HAL open science

Weaving Multiple Aspects in Sequence Diagrams

Jacques Klein, Franck Fleurey, Jean-Marc Jézéquel

To cite this version:

Jacques Klein, Franck Fleurey, Jean-Marc Jézéquel. Weaving Multiple Aspects in Sequence Diagrams. LNCS Transactions on Aspect-Oriented Software Development, 2007, LNCS 4620, pp.167-199. inria-00505223

HAL Id: inria-00505223

https://inria.hal.science/inria-00505223v1

Submitted on 23 Jul 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Weaving Multiple Aspects in Sequence Diagrams ⋆

Jacques Klein[1], Franck Fleurey[1], and Jean-Marc Jézéquel[2]

[1] IRISA/INRIA, Campus de Beaulieu,
35042 Rennes cedex, France,
jacques.klein@irisa.fr,
franck.fleurey@irisa.fr
[2] IRISA/ Université de Rennes 1, Campus de Beaulieu,
35042 Rennes cedex, France,
jezequel@irisa.fr

Abstract. Handling aspects within models looks promising for managing crosscutting concerns early in the software life-cycle, up from programming to design, analysis and even requirements. At the modeling level, even complex behavioral aspects can easily be described for instance as pairs of sequence diagrams: one for the pointcut specifying the behavior to detect, and the second one for an advice representing the wanted behavior at the join point. While this is fine for informal documentation purposes, or even intuitive enough when a single aspect has to be woven, a more precise semantics of both join point detection and advice weaving is needed for using these modeling artifacts for Model Driven Engineering activities such as code generation or test synthesis. This paper proposes various interpretations for pointcuts that allow multiple behavioral aspects to be statically woven. The idea is to allow join points to match a pointcut even when some extra-messages occur in between. However, with this new way of specifying join points, the composition of the advice with the detected part cannot any longer be just a replacement of the detected part by the advice. We have to consider the events (or the messages) of the join point, but also the events which occur between them, and merge them with the behavior specified within the advice. We thus also propose a formal definition of a new merge operator, and describe its implementation on the Kermeta platform.

1 Introduction

The idea of encapsulating crosscutting concerns into the notion of aspects looks very promising for complementing the usual notion of modules available in most languages. By localizing these crosscutting concerns, the software engineer can get a better control over variations, either in the product line context or for software evolutions. The need to isolate these crosscutting concerns has been popularized by the AspectJ programming language, but there is a growing interest in

⋆ This work has been partially supported by the European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe), 2004-2008.

also handling them earlier in the software life-cycle, for instance at design time [6], or during requirements analysis [2, 29, 24, 14] and notably through the Early Aspect community and the series of Early Aspect workshops[3].

At modeling level, even complex behavioral aspects can easily be described for instance as pairs of UML 2.0 Sequence Diagrams (SDs), one SD for the pointcut (specification of the behavior to detect), and the second one for an advice representing the wanted behavior at the join point. This is usually fine enough for informal documentation purposes, or even intuitive enough when a single aspect has to be woven. The idea of Model Driven Engineering is however that it should be possible to use these modeling artifacts beyond mere documentation purposes, for example for validation purposes (simulation or test case generation) and also for code generation, including targeting non-aspect-oriented platforms (e.g. vanilla Java, or real-time embedded systems). A more precise semantics of both join point detection and advice weaving is then needed.

In this paper, we focus on finite scenarios expressed by means of SDs. We will call *base scenario* a scenario which describes the concern that determine the dominante structure of the system, and *behavioral aspect* a pair of scenarios which describes a concern that crosscuts the base scenario. For join point detection at modeling time, we need to statically find where in the base scenarios are the join points. The partial order induced by a SD and the hierarchical nature of UML 2.0 SD (similar to High-Level Message Sequence Charts [13]) makes it necessary to address the problem at the semantic level [18] with static analysis techniques such as loop unrolling, etc.

For the composition of the advice into the base SD, when we are weaving a single aspect into a base SD and when a join point[3] is a strict sequences of messages, the composition is trivial once the join point has been identified: the advice SD just replaces the portion of the SD that is matched by the pointcut at the join point. However weaving multiple aspects at the same join point can be difficult if a join point is simply defined as a strict sequence of messages, because aspects previously woven might have inserted messages in between.

The contribution of this paper is to propose a new interpretation for pointcuts expressed as SDs to allow them to be matched by join points where some messages may occur between the messages specified in the pointcut. However, with this new way of specifying join points, the composition of the advice with the detected part cannot any longer be a replacement of the detected part by the advice. We have to consider the events (or the messages) of the join point which are not specified within the pointcut and merge them with the behavior specified within the advice. We thus propose a formal definition of a new merge operator, called an *amalgamated sum*, and describe its implementation on the meta-modeling platform Kermeta [19].

[3] Note that in this paper, we borrowed the term "join point" from AspectJ terminology. In contrast to AspectJ, however, we consider "join points" as a representation of an element or a collection of elements of the language of scenario used rather that as "well-defined points in the execution of the program" (cf. [16]). The term join point will be formally defined in Section 3.

The rest of the paper is organized as follows. Section 2 formally introduces the scenario language used and the notion of behavioral aspects. Section 3 introduces various interpretations for join points and Section 4 describes three detection algorithms for these join points. Section 5 presents our composition operator for sequence diagrams (*amalgamated sum*). Section 6 presents its implementation on the Kermeta platform [19]. Section 7 discusses future works whose aim at overcoming a current limitation of our approach. Section 8 compares our approach with related works, and section 9 concludes this work.

2 Sequence Diagrams and Aspects

2.1 Scenarios: UML 2.0 Sequence Diagrams

Scenario languages are used to describe the behaviors of distributed systems at an abstract level or to represent systems behavioral requirements. They are close to users understanding and they are often used to refine use cases with a clear, graphical and intuitive representation. Several notations have been proposed, among which UML 2.0 Sequence Diagrams (SDs) [21], Message Sequence Charts (MSCs) [13] or Live Sequence Charts [8]. In this paper, the scenarios will be expressed by UML 2.0 SDs. To define formally SDs in an easier way, we call basic sequence diagrams (bSD), a SD which corresponds to a finite sequence of interactions. We call combined sequence diagrams (cSDs) a SD which composes bSDs (with sequence, alternative and loop operators). In this way, a cSD can define more complex behaviors (even infinite behaviors if the cSD contains loops).

More specifically, bSDs describe a finite number of interactions between a set of objects. They are now considered as collections of events instead of ordered collections of messages in UML 1.x, which introduce concurrency and asynchronism increasing their power of expression. Figure 1 shows several bSDs which describe some interactions between the two objects *customer* and *server*. The vertical lines represent lifelines for the given objects. Interactions between objects are shown as arrows called messages like *log in* and *try again*. Each message is defined by two events: message emission and message reception which induces an ordering between emission and reception. In this paper, we use arrows represented with an open-head that corresponds to asynchronous messages[4] in the UML2.0 standard notation. Asynchronous means that the sending of a message does not occur at the same time as the corresponding reception (but the sending of a message does necessarily precede the corresponding reception). Consequently, in Figure 2, the event e_3 corresponding to the reception of the first message a and the event e_2 corresponding to the sending of the second message a are not ordered. Events located on the same lifeline are totally ordered from top to bottom (excepted in specific parts of the lifeline called coregions).

We recall that in the UML2.0 specification, the semantics of an Interaction (a Sequence Diagram) is a set of traces, i.e., a set of sequences of events. Consequently, all events are not totally ordered. For instance, in Figure 2, the bSD M

[4] We use asynchronous messages to be more general

generates two traces: $\{< e_1, e_3, e_2, e_4 >; < e_1, e_2, e_3, e_4 >\}$. These traces imply that the events e_2 and e_3 are not ordered. For this reason, we use the notion of partial order as used in other languages of scenarios as Message Sequence Charts to define formally the notion of bSD:

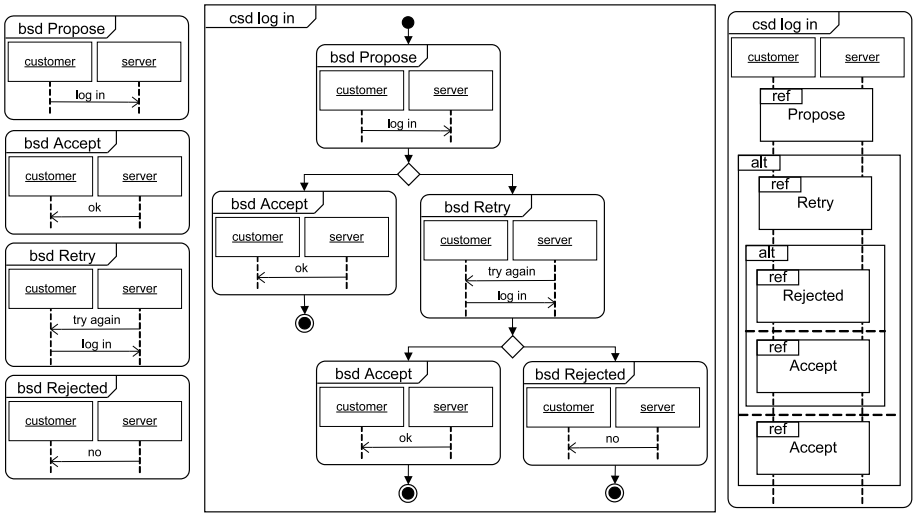


Fig. 1. Examples of bSDs and combined SD

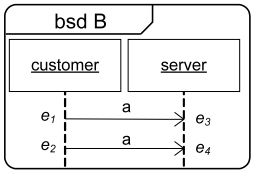


Fig. 2. Example of a bSD

Definition 1. *A basic sequence diagram is a tuple* $B = (I, E, \leq, A, \alpha, \phi, \prec)$ *where:* I *is a set of objects participating to the interaction,* E *is a finite set of events (message emissions and receptions),* $\leq$ *is a partial ordering imposed by lifelines and messages,* A *is a set of actions (message name), and* α *and* ϕ *are mappings associating respectively an action name and a location (i.e. an object affected by the event) with an event.* $\prec \subseteq E \times E$ *is a relation that pairs message emissions and receptions.*

In Definition 1, the sentence "$\leq$ is a partial ordering imposed by lifelines and messages" means that events are totally ordered along a same lifeline (for

instance, in Figure 2 the event e_1 precedes the event e_2, and the event e_3 precedes the event e_4), and a message emission must always precede the corresponding reception (for instance, the event e_1 precedes the event e_3, and the event e_2 precedes the event e_4). Then, by transitivity, the partial order $\leq$ is obtained. Note that the events within an individual lifeline are totally ordered only if each event is unique. To ensure the uniqueness of each event, we use a unique identifier for each event.

We will denote by $T(e)$, the type of an event e. The type of an event indicates whether an event is a send event or a receive event. We will denote by $min(E) = \{e \in E | \forall e' \in E, e' \leq e \Rightarrow e' = e\}$, the set of minimal events of E, i.e., the set of events which have no causal predecessor. We will denote by $pred_{\leq,E}(e) = \{e' \in E | e' \leq e\}$, the set of predecessors of the event e, and by $succ_{\leq,E}(e) = \{e' \in E | e \leq e'\}$, the set of successor of e. These two notations can be used with a subset E' of the set E: $pred_{\leq,E}(E') = \{e \in E | \exists e' \in E', e \leq e'\}$ and $succ_{\leq,E}(E') = \{e \in E | \exists e' \in E', e' \leq e\}$. Slightly misusing the notation, when M' is a bSD which is a "part" of a bSD M, we will denote $pred(M')$ as the set of events of M' plus the set of predecessors of the events of M'. Finally, we will also use, for instance, the notation $pred_{<,E}(e) = \{e' \in E | e' < e\}$ to denote the set of strict predecessors of the event e (order $<$ instead of $\leq$) .

Basic SDs alone do not have sufficient expressive power: they can only define finite behaviors, without real alternatives. For this reason, they can be composed with operators such as sequence, alternative and loop to produce a SD called combined SDs (cSD) (also called UML 2.0 Interaction Overview Diagram). Figure 1 shows two equivalent views of the same cSD called *log in* (one view is more compact). This cSD *log in* represents the specification of a customer log on a server. If the customer makes two bad attempts, then he/she is rejected. Else, he/she is accepted. We can see that the cSD allows an alternative between the bSDs Accept and Retry, and between the bSDs Accept and Rejected. The cSD also composes sequentially the bSDs Propose and Accept (denoted $Propose \bullet Accept$), the bSDs Propose and Retry (denoted $Propose \bullet Retry$), etc... The notion of sequential composition (noted $\bullet$ or *seq* with the UML2 notation) is central to understanding the semantics of cSD. Note that we use the notion of *weak sequential composition* presented in the UML 2.0 specification [21](p 454). Roughly speaking, (weak) sequential composition of two bSDs consists of gluing both diagrams along their common lifelines. Note that the sequence operator only imposes precedence on events located on the same lifeline, but that events located on different lifelines in two bSDs $M1$ and $M2$ can be concurrent in $M1 \bullet M2$. Sequential composition can be formally defined as follows:

Definition 2 (Sequential Composition). [5]

The sequential composition of two bSDs $M_1 = (I_1, E_1, \leq_1, A_1, \alpha_1, \phi_1, \prec_1)$ *and* $M_2 = (I_2, E_2, \leq_2, A_2, \alpha_2, \phi_2, \prec_2)$ *is the bSD* $M_1 \bullet M_2 = (I_1 \cup I_2, E_1 \uplus E_2, \leq_{1\bullet 2}, A_1 \cup A_2, \alpha_1 \cup \alpha_2, \phi_1 \cup \phi_2, \prec_1 \uplus \prec_2)$*, where:* $\leq_{1\bullet 2} = \left(\leq_1 \uplus \leq_2 \uplus \{(e_1, e_2) \in E_1 \times E_2 \mid \phi_1(e_1) = \phi_2(e_2)\}\right)^*$

To calculate the new partial ordering $\leq_{1\bullet 2}$, sequential composition consists in ordering events e_1 in bMSC M_1 and e_2 in bMSC M_2 if they are situated on the same lifeline, and then compute the transitive closure of this ordering. In this definition, $\uplus$ is the disjoint union of two multisets, i.e. an usual union operation where common elements of both sets are duplicated. This operator is necessary because even if the two operands have two identical events (events with the same name), the two events have to present in the result. Indeed, for instance imagine we want to make the sequential composition $B \bullet B$, where B is a bSD which contains only one message A. In this case, it is obvious that the two operands contain the same events, but in the result we want that all the events appear. Thus, in the sequential composition we have to copy and rename the identical events and it is made with the disjoint union.

The cSD *log in* can be considered as a generator of a set of behaviors. For instance, the cSD *log in* generates the set of behaviors $\{Propose \bullet Accept, Propose \bullet Retry \bullet Accept, Propose \bullet Retry \bullet Rejected\}$. This set of behaviors can be potentially infinite (as soon as a combined SD contains the operator loop, the set of bSDs generated is infinite), but in this paper we will only consider finite SDs.

Figure 3 depicts the sequence diagram metamodel used to implement the weaving process presented in this paper (the implementation is described in Section 6). We present this metamodel in this section to show that it fits very well with the previous definitions of bSD and cSD. In Figure 3, we can note that cSD has an automata structure, in that a cSD contains a set of nodes and a set of transitions which are linked to bSDs. In this way, cSD can compose bSDs through sequences, alternatives and loops. We can also note that a bSD contains a set of objects (class Instance), a set of events (class Event) and a partial order on the events. The partial order is built with the class EventCouple which orders two events: the event "prec" precedes the event "succ". A set of pairs of events $(prec, succ)$ forms the partial order. The class Event is linked to the class Instance. In this way, we obtain the mapping ϕ of Definition 1. Finally, the class Event contains an attribute "action" which represents the message name (with this attribute, we easily obtain the mapping α of Definition 1).

2.2 Behavioral Aspects

We define a *behavioral aspect* as a pair $A = (P, Ad)$ of bSDs. P is a pointcut, i.e. a bSD interpreted as a predicate over the semantics of a base model satisfied

[5] We recall that we use the notion of *weak* sequential composition. It also exists a strong sequential composition. In a strong sequential composition of two bSDs M_1 and M_2, all the events of M_1 have to occur before an event of M_2 can occur.

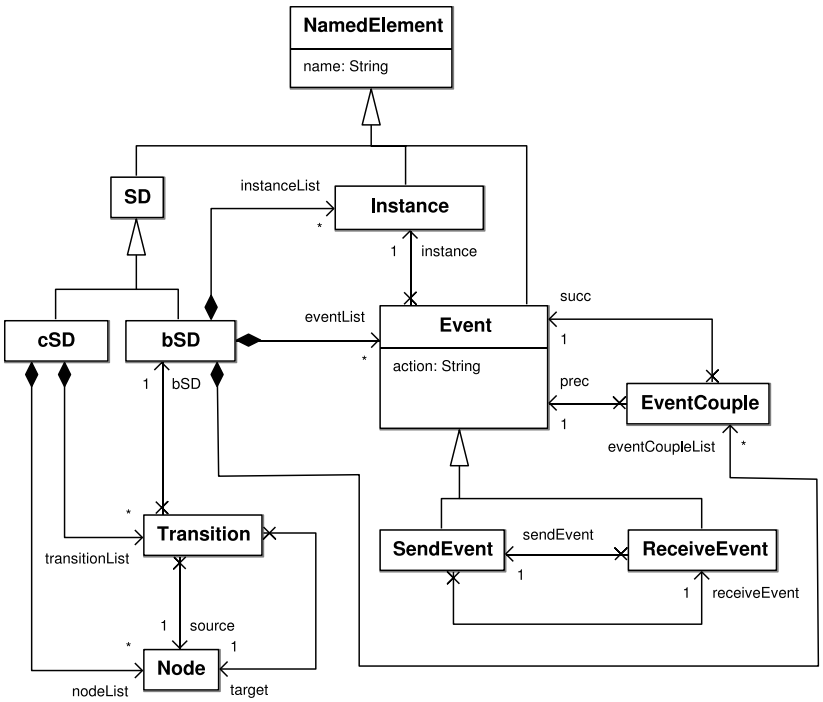


Fig. 3. Metamodel of SD

by all join points. *Ad* is an advice, i.e. the new behavior that should replace the base behavior when it is matched by *P*. Similarly to AspectJ, where an aspect can be inserted 'around', 'before' or 'after' a join point, we will show in the next sections that an advice may equally complete the matched behavior, replace it with a new behavior, or remove it entirely.

When we define aspects with sequence diagrams, we keep some advantages related to sequence diagrams. In particular, it is easy to express a pointcut as a sequence of messages. Figure 4 shows three behavioral aspects. The first allows the persistence of exchanges between the customer and the server. In the definition of the pointcut, we use regular expressions to easily express three kinds of exchanges that we want to save (the message *log in* followed by either the message *ok*, the message *try again*, or the message *no*). The second aspect allows the identification of a log in which fails. The third aspect allows the addition of a display and its update.

In Figure 1, the cSD *log in* represents a customer log in on a server. The customer tries to log in and either he succeeds, or he fails. In this last case, the customer can try again to log in, and either he succeeds, or the server answers "no". The expected weaving of the three aspects depicted in Figure 4 into the cSD *log in* is represented by the cSD in Figure 5.

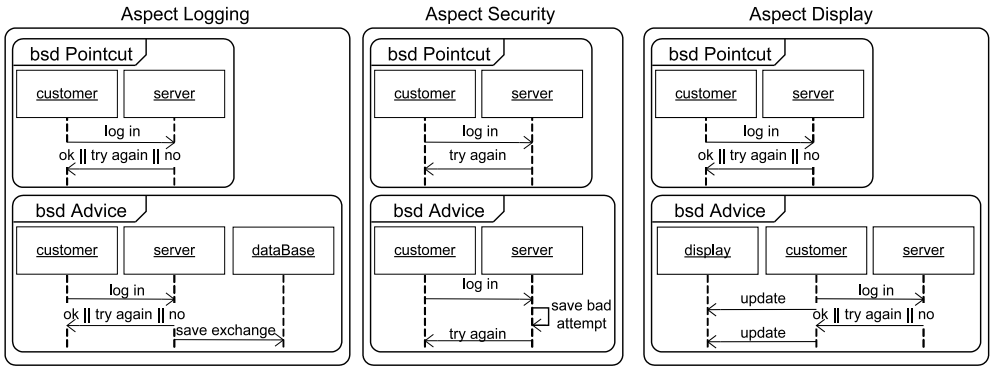


Fig. 4. Three behavioral aspects

3 Various definitions of join points

As mentioned in the introduction, weaving multiple aspects at the same join point can be difficult if a join point is simply defined as a strict sequence of messages, because aspects previously woven might have inserted messages in between. In this case, the only way to support multiple static weaving is to define each aspect in function of the other aspects, which is clearly not acceptable.

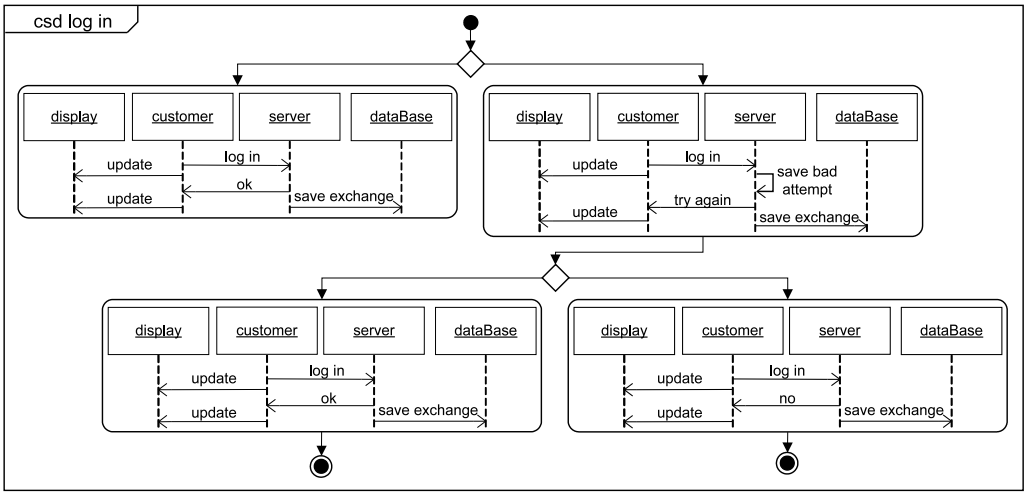


Fig. 5. Result of the weaving

The weaving of the three aspects depicted in Figure 4 allows us to better explain the problem. If the join points are defined as the strict sequence of messages corresponding to those specified in the pointcut, the weaving of these three aspects is impossible. Indeed, when the aspect *security* is woven, a message *save bad attempt* is added between the two messages *log in* and *try again*. Since the pointcut detects only a strict sequence of messages, after the weaving of the aspect *security*, the aspect *display* cannot be woven anymore. We obtain the same problem if we weave the aspect *display* first and the aspect *security* afterwards.

To solve this problem of multiple weaving, we introduce new formal definitions of join points which make possible the detection of join points where some events can occur between the events specified in the pointcut. In this way, when the aspect *security* is woven, the pointcut of the aspect *display* will allow the detection of the join point formed by the messages *log in* and *try again*, even if the message *save bad attempt* has been added.

In our approach, the definition of join point will rely on a notion of *part of a bSD*. A join point will be defined as a part of the base bSD such that this part corresponds to the pointcut. To define the notion of correspondence between a part and a pointcut, in Sub-Section 3.2, we introduce the notion of isomorphism between bSD. To define in a rigorous way the notion of join point, we also have to formally define the notion of part of a bSD. In Sub-Section 3.1, we propose four definitions for *parts of a bSD*, some of which allow the multiple weaving of aspects.

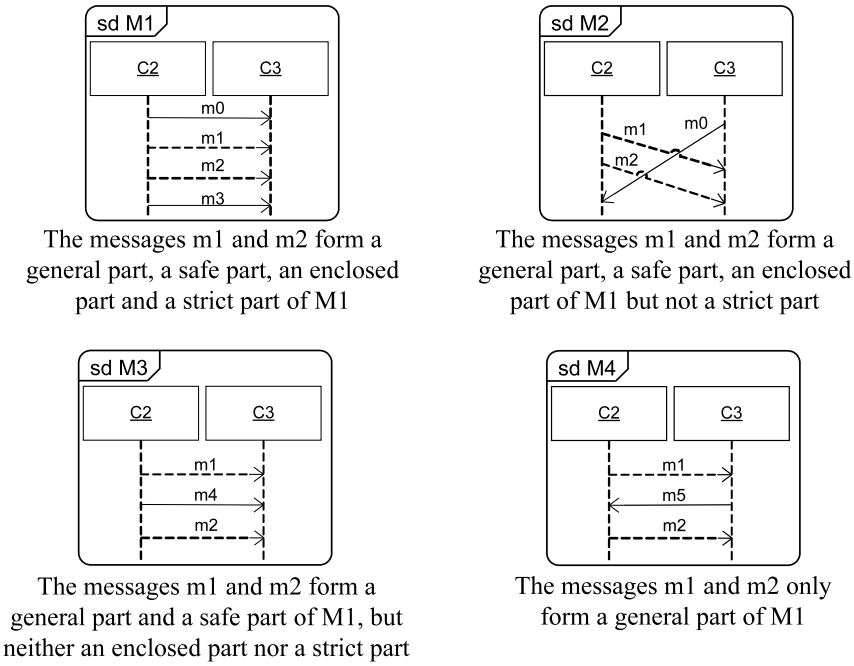


Fig. 6. Illustration of the notions of parts

3.1 Notion of Part of a bSD

We propose four definitions of parts of a bSD which allow the definition of four different types of join points. These definitions of parts will be called: *strict part, general part, safe part* and *enclosed part*. Before introducing formally the definitions of parts, we use Figure 6 to give a first intuitive idea of these parts.

Let us imagine we want to detect a message $m1$ followed by a message $m2$ from $C2$ to $C3$ in the bSDs in Figure 6. Firstly, it is clear that the messages $m1$ and $m2$ form a join point in the bSD $M1$, but it is not obvious that these two messages form a join point in the other bSDs because there is either a message which "surrounds" $m1$ and $m2$ (in $M2$), or a message between $m1$ and $m2$ (in $M3$ and $M4$).

We propose a first definition of part called *strict part* which only allows the detection of the message $m1$ and $m2$ in the bSD $M1$. This definition is the most restrictive, because with this definition, the wanted behavior can be presented in a bSD without to be detected when, for instance, it is surrounded by a message or when another message is present between the messages forming the wanted behavior.

Conversely, we propose a definition of part called *general part* which allows the detection of the message $m1$ and $m2$ in all the bSDs. This definition is the

less restrictive. Some messages can be present between the messages forming the wanted behavior.

We also propose one variant of strict part called *enclosed part*, and one variant of general part called *safe part*. An enclosed part allows the detection of the message $m1$ and $m2$ in the bSDs $M1$ and $M2$. As a strict part, an enclosed part allows the detection of a strict sequence of messages, but in addition, the sequence of messages can be surrounded by others messages as in the bSD $M2$. A safe part allows the detection of the message $m1$ and $m2$ in the bSDs $M1$, $M2$ and $M3$, i.e., a safe part allows the detection of a sequence of messages which is not necessarily a strict sequence of message, but unlike general part, the order on the events specified in a pointcut have to be preserved in a safe part (this last remark will be detailed afterwards).

Now, we formally introduce the four definition of parts. A strict part characterizing a strict sequence of messages can be defined by:

Definition 3 (Strict Part). *Let M be a bSD. We will say that M' is a strict part of M if there exist two bSDs X and Y such that $M = X \bullet M' \bullet Y$, $\bullet$ being the operator of sequential composition*[6]*.*

In Figure 6, the messages $m1$ and $m2$ form a strict part only into the bSD $M1$.

A general part, characterizing a part which can be "surrounded" by messages and where some messages can occur between the messages of the part, can be defined by:

Definition 4 (General Part). *Let $M = (I, E, \leq, A, \alpha, \phi, \prec)$ be a bSD. We will say that $M' = (I', E', \leq', A', \alpha', \phi', \prec')$ is a general part of M if:*

- $I' \subseteq I, \quad E' \subseteq E, \quad A' \subseteq A, \quad \alpha' = \alpha_{|E'}, \quad \phi' = \phi_{|E'}$;
- $\leq' \subseteq \leq_{|E'}, \quad \prec' = \prec_{|E'}, \quad \forall (e, f) \in \prec, e \in E' \Leftrightarrow f \in E'$.

In Figure 6, the messages $m1$ and $m2$ form a general part into all the bSDs.

A safe part allows the characterization of a join point where some events can occur between the events specified in the pointcut, if and only if the order of the events specified in the pointcut is preserved in the join points. A safe part can be formally defined by:

Definition 5 (Safe Part). *Let $M = (I, E, \leq, A, \alpha, \phi, \prec)$ be a bSD. We will say that $M' = (I', E', \leq', A', \alpha', \phi', \prec')$ is a safe part of M if:*

- *M' is a general part of M;*
- $\leq' = \leq_{|E'}$.

In Figure 6, the messages $m1$ and $m2$ form a safe part into the bSDs $M1$, $M2$ and $M3$. The order of the events of a safe part is the same as the order of the events of the initial bSD restricted to the events of the safe part ($\leq' = \leq_{|E'}$).

[6] Note that according to Definition 2, the sequential composition of two bSDs provides a bSD

That is why the messages $m1$ and $m2$ do not form a safe part into $M4$, because with only the messages $m1$ and $m2$, the receiving of the message $m1$ and the sending of the message $m2$ are not ordered whereas in the bSD $M4$, these two events are ordered (by transitivity) because of the message $m5$.

Finally, an enclosed part defines a strict sequence of messages but this sequence can be "surrounded" by others messages. More formally:

Definition 6 (Enclosed Part). *Let $M = (I, E, \leq, A, \alpha, \phi, \prec)$ be a bSD. We will say that $M' = (I', E', \leq', A', \alpha', \phi', \prec')$ is an enclosed part of M if:*

- *M' is a safe part of M;*
- *$pred_{\leq,E}(E') \cap succ_{\leq,E}(E') = E'$.*

In Figure 6, the messages $m1$ and $m2$ form an enclosed part into the bSDs $M1$ and $M2$. Since an enclosed part is a part where no event can be present between the events forming the enclosed part, the message $m1$ and $m2$ do not form an enclosed part into $M3$.

The set $pred_{\leq,E}(E') \cap succ_{\leq,E}(E')$, which represents the intersection between the set of predecessors of E' and the set of successors of E' [7], indicates the presence of events "between" the events of E'. Indeed, if an event $e \notin E'$ come between two events e' and e'' of M' ($e' \leq e \leq e''$ and $\phi(e') = \phi(e) = \phi(e'')$), then e belongs to $pred_{\leq,E}(E')$ and to $succ_{\leq,E}(E')$. Therefore $pred_{\leq,E}E' \cap succ_{\leq,E}(E') \neq E'$

Let us note that for the four proposed definitions of part of a bSD, the definitions are based on the semantics of the language of scenarios used, since we take account of the message names, but also of the partial order induced by the pointcut.

3.2 Join Point

Roughly speaking, a join point is defined as a part of the base bSD such that this part corresponds to the pointcut. Since we have defined four notions for parts of a bSD, we have four corresponding strategies for detecting join points. It remains to define the notion of correspondence between the pointcut and the part. To do so, we introduce the notions of morphisms and isomorphisms between bSDs.

Definition 7 (bSD Morphism). *Let $M = (I, E, \leq, A, \alpha, \phi, \prec)$ and $M' = (I', E', \leq', A', \alpha', \phi', \prec')$ be two bSDs. A bSD morphism from M to M' is a triple $\mu =< \mu_0, \mu_1, \mu_2 >$ of morphisms, where $\mu_0 : I \rightarrow I'$, $\mu_1 : E \rightarrow E'$, $\mu_2 : A \rightarrow A'$ and:*

$$(i)\ \ \forall (e,f) \in E^2, e \leq f \Rightarrow \mu_1(e) \leq' \mu_1(f) \qquad (iii)\ \ \mu_0 \circ \phi = \phi' \circ \mu_1$$
$$(ii)\ \ \forall (e,f) \in E^2, e \prec f \Rightarrow \mu_1(e) \prec' \mu_1(f) \qquad (iv)\ \ \mu_2 \circ \alpha = \alpha' \circ \mu_1$$

[7] Let us note that E' is necessarily inclued in $pred_{\leq,E}(E')$ and in $succ_{\leq,E}(E')$ because each event of E' is its own predecessor and its own successor ($e \leq e$, $\leq$ being reflexive by definition)

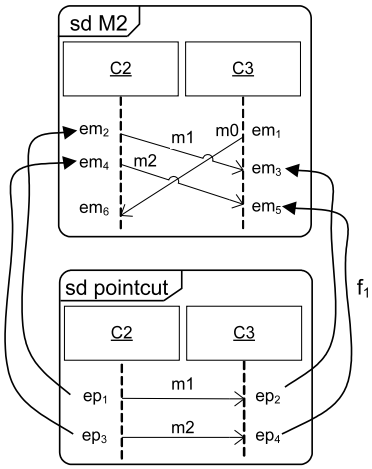


Fig. 7. Illustration of the notion morphism

Note that properties (i) and (ii) mean that by a bSD morphism the order and the type of the events are preserved (the type of an event is preserved means that, for instance, a sending event of M will be always associated with a sending event of M'). Note that property (iii) also means that all events located on a single lifeline of M are sent by μ_1 on a single lifeline of M'. Figure 7 shows a bSD morphism $f =< f_0, f_1, f_2 >: pointcut \rightarrow M2$ where only the morphism f_1 associating the events is represented (for instance, the event ep_1 which represents the sending of the message $m1$ is associated with the event em_2). Note that since each event of a bSD is unique, a bSD morphism f from a bSD M to a bSD M' always defines a unique part of M'.

Definition 8 (bSD isomorphism). *A bSD morphism $\mu = (\mu_0, \mu_1, \mu_2)$ from a bSD M to a bSD M' is an isomorphism if the three morphisms μ_0, μ_1, and μ_2 are isomorphic and if the converse morphism $\mu^{-1} = (\mu_0^{-1}, \mu_1^{-1}, \mu_2^{-1})$ is also a bSD morphism.*

With this definition of isomorphism, we can define the notion of join point in a general way:

Definition 9 (join point). *Let M be a bSD and P be a pointcut. Let M' be a part of M. We will say that M' is a join point if and only if there exists a bSD isomorphism $\mu = (\mu_0, \mu_1, \mu_2)$ from P to M' where the morphisms μ_0 and μ_2 are identity morphisms (P and M' have the same objects and action names).*

In a nutshell, for each definition of *a part of a bSD*, there is a corresponding definition of join point. In Figure 7, if we consider the *pointcut* depicted, it is easy to see that the messages $m1$ and $m2$ are a join point if we take the enclosed part, the safe part or the general part as definition of part of a bSD, because there exists a bSD isomorphism between the pointcut and an enclosed part, a safe part or a general part of $M2$.

3.3 Successive join points

To define the notion of successive join points the simple definition of join point is not precise enough. Indeed, in Figure 8, the pointcut $P1$ matches two different parts of $M1$, but these parts become entangled. Let us consider now the pointcut $P2$ and the bSD $M2$. If we take the definition of general part as the definition of part, there are four possible join points. Indeed, the first message a and the first message b can form a join point, as can the second message a and the second message b, but the first message a with the second message b or the second message a with the first message b can also form join points.

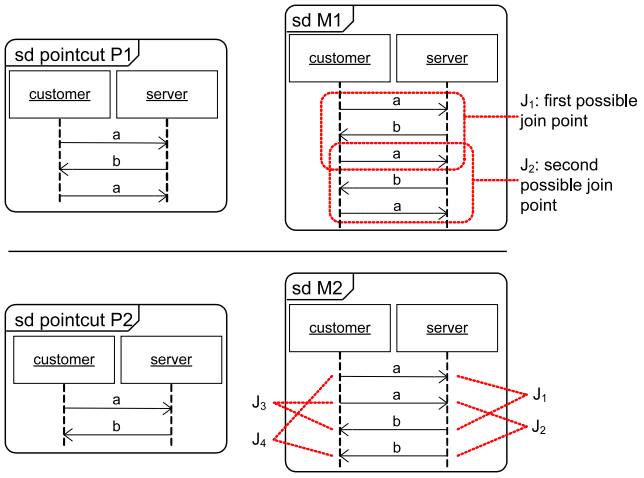


Fig. 8. Multiple possible matching

These multiple conflicting join points pose a problem. Among the conflicting join points, which should be chosen? Considering all these potential join points might not make sense from the point of view of someone using aspect weaving tools. In practice, we can reasonably expect that when a join point is detected, no elements of this join point can be used to form another join point. Roughly speaking, we can define successive join points as a set of disjoint join points taken in sequence.

The advantage of this definition is that it is safe in the sense that the weaving of the advice at a join point cannot conflict with the weaving of the advice at another join point. For instance, let us imagine that a weaving of an aspect removes a join point. If we weave this aspect into the bSD $M2$ in Figure 8, and if $J1$ is the first detected join point, then the messages a and b forming $J1$ are removed. In this case, the conflicting join points $J3$ and $J4$ have no meaning anymore since one of its constituent messages no longer exists.

However this answer is debatable. In the proposed example, $J1$ and $J2$ can form a join point because they don't share the same messages. The ideal solution

is perhaps to give the choice to the user by proposing several semantics of notion of successive join points. Nevertheless, in the sequel of this paper, we will only give a definition of the notion of successive join points which is (in an informal way) a sequence of disjoint join points. Other semantics of successive join points could be considered as interesting future work.

To define this sequence of disjoint join points, firstly we propose a way to order the parts of a bSD which are isomorphic to a pointcut in order to find the first join point matched by the pointcut. Then we show the first join point is always unique, because the order defined on the parts is a lattice. Secondly, we define successive join points in an inductive way by considering the first join point J which appears in a bSD M, and by continuing with M minus J.

Definition 10 (ordered parts). *Let $M = (I_M, E_M, \leq_M, A_M, \alpha_M, \phi_M, \prec_M)$ be a bSD and $P = (I_P, E_P, \leq_P, A_P, \alpha_P, \phi_P, \prec_P)$ be a pointcut. Let J_1 and J_2 be two parts of M such that there exist two bSD isomorphisms $f =< f_0, f_1, f_2 >: P \rightarrow J_1$ and $g =< g_0, g_1, g_2 >: P \rightarrow J_2$. We will say that J_1 precedes J_2 (or that J_2 succeeds J_1), denoted $J_1 \ll J_2$,if and only if:*

$$\forall e \in E_P \text{ such that } T(e) = send, f_1(e) \leq_M g_1(e).$$

In Figure 8, with this order we can say that the part J_1 precedes J_2 in bSD $M1$. We can also say that the part formed by the first message a and the first message b in the bSD $M2$ precedes all the other parts formed by a message a and a message b.

Afterwards, we will be especially interested in the *minimum* part of the order $\ll$, that is to say the part which precedes all the other parts. For a set $\mathcal{J}_{P,M}$ of all the parts of a bSD M isomorphic to a pointcut P, we will denote by $min(\mathcal{J}_{P,M})$ the minimum part. In the same way, if $\mathcal{J}_{P,M}$ is the set of all the join points of P in M, we will call the *minimum join point* the join point equal to $min(\mathcal{J}_{P,M})$. However, $\ll$ doesn't define a total order. For instance, in Figure 8, $J3$ and $J4$ are not ordered by $\ll$. Therefore, it is not obvious that $min(\mathcal{J}_{P,M})$ is unique. To demonstrate the uniqueness of $min(\mathcal{J}_{P,M})$, we show that $\ll$ is a lattice.

Theorem 1. *Let $\mathcal{J}_{P,M}$ be the set of join points of a bSD M corresponding to a pointcut P and let $\ll$ be the order on these join points as defined by Definition 10, then $(\mathcal{J}_{P,M}, \ll)$ is a lattice.*

The proof of this theorem is given in Appendix.

Now, we can inductively define successive join points as follows:

Definition 11 (Successive Join Points). *Let M be a bSD and P be a pointcut. Let $J_1, J_2, \ldots J_k$ be k parts of M isomorphic to P. These k parts are successive join points of P in M if:*

1. *J_1 is the minimum join point of P in M ;*
2. *$\forall i \in \{2 \ldots k\}, J_i$ is the minimum join point of P in M', M' being the bSD which contains the events of M minus the events of J_{i-1} and all the events which precede the events of J_{i-1}, so $M' = M - pred(J_{i-1})$.*

Taking the minimum join point every time guarantees the uniqueness of the successive join points. Roughly speaking, successive join points are detected in sequence at the earliest position where they appear in a bSD.

However the result $M' = M - pred(J_{i-1})$ is not always a well-formed bSD. Indeed, in Figure 9, the minimum join point J_1 of P in M is formed by the two first messages a and b. When we remove the events $pred(J_1)$ (the events of J_1 and the events which precede J_1), we have to remove the event corresponding to the sending of the message c. Therefore, the result $M' = M - pred(J_1)$ is only formed by the two last messages a and b, and the event corresponding to the reception of the message c. This is not really a problem because the algorithms proposed afterwards can be applied even if a bSD is of the kind of M'.

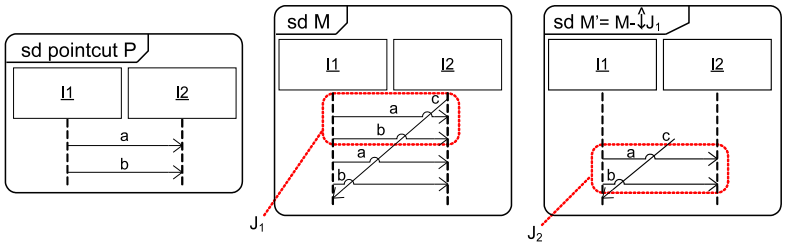


Fig. 9. Example of a not well-formed bSD

3.4 Which detection stategies should be chosen?

Each definition of part of a bSD presented in the previous sub-section leads to a specific join point detection strategy. This sub-section discusses some arguments for and against these strategies.

First, it is important to note that for the four proposed definitions of part of a bSD (so the four strategies), the definitions are based on the semantics of the language of scenarios used, since we take account of the message names, but also of the partial order induced by the pointcut.

The definition of strict part is the most restrictive, because with this definition, the wanted behavior can be presented in a bSD without being detected when, for instance, it is surrounded by a message. On the other hand, this definition is easy: we only search a decomposition of a base bSD M such that $M = M_1 \bullet J \bullet M_2$ (J being the join point). In [18], we have showed that this simplicity allows the achieving of good decidability results for join point detection in infinite scenarios.

Conversely, the definition of general part is the least restrictive. Some messages can be present between the messages forming the wanted behavior. This fact can imply the detection of join points interleaved with behaviors not expected by the user. Moreover, the partial order defined by the pointcut is not necessarily preserved in the detected join point. The major advantage of a join point defined as a general part remains the property to easily weave several aspects at the same join point.

The definitions of enclosed part and safe part combine the advantages and the drawbacks of a strict part and a general part. An enclosed part looks like a strict part, but it can be surrounded by some messages. Therefore, if we want to look for a strict sequence of messages in a certain way, this definition seems to be appropriate. However, an enclosed part has the drawback that it does not tolerate the weaving of multiple aspects at the same join point. If we want to weave several aspects at the same join point, while the partial order defined by the pointcut is preserved, the definition of safe part seems to be appropriate. However, a safe part has the drawback for the detection of join points interleaved with behaviors not expected by the user, because some messages can be present between the messages of the join points.

Despite this short discussion on the advantage and the drawbacks of each definition, the main interest of the proposed approach is that a user can choose as he/she wishes the semantics of the weaving in finite scenarios. The user is free to choose the definition of part which suits him/her the better by adapting the algorithm of detection according to the chosen definition. We will show how this flexibility can be easily implemented with the Kermeta environment in Section 6.

4 Join point detection

In [18], Klein et al. propose an algorithm to detect a strict part, i.e. a strict sequence of messages. In this paper we propose three new algorithms to detect join points defined as an *enclosed part*, a *general part*, or a *safe part* of a bSD.

Firstly, in Sub-Section 4.1, we introduce a general algorithm which contains two "abstract functions" *findSetsOfEvent* and *min*. Secondly, in Sub-Section 4.2, we show how these functions can be specialized for each notion of join points to obtain the three new algorithms.

4.1 General Algorithm

Algorithm 1 allows the construction of an isomorphism $\mu = (\mu_0, \mu_1, \mu_2)$ from a pointcut P to a part M' of a bSD M, such that μ_0 and μ_2 are identity morphisms. In this way, the isomorphism indicates the first join point M' in M. We denote by $\pi_i(M) \subseteq E_M$ the projection of a bSD M on an object i of M and by $\pi_E(M)$ the restriction of a bSD M to a subset $E \subseteq E_M$. Moreover, we use a function β_E which, for an event e of E, gives the position of e on the object containing e. More specifically, the position of an event on an object is defined by the number of events which precede it on this object: $\forall e \in E, \beta_E(e) = card(\{e' \in E | \phi(e) = \phi(e') \wedge e' \leq e\})$. Finally, we introduce the function $\mathit{\Gamma}_{E,o}(n)$ which gives the event of E localized on the nth position on the object o ($\mathit{\Gamma}_{E,o}(n) = e$ such that $\beta_E(e) = n \wedge \phi(e) = o$).

For all objects of a pointcut P, the first part of the algorithm (line 1 to 4) allows the construction of the sets of events of M localized on the same object, such that the actions related to these events are the same as the actions related to the events of P. The variable w_i represents a word of all events on an object i of the base bSD M. With the function $findSetsOfEvent$, we take, for each

object i, all the set of (strict or non-strict) sequence of events of M which have the same action names as the events of P on the object i. Since the decision to take a strict sequence of events or a non-strict sequence of events depends on the definition of parts, the function $findSetsOfEvent$ has to be detailed for each definition of parts.

The second part of the algorithm (line 5 to 13) allows the construction of a part M' of M when it is possible. After the first part of the algorithm, with the function min, we take the first (or minimum) set of events forming a part. Since we propose four definitions of parts, this function min has to be specified for each definition of parts. The notion of minimum set of events or minimum parts is the one defined in the previous section (related to the definition of ordered parts, Definition 10).

An example of how the algorithm works in a practical way is given in the following sub-section.

Note that to detect successive join points in a base bSD M, we start to apply Algorithm 1 on M to obtain the first join point, which we denote J_1 (more precisely, we obtain an isomorphism $\mu = (\mu_0, \mu_1, \mu_2)$ which defines J_1). Secondly, we apply Algorithm 1 on $M' = M - pred(J_1)$ to obtain the second join point J_2, and then we continue in this way as long as the last join point obtained is not null.

Algorithm 1 Abstract Algorithm of Join Point Detection (P,M)

input:pointcut $P = (I_P, E_P, \leq_P, , A_P, \alpha_P, \phi_P, \prec_P)$,
bSD $M = (I_M, E_M, \leq_M, A_M, \alpha_M, \phi_M, \prec_M)$
output:$\mu = (\mu_0, \mu_1, \mu_2) : P \rightarrow M'$,$M' = (I_{M'}, E_{M'}, \leq_{M'}, A_{M'}, \alpha_{M'}, \phi_{M'}, \prec_{M'})$ join point of M

1: For each $i \in I_P$ do
2: $w_i = \pi_i(M)$ `/* a word of all events on the object i */`
3: $V_i = findSetsOfEvent(w_i, \pi_i(P))$
4: End For
5: $E_{M'} = min(\cup_{i \in I_P} V_i)$
6: If $(E_{M'} = \emptyset)$ then
7: return($null$)
8: Else
9: μ_0 is the identity isomorphism from I_P to $\phi_M(E_{M'})$,
10: μ_2 is the identity isomorphism from A_P to $\alpha_M(E_{M'})$,
11: μ_1 is the isomorphism from E_P to $E_{M'}$ such that $\forall e \in E_P$, $\mu_1(e) = \Gamma_{v_{\phi(e)},\phi(e)} \circ \beta_{E_P}(e)$ `/* for each object` o `of` I_p`,` μ_1 `is built by associating with the event of` o `in the ith position, the event belonging to` $E_{M'}$ `on` o `in the ith position.*/`
12: return($\mu = (\mu_0, \mu_1, \mu_2)$)
13: End If

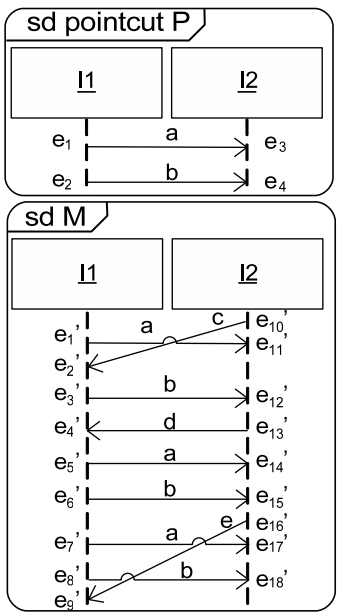
4.2 Specialization of the Abstract Algorithm

Enclosed Part Detection

For the detection of enclosed part, the function $findSetsOfEvent$ is equivalent to $V_i = \{v \in E_M^* \mid \exists u, w, w_i = u.v.w \wedge \alpha(v) = \alpha(\pi_i(P))\}$. For a word of events w_i on the object i, the function $findSetsOfEvent$ returns a set V_i where each element v of V_i is a strict sequence of events which have the same action names as the events of P on the object i.

With the function min, it remains to check if the order of the events of P is the same as the order of the events associated to M'. For that, we check if for all pairs of sending-reception of events of P, the events of M' at the same position also form a pair of sending-reception of events. Then, we take the first (or minimum) set of events satisfying the properties. More formally, the function min can be rewritten by:

$$min\{v_1, \ldots, v_{|I_P|} \in V_1 \times \cdots \times V_{|I_P|} | \\ \forall (e, f) \in \prec_P, \left(\Gamma_{v_{\phi(e)},\phi(e)} \circ \beta_{E_P}(e), \Gamma_{v_{\phi(f)},\phi(f)} \circ \beta_{E_P}(f)\right) \in \prec_M\}$$



variable	value
$\alpha(\pi_{I1}(P))$	$\alpha(e_1 e_2) = ab$
$\alpha(\pi_{I2}(P))$	$\alpha(e_3 e_4) = ab$
w_{I1}	$e_1' e_2' e_3' e_4' e_5' e_6' e_7' e_8' e_9'$
w_{I2}	$e_{10}' e_{11}' e_{12}' e_{13}' e_{14}' e_{15}' e_{16}' e_{17}' e_{18}'$
V_{I1}	$\{e_5' e_6'; e_7' e_8'\}$
V_{I2}	$\{e_{11}' e_{12}'; e_{14}' e_{15}'; e_{17}' e_{18}'\}$
$E_{M'}$	$\{e_5' e_6'; e_{14}' e_{15}'\}$

Fig. 10. Illustration of the general algorithm using the enclosed part strategy

In Figure 10, with the pointcut P and the bSD M, we are going to show how Algorithm 1 works in a practical way if the function $findSetsOfEvent$ and min are defined as above. The table in Figure 10 represents the values of some variables used in the algorithm. The two first variables $\alpha(\pi_{I1}(P))$ and $\alpha(\pi_{I2}(P))$ (used in line 3 of the algorithm) represent respectively the label of the projection

of P on the objects $I1$ and $I2$. These two labels are equal to ab. The two next variables w_{I1} and w_{I2} (in Figure 10, in table) represent the projection of M on respectively the objects $I1$ and $I2$ (computed in line 2 of the algorithm). Then, for I_1, with the function $findSetsOfEvent$, the algorithm computes the sets of successive events of w_{I1} which have the same action names (or labels) as $\alpha(\pi_{I1}(P))$. We obtain $V_{I1} = \{e'_5e'_6; e'_7e'_8\}$ since the labels of $e'_5e'_6$ and $e'_7e'_8$ are equal to ab. We do the same for I_2 and we obtain $V_{I2} = \{e'_{11}e'_{12}; e'_{14}e'_{15}; e'_{17}e'_{18}\}$. At line 5, with the function min, the algorithm computes the first (or minimum) set of events which form an enclosed part. The first set of events is $\{v_{I1} = e'_5e'_6; v_{I2} = e'_{11}e'_{12}\}$, but it does not satisfy the properties of line 5. Indeed,

$$\forall(e_1, e_3) \in \prec_P,\ \big(\Gamma_{v_{\phi(e_1)},\phi(e_1)} \circ \beta_{E_P}(e_1), \Gamma_{v_{\phi(e_3)},\phi(e_3)} \circ \beta_{E_P}(e_3)\big) = \big(\Gamma_{v_{I1},I1}(1), \Gamma_{v_{I2},I2}(1)\big) = (e'_5, e'_{11}) \notin \prec_M.$$

The set of events $\{v_{I1} = e'_5e'_6; v_{I2} = e'_{14}e'_{15}\}$ is the first set which satisfies the properties, so $E_{M'} = \{e'_5e'_6; e'_{14}e'_{15}\}$. The rest of the algorithm builds the isomorphism $\mu = (\mu_0, \mu_1, \mu_2)$ from P to the bSD formed by the events of $E_{M'}$.

Safe Part Detection

For the detection of safe part, the function $findSetsOfEvent$ is equivalent to

$$V_i = \{v = x_1.x_2...x_k \in E^*_M \mid \exists u_i \in E^*_M, i \in \{1...k+1\},\ w_i = u_1.x_1.u_2.x_2...u_k.x_k.u_{k+1} \wedge \alpha(v) = \alpha(\pi_i(P))\}.$$

In this way, we can detect a join point even if there are some events (represented by the u_i) between the events of the join point. Let us note that for $i \in \{1...k+1\}$, u_i can contain no event.

The function min looks like the one defined for the detection of enclosed pattern, but in addition, we also have to check if the order of the events of $E_{M'}$ is the same as the order of E_M restricted to the event of $E_{M'}$ (we check if $\leq_{M'} = \leq_{M|E_{M'}}$), because the fact that we allow the presence of other events between the events of a general part can introduce a difference between $\leq_{M'}$ and $\leq_{M|E_{M'}}$. Formally:

$$min\{v_1, \ldots, v_{|I_P|} \in V_1 \times \cdots \times V_{|I_P|} \mid \forall(e, f) \in \prec_P, \big(\Gamma_{v_{\phi(e)},\phi(e)} \circ \beta_{E_P}(e), \Gamma_{v_{\phi(f)},\phi(f)} \circ \beta_{E_P}(f)\big) \in \prec_M \wedge \leq_{M'} = \leq_{M|E_{M'}}\}$$

General Part Detection

For the detection of general part, the function $findSetsOfEvent$ is the same as the one used for the detection of safe part.

The function min is similar to the one used for the detection of safe part, except for one difference. According to the definition of a general part, it is not necessary to check whether $\leq_{M'} = \leq_{M|E_{M'}}$. So, in the function min, this checking is not performed (let us note that the property $\leq_{M'} \subseteq \leq_{M|E_{M'}}$ is always verified).

5 Operator of composition

Now that we can detect join points in a base bSD, it remains to compose the bSD Advice with the join points. In [18], they use the notion of strict part to define the join points. If we note by J the join point and by B the base

bSD, by definition, there exist two bSDs B_1 and B_2 such that we can write $B = B_1 \bullet J \bullet B_2$ ($\bullet$ being the operator of sequential composition). If we note Ad the advice representing the expected behavior, all you have to do to compose the advice with the join point is to replace the join point by the advice, and the woven bSD is $B = B_1 \bullet Ad \bullet B_2$.

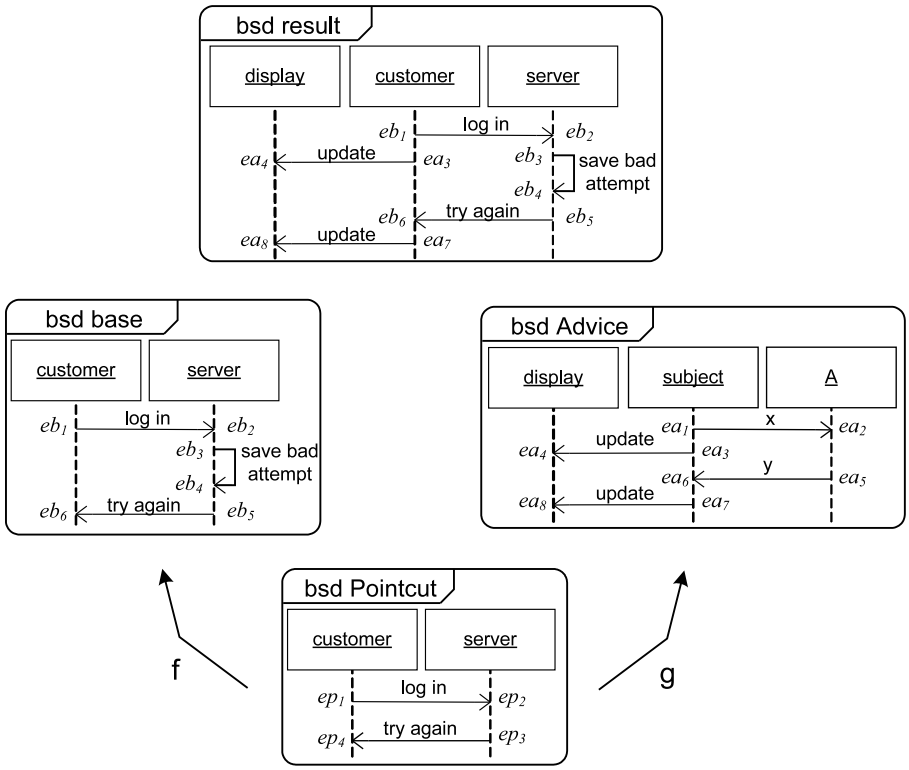
When we use the notions of general part, safe part or enclosed part to define the join points, the composition of the advice is not so easy. Indeed, with these kinds of join points, some messages can surround a join point or some messages can be present between the messages forming the join point. In these cases, it is not possible to simply replace a join point by an advice because the result cannot be always expressed with the standard operators of composition such as the sequential composition operator. Therefore, we have to define a new operator of composition which takes into account the common parts between a join point and an advice to produce a new bSD which does not contain copies of similar elements of the two operands. We propose an operator of composition for bSDs called *left amalgamated sum*. This sum is inspired by the amalgamated sum proposed in [17]. We add the term $left$ because our operator is not commutative, but it imposes a different role on each operand.

Figure 11 shows an example of left amalgamated sum where the two bSDs $base = (I_b, E_b, \leq_b, A_b, \alpha_b, \phi_b, \prec_b)$ and $advice = (I_a, E_a, \leq_a, A_a, \alpha_a, \phi_a, \prec_a)$ are amalgamated. For that, we use a third bSD which we call bSD $pointcut = (I_p, E_p, \leq_p, A_p, \alpha_p, \phi_p, \prec_p)$ and two bSD morphisms $f : pointcut \rightarrow base$ and $g : pointcut \rightarrow advice$ which allow the specification of the common parts of the two bSDs *base* and *advice*. Moreover, f has to define an enclosed part, a safe part or a general part M' in the bSD *base* such that f is an isomorphism from the *pointcut* to M'. We can note that the morphism f is automatically obtained with the process of detection described into the previous section.

The morphism g, which indicates the elements shared by the advice and the pointcut, has to be specified when the aspect is defined. In this way, g allows the specification of abstract or generic advices which are "instantiated" by the morphism. For instance, it is not mandatory that the advice contains objects having the same name as those present in the pointcut. In the three aspects in Figure 4, the morphism g is not specified but it is trivial: for each aspect, we associate the objects and the actions having the same names, and the events corresponding to the actions having the same name. The advice of the aspect Display in Figure 4 could be replaced by the "generic" Advice in Figure 11. It is the morphism g which indicates that the object *customer* plays the role of the object *subject* and that the object *server* plays the role of the object A.

In Figure 11, the elements of the bSDs *base* and *advice* having the same antecedent by f and g will be considered as identical in the bSD *result*, but they will keep the names specified in the bSD *base*. For instance, the objects *subject* and A in the bSD *advice* are replaced by the objects *customer* and *server*. All the elements of the bSD *base* having an antecedent γ by f such that γ has not an image by g in the bSD *advice* are deleted. This case does not appear in the example proposed, but in this way we can delete messages of the

bSD *base*. For instance, in an amalgamated sum, if the right operand (the bSD advice in the example) is an empty bSD then the part of the left operand which is isomorphic to the *pointcut* (that is to say the join point), is deleted. Finally, all the elements of the bSDs *base* and *advice* having no antecedent by f and g are kept in the bSD *result*, but the events of the bSD *advice* will always form a "block" around which the events of the bSD *base* will be added. For instance, in Figure 11, in the bSD *base*, if there were an event e on the object *customer* just after the message *try again*, then this event e would be localized just after the sending of the message *update* (event ea_7) in the woven SD.



$f=(f_0,f_1,f_2)$: Pointcut → base, where
- *f_0: $I_P \rightarrow I_b$ is the identity*
- *f_1 respectively sends ep_1, ep_2, ep_3 and ep_4 to eb_1, eb_2, eb_5 and eb_6*
- *f_2: $A_p \rightarrow A_b' \subset A_b$ is the identity*

$g=(g_0,g_1,g_2)$: Pointcut → advice, where
- *g_0: $I_P \rightarrow I_a' \subset I_a$ sends customer to subject and server to A*
- *g_1 respectively sends ep_1, ep_2, ep_3 and ep_4 to ea_1, ea_2, ea_5 and ea_6*
- *g_2: $A_p \rightarrow A_a' \subset A_a$ sends log in to x and try again to y*

Fig. 11. An example of left amalgamated sum

Formally, a left amalgamated sum is defined by:

Definition 12 (left amalgamated sum). *Let* $M_0 = (I_0, E_0, \leq_0, A_0, \alpha_0, \phi_0, \prec_0)$, $M_1 = (I_1, E_1, \leq_1, A_1, \alpha_1, \phi_1, \prec_1)$ *and* $M_2 = (I_2, E_2, \leq_2, A_2, \alpha_2, \phi_2, \prec_2)$ *be three bSDs. Let* $f =< f_0, f_1, f_2 >: M_0 \rightarrow M_1$ *and* $g =< g_0, g_1, g_2 >: M_0 \rightarrow M_2$ *be two bSDs morphisms such that* $f(M_0)$ *defines a part* M_1' *of* M_1 *and that* f *is a isomorphism from* M_0 *to* M_1'*. The left amalgamated sum of* M_1 *and* M_2 *is the bSD* $M = M_1 +_{f,g} M_2$ *where* $M = (I, E, \leq, A, \alpha, \phi, \prec)$ *is defined by:* $I = I_1 \cup \{i_2 \in I_2 | \nexists i_0 \in I_0, g_0^{-1}(i_2) = i_0\}$*;*

$E = \{e_1 \in E_1 | \exists e_0 \in E_0, \exists e_2 \in E_2, f_1^{-1}(e_1) = e_0 \wedge g_1(e_0) = e_2\} \cup \{e_1 \in E_1 | \nexists e_0 \in E_0, f_1^{-1}(e_1) = e_0\} \cup \{e_2 \in E_2 | \nexists e_0 \in E_0, g_1^{-1}(e_2) = e_0\}$*;*

$$\leq = \left(\begin{array}{l} \{(e_1, e_2) \in (E_1 \cap E)^2 | e_1 \leq_1 e_2\} \cup \\ \{(e_1, e_2) \in (E_2 \cap E)^2 | e_1 \leq_2 e_2\} \cup \\ \{(e_1, e_2), e_1 \in (f_1(E_0) \cap E), e_2 \in (E_2 \cap E) | \\ \qquad \exists e_2' \in E_2, e_2' = g_1 \circ f_1^{-1}(e_1) \wedge e_2' \leq_2 e_2\} \cup \\ \{(e_1, e_2), e_1 \in (E_2 \cap E), e_2 \in (f_1(E_0) \cap E) | \\ \qquad \exists e_2' \in E_2, e_2' = g_1 \circ f_1^{-1}(e_2) \wedge e_1 \leq_2 e_2'\} \cup \\ \{(e_1, e_2), e_1 \in \big(pred_{<_1, E_1} f_1(E_0) - f_1(E_0)\big), e_2 \in (E_2 \cap E) | \\ \qquad \phi(e_1) = \phi(e_2)\} \cup \\ \{(e_1, e_2), e_1 \in (E_2 \cap E), e_2 \in \big(succ_{<_1, E_1} f_1(E_0) - f_1(E_0)\big) | \\ \qquad \phi(e_1) = \phi(e_2)\} \end{array} \right)^*$$

$$\forall e \in E, \alpha(e) = \begin{cases} \alpha_1(e) \ \textit{if } e \in E_1 \\ \alpha_2(e) \ \textit{if } e \in E_2 \end{cases};$$

$$\forall e \in E, \phi(e) = \begin{cases} \phi_1(e) \ \textit{if } e \in E_1 \\ \phi_2(e) \ \textit{if } e \in E_2 \end{cases};$$

$A = \alpha(E)$*;*

$\prec = (\prec_1 \cup \prec_2) \cap E^2$

The first line of the definition of $\leq$ means that each pair of events of E_1 present in E and ordered by $\leq_1$ remains ordered by $\leq$. The second line is equivalent but for the events of E_2. The third line means that an event e_1 of E_1 present in E precedes an event e_2 of E_2 present in E, if there exists an event e_2' of E_2 preceding e_2 and corresponding to e_1 in M_0. The fourth line means that an event e_2 of E_1 present in E succeeds an event e_1 of E_2 present in E, if there exists an event e_2' of E_2 succeeding e_1 and having the same antecedent as e_2 in M_0. Finally, the fifth line means that an event e_1 of E_1 preceding the part detected in M_1, will precede all event e_2 of E_2 if e_1 and e_2 are localized on the same object in M. The last line is equivalent but for the events of E_1 which succeed the detected part.

Let us note that this operator of composition can lead to some situations where there are several possibilities to order the events. For instance, in Figure 11, let us suppose that the messages *update* in the bSD *advice* are sent by the object A instead of the object *customer*. Then, when we compose the bSD *base* with the bSD *advice*, the sending of the message *update* and the message *save bad attempt* cannot be ordered. In this case, it is the designer who has to specify the expected order.

6 Implementation with Kermeta

To apply the detection and composition algorithms proposed in this paper on practical examples, we have implemented them within the Kermeta environment. This section is divided in three sub-sections. The first one presents the Kermeta environment and details our motivations for using it. The second details how the weaving process is implemented, and the third presents the use of our weaver from a user perspective.

6.1 The Kermeta environment

Kermeta [19] is an open source meta-modeling language developed by the Triskell team at IRISA. It has been designed as an extension to the EMOF 2.0 to be the core of a meta-modeling platform. Kermeta extends EMOF with an action language that allows specifying semantics and behaviors of metamodels. The action language is imperative and object-oriented. It is used to provide an implementation of operations defined in metamodels. As a result the Kermeta language can, not only be used for the definition of metamodels but also for implementing their semantics, constraints and transformations.

The Kermeta action language has been specially designed to process models. It includes both Object Oriented (OO) features and model specific features. Kermeta includes traditional OO static typing, multiple inheritance and behavior redefinition/selection with a late binding semantics. To make Kermeta suitable for model processing, more specific concepts such as opposite properties (i.e. associations) and handling of object containment have been included. In addition to this, convenient constructions of the Object Constraint Language (OCL), such as closures (e.g. each, collect, select), are also available in Kermeta.

A complete description of the way the language was defined can be found in [19]. It was successfully used for the implementation of a class diagram composition technique in [25] but also as a model transformation language in [20]. To implement the detection and composition techniques proposed in this paper we have chosen to use Kermeta for two reasons. First, the language allows implementing composition by adding the algorithm in the body of the operations defined in the composition metamodel. Second, Kermeta tools are compatible with the Eclipse Modeling Framework (EMF) [5] which allows us to use Eclipse tools to edit, store, and visualize models.

6.2 The weaving process as model transformations

As detailed previously, the weaving process consists of two steps. Firstly, the detection step uses the pointcut model and the base model to compute a set of join points. Each join point is characterized by a morphism from the pointcut to a corresponding elements in the base model. Secondly, using these morphisms, the advice is composed with each join point in the base model. The first step processes models to extract join points and the second is a model transformation. Figure 12 details the input and output models of these two steps (each ellipse

is a model and the black rectangle on the top left-hand corner indicates its metamodel). Except for morphisms, all models are SDs.

The first step to process or transform models in Kermeta is the definition of the input and output metamodels. Thanks to the compatibility of Kermeta with Eclipse tools, we have used Omondo UML [22] which provides a graphical editor for metamodels in addition to UML editors. Figure 3 presents the simple metamodels we are using for SDs. We use this sequence diagram metamodel rather than that of UML2.0 for two major reasons. Firstly, as shown in Section 2, the metamodel in Figure 3 fits very well with the formal definitions introduced in this paper. So, the metamodel is relatively small, concise and easy to understand, and the algorithms presented in this paper are easier to write with this metamodel rather than with that of UML2.0. Secondly, it is very simple to write a transformation from the UML2.0 sequence diagram metamodel to the metamodel in Figure 3 because the concepts are very close. So, we can apply the weaving on a model compliant to the UML2.0 metamodel by performing a transformation from the UML2.0 sequence diagram metamodel to the metamodel in Figure 3 before the weaving process.

Once the metamodel is defined this way, EMF provides generic tools to create, edit and save instance models. Kermeta allows, on one hand to complete the metamodel with the specification of the bodies of operation and on the other hand to process models created with EMF. We used the same process to define a simple metamodel to represent morphisms. This metamodel contains only one class called Morphism which encapsulates associations between, instances, messages and events of two SDs.

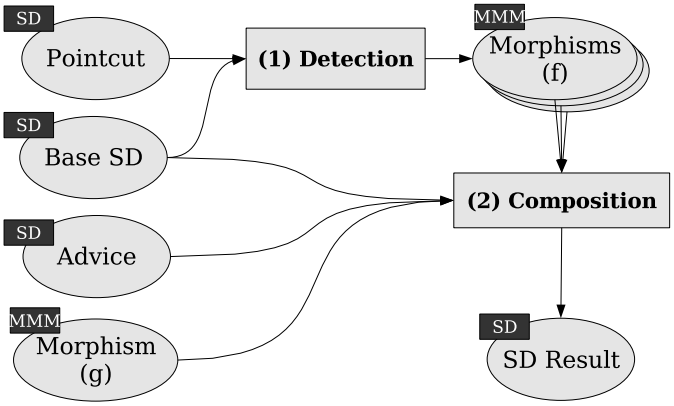


Fig. 12. Transformation of Models

Using the metamodels for SDs and morphisms, we have designed and implemented the complete weaving process. For the detection transformation we have defined a Kermeta abstract class Detection and three sub-classes to implement

the different detection strategies. The composition is implemented in a single Kermeta class.

Both the implementation of detection algorithms and the implementation of the composition operator were used to validate the techniques proposed in this paper. The composition was implemented first and tested by providing test cases composed of a base scenario, a pointcut scenario, an aspect scenario, and the morphisms between the pointcut and the advice and between the pointcut and the base scenario. We chose the set of test cases to intuitively cover the structures of sequence diagrams such as messages between two instances, messages on a single instance, crossing messages or cSD with alternatives, etc. For all test cases, we checked manually that the composed models correspond to the expected models. The implementation of the detection algorithms was tested using various simple scenarios corresponding to detection and non-detection cases. We especially had to test the detection algorithms in situations where several potential matches could be chosen. In addition to the testing of each step of the weaving, we applied our prototype tool on several small academic examples.

6.3 Using the prototype tool

This goal of this section is to present the use of our weaving technique from a user perspective.

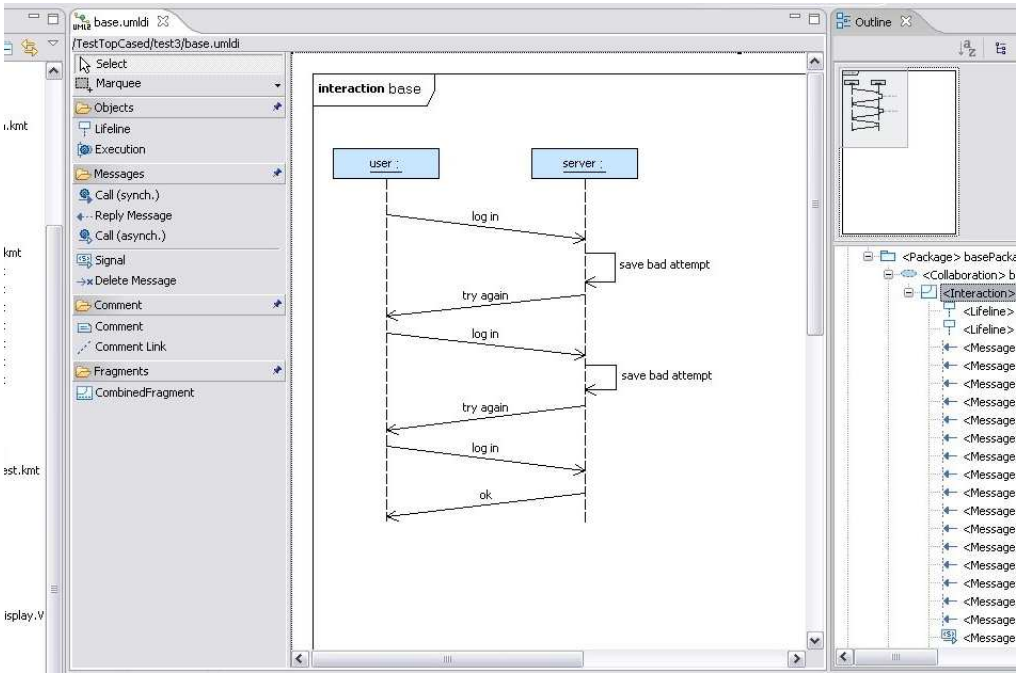


Fig. 13. Screenshot of the base scenario

First, the developer has to model the base scenario of his/her application. To do so we use the UML 2.0 sequence diagram editor available in the TopCaseD eclipse plugin [27]. Figure 13 presents a screenshot of this editor with a base model. The base model consists of an interaction between two instances names *user* and *server*. Figure 14 presents the two scenarios of a behavioral aspect to weave in the base model. The pointcut and advice are presented respectively at the top and at the bottom of the figure. This goal of this aspect is to update a *display* object whenever a *customer* object sends a *log in* message or receives a response from the *server*.

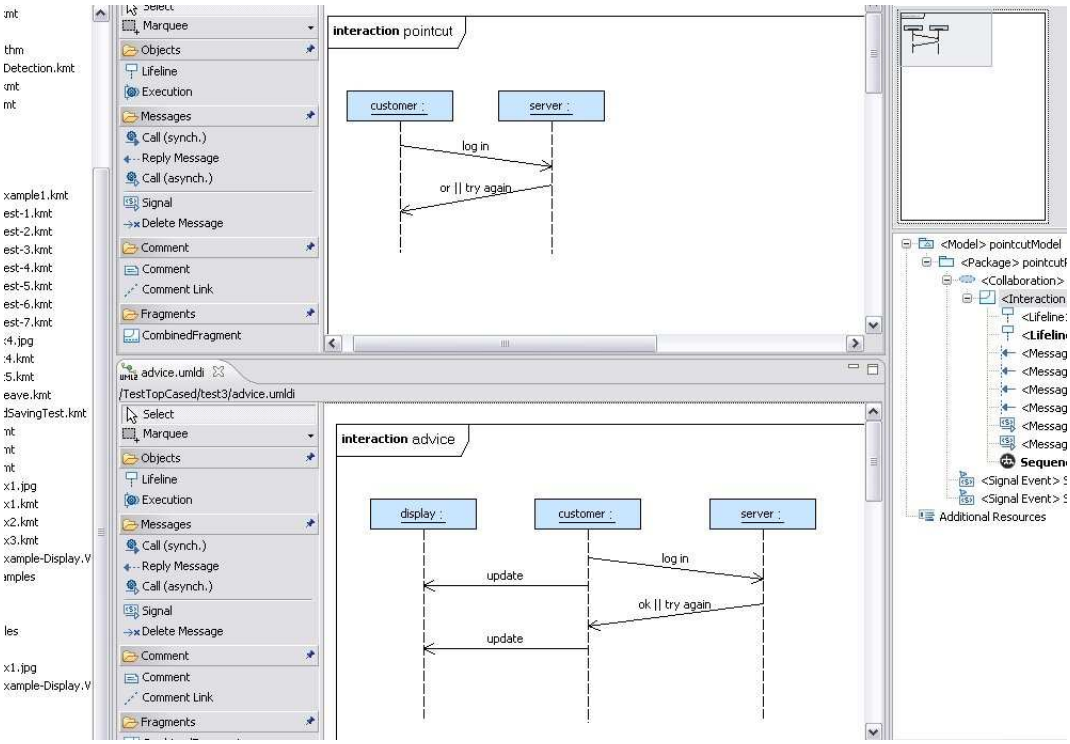


Fig. 14. Screenshot of the aspect scenarios

Once the scenarios for both the base model and the behavioral aspect are defined, a wizard can be used to perform the weaving. Figure 15 presents a screenshot of this wizard. To apply our weaving algorithms, the user has to provide the scenarios corresponding to the pointcut and advice and specify in which base models the weaving should be applied. In addition to this, the user can choose the detection strategy to use. If the *strict sequence of messages* is selected then the detection strategy corresponds to the notions of *strict part* and *enclosed part* of a bSD. The check-box *allow surrounding messages* allows

choosing between these two strategies. It the *non-strict sequence of messages* is selected, then the notions of *safe part* and *general part* of a bSD are used. The check box *preserve event order* allows choosing between these two strategies. After choosing the detection strategies, the weaving can be performed at once using the *Weave All* button or interactively using the *Weave* and *Skip* buttons.

Fig. 15. Screenshot of the weaving wizard

Figure 16 presents the result of the weaving of the behavioral aspect in the base model of figure 13, with as settings in the wizard, "Non-strict message sequence" and "Preserve event order" selected.

7 Future Works

The algorithms of join point detection proposed in this paper (when the join points are enclosed parts, safe parts or general parts of a bSD) only work for bSDs or combined SDs which generate a finite number of behaviors (cSDs without loop, in this case the weaving can be applied to each bSDs of the set generated by a cSD). When the join points are strict parts of a bSD, the join point detection within infinite behavior is already solved in [18]. More specifically, the detection of join points within infinite behaviors always terminates when the pointcut is connected, i.e., when the pointcut has no parallel component (the pointcut cannot be written as a parallel composition of two other bSDs). However for the

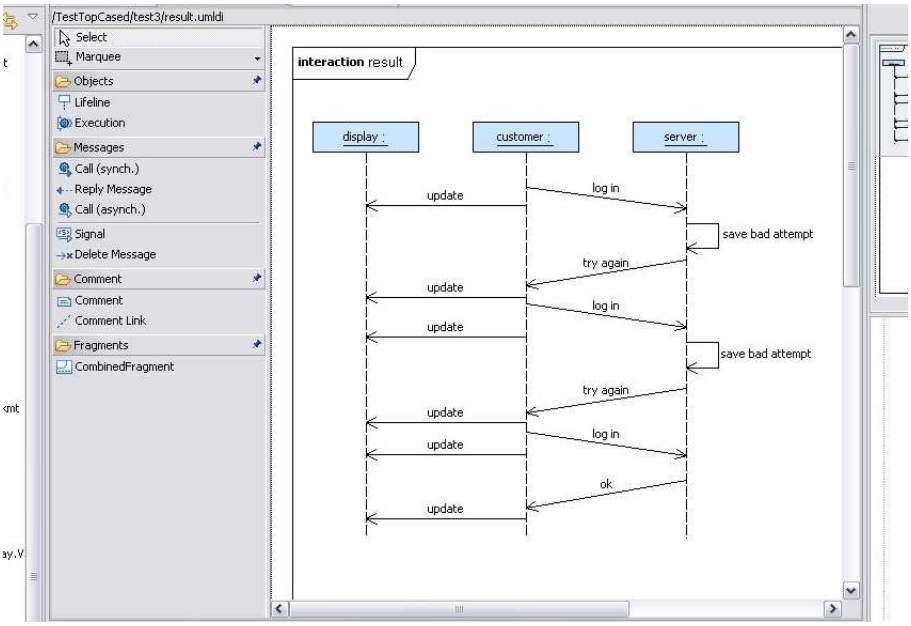


Fig. 16. Screenshot of the result

new definitions of join points proposed in this paper, the problem of detection is more complicated. For instance, let us consider the behavioral aspect and the cSD *example* depicted in Figure 17. When the join points are general parts of a bSD, the join point formed by the messages a and b is detected in each behavior generated by the cSD *example*. So, the expected behaviors allows any number of messages c between a and b. Since the message d surrounds this potentially infinite number of messages c, the expected behaviors cannot be represented with SDs (we cannot isolate anymore the message c in a loop).

When we consider the join points as general parts, safe part or enclosed part, our future works are to identify the cases for which our static weaving is always possible, even if the base scenario generates an infinite number of behaviors.

In the paper we have chosen to limit the approach to simple name matching. However, in future work, our approach could be extended with more powerful matching mechanisms such as roles or wildcards on object names.

8 Related Works

Clarke and Baniassad [7] use the *Theme/UML approach* to define aspects. Theme/UML introduces a theme module that can be used to represent a concern at the modeling level. Themes are declaratively complete units of modularization, in which any of the diagrams available in the UML can be used to model one view of the structure and behavior the concern requires to execute. In Theme/UML,

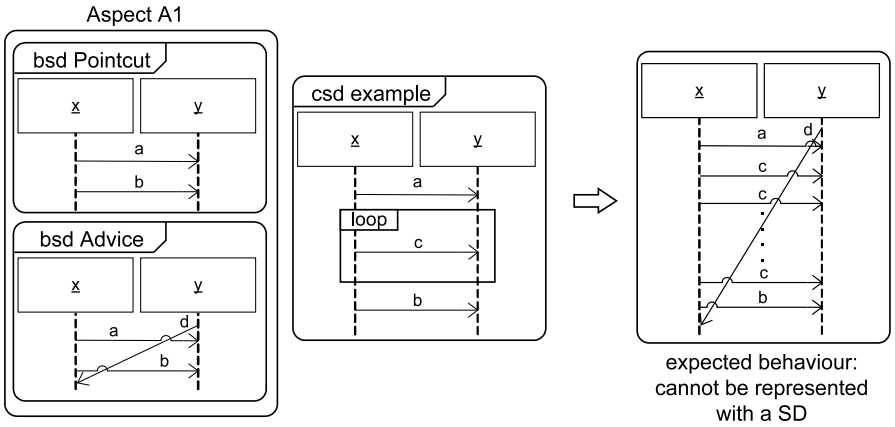


Fig. 17. Impossible weaving

a class diagram and sequence diagrams are typically used to describe the structure and behaviors of the concern being modeled. The question addressed by their work is more the specification of aspects than the weaving process into non-aspectual models, but our definitions and detection of join point, and our operator of composition can easily be adapted to the Theme approach to keep the advantages of the two approaches.

Similarly to our approach, Whittle and Araujo ([29] and [2]) represent behavioral aspects with scenarios. Aspectual scenarios are modeled as interaction pattern specifications (IPSs introduced in [12]) and are composed with specification scenarios. The weaving process defined in [29] composes scenarios with instantiation and ad-hoc composition operators. The weaving process proposed by [2] is performed in two steps. The first step is to generate state machines from the aspects and from the specification. The weaving process is then a composition of these state machines. However, in these two approaches, the detection of join points is not automated: users have to specify them with a binding relation between an aspect and a specification. Moreover, their composition operator for SDs is not defined formally.

In [26], Stein et al. introduce a way to express various conceptual models of pointcuts in aspect-oriented design. But, they do not provide a way to detect the join points specified by these pointcuts. By contrast, in our approach the detection of the join points and their composition is automatic.

More generally, in [10] and [11], Douence et al. are interested in event pattern matching, which is close to our approach. A significant difference is that they use a monitor to perform event pattern matching at runtime on a program execution, whereas our weaving is made statically at a modeling level. Similar to Douence et al., Walker and Viggers [28] have proposed *declarative event patterns* as a means to specify patterns of events to detect sequence of events in the execution of a system based on a context-free-language-based pattern matching, while Allan et

al. [1] have proposed a new history-based language feature called *tracematches* that enables the programmer to trigger the execution of extra code by specifying a regular pattern of events in a computation trace. Our approach differs from both in that we allow the entire pattern (join point) to be replaced or completed, rather than just the final event in the pattern. We can do that because our weaving is static. We do not perform the weaving during the execution of the sequence diagram, but we transform a sequence diagram into another sequence diagram where the aspect is woven.

Still at a programming level, recently Bockisch et al. [4] have proposed a novel implementation of the mechanism of cflow present in AspectJ for which the efficiency of join point detection for dynamic weaving is improved. However, it is only applicable for the detection of sequence of messages in the control flow of a method, whereas with our approach, we can detect any interactions. Moreover, since our weaving is static, performance is not a primary issue.

The aspect model and in particular the mechanism to identify join points plays a critical role in the applicability of the aspect-oriented methodology. According to Kiczales [15], the pointcuts definition language probably has the most relevant role in the success of the aspect-oriented technology but most of the solutions proposed so far are too tied to the syntax of the programs manipulated.

Ostermann et al. [23] try to address this problem by proposing a static joint point model that exploits information from different models of program semantics. They show that this model of joint points increases the abstraction level and the modularity of pointcuts.

9 Conclusion

In this paper we have proposed a technique to statically weave behavioral aspects into sequence diagrams. Our weaving process is automated, and takes into account the semantics of the model used, i.e., the partial order that a SD induces.

To enable the weaving of multiple aspects, we have proposed a new interpretation for pointcuts to allow join points to match them more flexibly. However, with this new way of specifying join points, the composition of the advice with the detected part could not any longer be a replacement of the detected part by the advice. We thus had to consider the events (or the messages) of the join point which are not specified within the pointcut and merge them with the behavior specified within the advice. We proposed a formal definition for such a merge operator, and described its implementation on the Kermeta platform. Moreover, we have presented the use of our weaving technique from a user perspective.

However, our approach suffers from limitations: our algorithms for join point detection only work for bSDs or combined SDs which generate a finite number of behaviors. This has to be considered for further research.

References

1. C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching

with free variables to aspectj. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, volume 40, pages 345–364. ACM Press, 2005.
2. J. Araujo, J. Whittle, and Kim. Modeling and composing scenario-based requirements with aspects. In *Proceedings of RE 2004*, Kyoto, Japan, September 2004.
3. E. Aspect, 2006. http://www.early-aspects.net/.
4. C. Bockisch, S. Kanthak, M. Haupt, M. Arnold, and M. Mezini. Efficient control flow quantification. In *OOPSLA '06: Proceedings of the 21th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, volume 41, pages 125–138. ACM Press, 2006.
5. F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. Grose. *Eclipse Modeling Framework*. The Eclipse Series. Addison Wesley Professional, 2003.
6. S. Clarke. *Composition of Object-Oriented Software Design Models*. PhD thesis, Dublin City University, 2001.
7. S. Clarke and E. Baniassad. *Aspect-Oriented Analysis and Design: The Theme Approach*. Number ISBN: 0-321-24674-8. Addison Wesley, 2005.
8. W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. volume 19, pages 45–80, 2001.
9. B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge Mathematical Textbooks, 1990.
10. R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of GPCE'02*, LNCS. Springer, 2002.
11. R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Reflection'01*, pages 170–186, 2001.
12. R. B. France, D.-K. Kim, S. Ghosh, and E. Song. A uml-based pattern specification technique. *IEEE TSE, vol.30(3), 193-206, March 2004*, 2004.
13. ITU-TS. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)*. ITU-TS, Geneva, September 1999.
14. I. Jacobson and P.-W. Ng. *Aspect-Oriented Software Development with Use Cases*. Addison-Wesley, 2004.
15. G. Kiczales. The fun has just begun. Keynote of AOSD'03, 2003.
16. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.
17. J. Klein, B. Caillaud, and L. Hélouët. Merging scenarios. In *Workshop on FMICS*, pages 209–226, Linz, Austria, sep 2004.
18. J. Klein, L. Hélouet, and J.-M. Jézéquel. Semantic-based weaving of scenarios. In *AOSD*, Bonn, Germany, 2006. ACM.
19. P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. Weaving executability into object-oriented meta-languages. In *Proc. of MODELS/UML*, LNCS, Jamaica, 2005.
20. P.-A. Muller, F. Fleurey, D. Vojtisek, Z. Drey, D. Pollet, F. Fondement, P. Studer, and J.-M. Jézéquel. On executable meta-languages applied to model transformations. In *Model Transformations In Practice Workshop*, Jamaica, 2005.
21. OMG. Uml superstructure, v2.0. OMG Document number formal/05-07-04, 2005.
22. Omondo, 2006. http://www.omondo.com.
23. K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *Proceedings of ECOOP'05*. Springer LNCS, 2005.
24. A. Rashid, A. M. D. Moreira, and J. Araújo. Modularisation and composition of aspectual requirements. In *proceedings of AOSD'03*, pages 11–20, 2003.
25. R. Reddy, R. France, S. Ghosh, F. Fleurey, and B. Baudry. Model composition - a signature-based approach. In *AOM Workshop*, Montego Bay, Oct. 2005.

26. D. Stein, S. Hanenberg, and R. Unland. Expressing different conceptual models of join point selection in aspect-oriented design. In *AOSD*, Bonn, Mars 2006.
27. TopCaseD, 2006. http://www.topcased.org/.
28. R. J. Walker and K. Viggers. Implementing protocols via declarative event patterns. *In ACM Sigsoft International Symposium on Foundations of Software Engineering (FSE-12)*, 29(6):159–169, 2004.
29. J. Whittle and J. Araújo. Scenario modelling with aspects. *IEE Proceedings - Software*, 151(4):157–172, 2004.

Appendix

This appendix contains the proof of Theorem 1.

Proof:

To demonstrate Theorem 1, we assume that two overtaking messages cannot have the same name and we use the following lemma which can be found in the book "*Introduction to Lattices and Order*" [9](p.110):

Lemma 1. *Let $(L, \vee, \wedge)$ be a triple where L is a non-empty set equipped with two binary operations $\vee$ and $\wedge$ which satisfy for all $a, b, c \in L$:*

- (1) $a \vee a = a$ *and* $a \wedge a = a$ *(idempotency laws);*
- (2) $a \vee b = b \vee a$ *and* $a \wedge b = b \wedge a$ *(commutative laws);*
- (3) $(a \vee b) \vee c = a \vee (b \vee c)$ *et* $(a \wedge b) \wedge c = a \wedge (b \wedge c)$ *(associative laws);*
- (4) $a \vee (a \wedge b) = a$ *et* $a \wedge (a \vee b) = a$ *(absorption laws).*

then:

- *(i)* $\forall a, b \in L, a \vee b = b \Leftrightarrow a \wedge b = a$*;*
- *(ii) If we define* $\leq$ *by* $a \leq b$ *if* $a \vee b = b$*, then* $\leq$ *is an order relation;*
- *(iii) With* $\leq$ *as in (ii),* $(L, \leq)$ *is a lattice such that* $\forall a, b \in L, a \vee b = sup\{a, b\}$ *and* $a \wedge b = inf\{a, b\}$*.*

■

We will show that $\mathcal{J}_{P,M}$ can be equipped with two binary operations $\vee$ and $\wedge$ which verify the properties 1 to 4 of the lemma.

Let $\mathcal{J}_{P,M}$ be the set of join points corresponding to a pointcut $P = (I_P, E_P, \leq_P, A_P, \alpha_P, \phi_P, \prec_P)$ in a bSD $M = (I, E, \leq, A, \alpha, \phi, \prec)$. Let $\vee$ and $\wedge$ be the operators defined for each J_i, J_j of $\mathcal{J}_{P,M}$ by:

$$\begin{aligned} J_i \vee J_j = & \{e, f \in J_i | e \prec f, \exists e' \in E_P, e = \mu_{i_1}(e'), \mu_{j_1}(e') \leq e\} \cup \\ & \{e, f \in J_j | e \prec f, \exists e' \in E_P, e = \mu_{j_1}(e'), \mu_{i_1}(e') \leq e\} \end{aligned}$$

$$\begin{aligned} J_i \wedge J_j = & \{e, f \in J_i | e \prec f, \exists e' \in E_P, e = \mu_{i_1}(e'), e \leq \mu_{j_1}(e')\} \cup \\ & \{e, f \in J_j | e \prec f, \exists e' \in E_P, e = \mu_{j_1}(e'), e \leq \mu_{i_1}(e')\} \end{aligned}$$

$\mu_i =< \mu_{i_0}, \mu_{i_1}, \mu_{i_2} >$ and $\mu_j =< \mu_{j_0}, \mu_{j_1}, \mu_{j_2} >$ being the isomorphisms associating P to the respective join points J_i and J_j.

For $(\mathcal{J}_{P,M}, \vee, \wedge)$, the properties (1) and (2) of the lemma are verified (trivial). Let J_i, J_j and J_k be three join points and $\mu_i =< \mu_{i_0}, \mu_{i_1}, \mu_{i_2} >$, $\mu_j =< \mu_{j_0}, \mu_{j_1}, \mu_{j_2} >$ and $\mu_k =< \mu_{k_0}, \mu_{k_1}, \mu_{k_2} >$ the three isomorphisms associating respectively P to J_i, J_j and J_k. Let e and f be two events of M such that $e \prec f$. If e and f belong to J_i and $(J_i \vee J_j) \vee J_k$, let e' be the corresponding event in P such that $e = \mu_{i_1}(e')$, then according to the definition of $\vee$, e succeeds to $\mu_{j_1}(e')$ and $\mu_{k_1}(e')$. Therefore, e and f also belong to $J_i \vee (J_j \vee J_k)$. In this way, we easily show that $(J_i \vee J_j) \vee J_k = J_i \vee (J_j \vee J_k)$. In the same way, we also show that $(J_i \wedge J_j) \wedge J_k = J_i \wedge (J_j \wedge J_k)$. Finally, to prove the property (4), let us consider the two join points J_i and J_j and their associated morphisms μ_i and μ_j. Let e_2 and f_2 be two events belonging to J_j and $J_i \vee (J_i \wedge J_j)$ (and consequently to $J_i \wedge J_j$) but not to J_i. Let us note e' the event belonging to P such that $e_2 = \mu_{j_1}(e')$. If $e_1 = \mu_{i_1}(e')$, then since e_2 belongs to $J_i \wedge J_j$, $e_2 \leq e_1$, and since e_2 belongs to $J_i \vee (J_i \wedge J_j)$, $e_1 \leq e_2$. Impossible, therefore all the events of $J_i \vee (J_i \wedge J_j)$ belong to J_i.

According to the lemma, $(\mathcal{J}_{P,M}, \ll')$, with $\ll'$ defined by $J_i \ll' J_j$ if $J_i \vee J_j = J_j$, is a lattice. Moreover $\ll'$ is equivalent to the order $\ll$ of Definition 10. The equivalence is easy to demonstrate. Let J_i and J_j be two join points, and μ_i and μ_j their associated isomorphisms to P. If $J_i \ll' J_j$, by definition $J_i \vee J_j = J_j$, and thus all the message send events of J_j succeed those of J_i. The converse is trivial.

$\boxdot$