the rule was wrong. In such cases, it was not necessary to take any inconsistency resolution action. The way that was adopted to handle such inconsistencies was to record them along with an annotation that the overlap relation that gave rise to them were wrong.
- There were cases, where the violation of a consistency rule should be tolerated as the relevant rule should not be satisfied by particular pairs of overlapping messages. This was, for instance, the case with some of the violations of rule CR1 in which two overlapping messages were dispatching messages with the same signature but different activations which had not themselves been detected as overlapping messages. In these cases, the non overlapping messages were invoking *polymorphic* operations (i.e., operations defined in different classes with the same signature and different implementations as, for example, the operations `SearchForm.actionPerformed(e: ActionEvent)` and `DatabaseActionListened.actionPerformed(e: ActionEvent)` in Figure 2) and therefore the resolution of the inconsistency was not necessary. To deal with such cases, it is, in principle, possible either to amend the conditions of the relevant rule so as to ignore non overlapping messages that invoke polymorphic operations, or to add inconsistency handling contexts for the rule that ignore its violations by such messages. Although both these strategies can be accommodated by the method, Reconciliation+ does not incorporate at its current stage of development criteria for helping designers to decide which of the two options is more appropriate in specific circumstances.
- The selection of the best inconsistency handling strategy in a given situation may depend on the satisfiability (or unsatisfiability) of more than one rules. The violation of both rule CR1 and CR2, for instance, in some cases led to the realisation that the overlap relation that gave rise to the inconsistencies was wrong. In these cases, the relevant inconsistencies were ignored. In other cases, however, where CR3 was violated but CR1 was satisfied the inconsistencies were resolved by changing the name of one of the overlapping messages. In Reconciliation+, cases like these could be handled by specifying the situations of the inconsistency handling contexts for a specific rule so as to check if the pair of the overlapping messages that violated the rule has also violated another rule.

8 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][15][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 and the consistency checking is performed using a tool developed using an XML development platform. A critical survey of all the above techniques may be found in [16].

9 Conclusions and further work

In this paper, we have presented Reconciliation+, a method that guides designers in reconciling object interaction diagrams specified as part of software design models. The method detects overlaps between messages in interaction diagrams, checks consistency rules that overlapping messages must satisfy, and provides 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 activity. Reconciliation+ can be applied at the design phase of software development following the specification of, at least, partial object-oriented software design models defining the basic class structure and interactions of a system. The method can be used in conjunction with development approaches which require the development of design models prior to implementation and approaches that advocate an incremental development of such models and/or software systems.

We have evaluated Reconciliation+ in a case study the objectives of which were to measure the recall and precision of the overlap detection algorithm deployed by the method, investigate its sensitivity to variations of model granularity, and investigate the diversity of the strategies that may be needed in handling inconsistencies. This case study has shown positive preliminary results regarding the recall and precision of the overlap detection algorithm of the method, and has demonstrated some diversity in the nature of inconsistency handling strategies which can be accommodated by the method. It has also shown that the overlap detection algorithm is not prohibitively sensitive to the degree of elaboration and completeness of the models it is applied to. Thus, Reconciliation+ can be applied to models specified at varying levels of completeness.

Further experimentation is, however, required to confirm these findings. Furthermore, it is necessary to evaluate the method against some *usability criteria* including the difficulty in extending its process model with new consistency rules and ways of handling their violations, as well as with intermediate decisions (choice contexts) to guide software designers in selecting amongst alternative inconsistency handling options. Currently, we are evaluating the method along these lines.

Acknowledgements
The work presented in this paper has been partially funded by the EPSRC grant no. GR/M57422.

References

[1] Boehm B, In H (1996), Identifying Quality Requirements Conflicts, IEEE Software, 25-35.

[2] Cheung K, Chow K, Cheung T (1998), Consistency Analysis on Lifecycle Model and Interaction Model, Proc. of the 7th Int. Conference on Object-Oriented Information Systems, 427-441.

[3] Easterbrook S (1991), Handling Conflict between Domain Descriptions with Computer-Supported Negotiation, Knowledge Acquisition, 3: 255-289.

[4] Emmerich W, Finkelstein F, Montangero C, Antonelli S, Armitage S (1999), Managing Standards Compliance, IEEE Transactions on Software Engineering, 25(6): 836-851.

[5] Finkelstein A., Gabbay D, Hunter, A, Kramer, J, and Nuseibeh, B (1994), Inconsistency Handling In Multi-Perspective Specifications, IEEE Transactions on Software Engineering, 20(8): 569-578.

[6] Glinz M (1995), An Integrated Formal Model of Scenarios Based on Statecharts, In Proc. of the 5th European Software Engineering Conference, LNCS 989, Springer-Verlag, 254-271.

[7] Heimdahl M.P.E, Leveson N (1996), Completeness and Consistency in Hierarchical State-Based Requirements, IEEE Transactions in Software Engineering, 22(6): 363-377.

[8] Heitmeyer C, Jeffords R, Kiskis D (1996), Automated Consistency Checking Requirements Specifications, ACM Transactions on Software Engineering and Methodology, 5(3): 231-261.

[9] OMG, Unified Modeling Language Specification (Action Semantics) – V. 1.4. Available from: http://www.omg.org/technology/documents/modeling_spec_catalog.htm .

[10] Papadimitriou C, Steiglitz K (1982), Combinatorial Optimisation: Algorithms and Complexity, Prentice-Hall Inc.

[11] Pohl K (1996), Process-Centred Requirements Engineering, Advanced Software Development Series, J. Kramer (ed), Research Studies Press Ltd., ISBN 0-86380-193-5, London.

[12] Robinson, W. and Fickas S (1994), Supporting Multi-Perspective Requirements Engineering, In Proc. of the IEEE Conference on Requirements Engineering, IEEE Computer Society Press, 206-215.

[13] Si-Said S, Rolland C, Grosz G (1996), MENTOR: A Computer Aided Requirements Engineering Environment, Proc. of the 8th International Conference on Advanced Information Systems Engineering, 22-43.

[14] Spanoudakis G, Constantopoulos P (1996), Elaborating Analogies from Conceptual Models, International Journal of Intelligent Systems, 11(11): 17-974.

[15] Spanoudakis G, and Finkelstein A (1997), Reconciling requirements: a method for managing interference, inconsistency and conflict, Annals of Software Engineering, Special Issue on Software Requirements Engineering, 3: 459-475.

[16] Spanoudakis G, Zisman A. (2001), 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, 329-380.

[17] Lamsweerde A, Darimont R, Letier E (1998), Managing Conflicts in Goal-Driven Requirements Engineering, IEEE Transactions on Software Engineering, 24(11): 908-926.

[18] Zisman A, Emmerich W, Finkelstein A (2000), Using XML to Specify Consistency Rules for Distributed Documents, Proc. of 10th Int. Workshop on Software Specification and Design.

[19] Shafer G (1976), A Mathematical Theory of Evidence, Princeton University Press.

[20] http://java.sun.com/j2se/1.3/docs/guide/awt/

[21] Spanoudakis G (2000), An Algorithm for Detecting Overlaps between Models of Object Interactions, Technical Report Series, TR-2000/03, ISSN 1364-4009, Department of Computing, City University.

[22] http://www.rational.com/products/rose/index.jsp.