UNIVERSITY OF OSLO
Department of Informatics

Weaving of UML Sequence Diagrams using STAIRS

Research Report 367

Roy Grønmo, Fredrik Sørensen, Birger Møller-Pedersen, Stein Krogdahl

ISBN 82-7368-325-7
ISSN 0806-3036

October 2007

Weaving of UML Sequence Diagrams using STAIRS

Roy Grønmo,Fredrik Sørensen,
Birger Møller-Pedersen, Stein Krogdahl

October 2007

Abstract

In this report we explore aspect-oriented modeling for UML 2.0 sequence diagrams. We ensure that the aspect weaving is semantics-based by using a formal trace model for sequence diagrams. A major challenge is to handle unbounded loops which produce infinite traces. We establish a systematic way to permutate and rewrite the original loop definition so that the weaving in many typical cases can be performed on a finite structure. We prove that it is always sufficient to consider a loop with upper bound relative to the pointcut definition to discover if the loop has infinitely repeating matches. A running example illustrates the approach and a prototype weaving tool is being implemented.

1 Introduction

Aspect-orientation for programming has emerged as a promising way to separately define cross-cutting parts of programs, in order to achieve separation of concern. We believe that the same potential is there also for modeling. This report explores aspect-oriented modeling for UML 2 sequence diagrams [9].

In aspect-oriented programming the `base program` is the main program upon which one or more aspects may define some cross-cutting code as additions or changes. An aspect is defined by a pair (pointcut and advice), where the `pointcut` defines where to affect the base program and the corresponding `advice` defines what to do in the places identified by the pointcut. Analogously we term our main sequence models as the `base models`, and we define aspect diagrams consisting of pointcut and advice diagram both based upon the concrete syntax of sequence diagrams.

If the models are only used for documentation and illustration of the system behaviour, then we do not need to actually weave the aspect model and the base model. However, within model-driven development the models are used actively to produce the implementation, test programs or to maintain relationships between different models. In such an environment we need to weave the aspect and the base model. The aspect diagram defines the cross-cutting model

elements to influence the base model, so that an aspect weaver can produce a woven result in the form of a new model.

Many aspect-oriented approaches suffer because they rely on a pure syntactic pointcut matching and weaving. This forces the aspect developer to think in terms of how the program or model is organized syntactically. With a semantics-based approach the aspect developer shall be able to conceptually express when and how to apply the aspect.

We illustrate the problem of syntactic-based pointcut matching with an example (Figure 1). The advice diagram is left out since it is irrelevant for the matching process. There is a pointcut expressing that the message $m1$ from $L1$ to $L2$ is followed by the message $m2$ from $L1$ to $L2$. The base model has two consecutive *alt* operators defined using the combined fragment notation of UML sequence diagrams. An *alt* operator defines a choice of different alternatives, where the alternatives are given as operands separated by a dashed line. If we apply the poincut on the base model of the figure with pure syntactic-based matching, then we will not find any matches. However, one possible execution trace will choose the second operands of the two *alt* operators which then should result in an exact match of the specified pointcut.

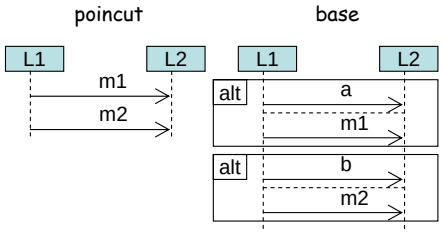

Figure 1: Syntactic-based matching problem

Grosu and Smolka [4] show how an automata-based representation can be used to formalize UML 2.0 sequence diagrams, so that semantics-based weaving is possible. However, this is restricted to *bounded sequence diagrams*. Unbounded sequence diagrams occur when there are loops involved and when an execution run cannot be described by a regular expression. An example of an unbounded sequence diagram is shown in Figure 2a (example taken from [4]). The problem is that this sequence diagram produces traces of the form $(!m)^n(?m)^n$ which is not a regular expression. An unbounded sequence diagram must produce infinite traces meaning that there must be at least one unbounded loop. It is however not a sufficient criterion to be unbounded that it contains an unbounded loop as we see in Figure 2b. The unbounded diagram produces traces that can be desribed as a regulal expression: $(!m1, ?m1, !m2, ?m2)^*$

We avoid the limitation of only using bounded sequence diagrams by using STAIRS [10] traces as our formal model for UML 2 sequence diagrams. The syntactic sequence definition is by STAIRS translated into a set of traces that represent the set of possible execution runs. By working on the STAIRS

traces instead of the syntactic model we achieve a semantics-based matching and weaving. This will allow the aspect modeller to think in terms of *what an aspect should capture* instead of *how the model may be syntactically defined*.

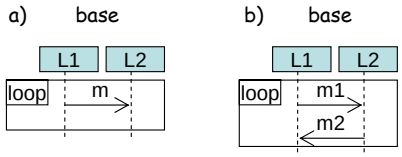

Figure 2: a)unbounded b) bounded

In the remainder of this report we will use a base model example with a loop as the control node. The base model example (Figure 3) is adopted from Klein et al. ([6]) who used a similar example for Message Sequence Charts. The example has been translated into an equivalent version for UML sequence diagrams. The model starts with a *login* attempt from *Customer* to *Server*. At the end the *Server* will finally answer with an *ok* message to indicate successfull login. In between these two events there may be zero or more iterations of a loop. The loops first message, *tryAgain*, informs of login failure, while the second message, *newAttempt*, is a new customer login attempt.

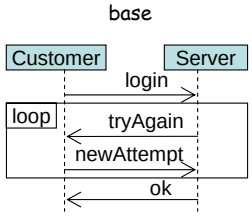

Figure 3: Login example

This report is organized as follows; Section 2 presents the STAIRS formal model for sequence diagrams; Section 3 proposes aspect diagrams for sequence diagrams; Section 4 shows how we achieve a semantics-based matching; Section 5 defines how to calculate weave instructions given an aspect diagram as input; Section 6 defines how to reverse-engineer a set of traces into a sequence diagram, which is an important part of the weaving process; Section 7 presents the full weave algorithm; Section 8 shows how to deal with the problematic unbounded loops that lead to infinite traces; Section 9 illustrates the approach by five examples; Section 10 shows a tool which implements the approach; Section 11 contains related work; Section 12 discusses the approach; and finally Section 13 concludes the report.

2 STAIRS

In order to do a semantics-based pointcut matching and semantics-based weaving, we need to define a semantic model for sequence diagrams. For this purpose we choose to use the well established STAIRS formal model where the semantics of a sequence diagram is understood as a set of execution traces. Both the trace set and individual traces may be finite or infinite. First we define the syntax for interactions by the EBNF in Figure 4 (adopted and simplified from Runde et al. [10]).

<Interaction>	⟶	<Empty> | <Event> | <Weak Seq> | <Alternatives> | <Loop>
<Empty>	⟶	skip
<Event>	⟶	<Kind> <Message>
<Kind>	⟶	<Transmission> | <Reception>
<Transmission>	⟶	!
<Reception>	⟶	?
<Message>	⟶	(Signal, <Transmitter>, <Receiver>)
<Transmitter>	⟶	Lifeline
<Receiver>	⟶	Lifeline
<Alternatives>	⟶	alt [<Interaction list>]
<Loop>	⟶	loop Set [<Interaction list>]
<Weak seq>	⟶	seq [<Interaction list>]
<Interaction list>	⟶	<Interaction> | <Interaction list> , <Interaction>

Figure 4: Syntax for sequence diagrams

The syntax model is only a subset of those given by Runde et al. We focus on the operators *seq*, *alt* and *loop*. The first two operators are chosen because they are the basic operators from which we also may define several other operators. The *loop* is included since it provides some challenges in the context of semantics-based weaving. In this report we restrict the loop body to use only the *seq* operator and not *alt*s or inner *loop*s. Notice that the *loop* operator has an optional integer *Set* to define the possible iterations of the loop. This can be expressed in many different ways such as UML cardinality $lowerBound..upperBound$, explicit enumeration $\{1, 3, 5\}$ (if omitted it corresponds to $0..*$). $lowerBound..upperBound$ means that the loop will iterate either $lowerBound$, or $lowerbound + 1$,..., or $upperBound$ iterations.

Now we return to the example model to see how it can be represented with the textual syntax ($C = Customer, S = Server$):

$$
\begin{array}{ll}
seq[& !(login, C, S), ?(login, C, S) \\
 & loop[!(tryAgain, S, C), ?(tryAgain, S, C), \\
 & \quad !(newAttempt, C, S), ?(newAttempt, C, S)], \\
 & !(ok, S, C), ?(ok, S, C)]
\end{array}
$$

Each message is represented by two events, a transmission event (!) and a reception event (?). In some of our examples in this report, we omit the transmitter and receiver lifelines for readability when this information is unambigously defined by the diagram. We require that an interaction representing a base diagram as well as the pointcut and advice diagram (defined in the next section) are well-formed, meaning that both the transmission and reception event of a message must be included when the corresponding transmitter and receiver lifelines are present. Furthermore, we require that both lifelines are present. The syntax model is however recursively defined, so that inner interactions may be defined without well-formedness requirements, e.g. the interaction $seq\,[!(m, A, B)]$.

In STAIRS the semantics of a sequence diagram is defined by a function $[\![i : Interaction]\!] = (p, n)$, where p is the set of positive traces and n is the set of negative traces. Positive traces define valid behaviour and negative traces define invalid behaviour, while all other traces are defined as inconclusive. The negative traces does not seem to raise particular issues upon the approach taken in this report. Thus, we consider only positive traces in this report, and we will simplify the semantic function as $[\![i : Interaction]\!] = p$.

Figure 5 shows the STAIRS definitions we need in this report. The semantics of *skip* is the empty trace, and the semantics of a single event is the single trace with the event itself. The semantics of the syntactic *seq* operator is the semantics of its operand for a single interaction, while it uses the semantic weak sequencing operator, $\gtrsim$, between all trace sets. The $\gtrsim$ operator will produce a set of traces that follow the principle that orderings are only given for each lifeline and its events in isolation.

An event takes place on a lifeline $l1$ if it is a transmission event $!(signal, l1, l2)$ or a reception event $?(signal, l2, l1)$. An intuitive idea behind the semantic definitions of the weak sequencing is that messages are sent asynchronously and that they may happen in any order on different lifelines, but sequentially on the same lifeline.

The semantics of the *alt* operator is the union of the semantics of each individual operand. The *loop* operator is defined as the union of the semantics of the loop with all possible fixed numbers of iterations. The semantics of a loop with a fixed number fix iterations is defined recursively so that the semantics is equal to the loop operand semantics repeated fix times with the weak sequencing operator ($\gtrsim$) between each. If the loop has no upper bound, then we will have infinitely long traces.

The semantic rules we have defined can be used to calculate the set of traces for a sequence diagram defined with the textual syntax. The resulting set of traces of the base model of the running example is:

$$\{\langle !login, ?login, !ok, ?ok\rangle,$$
$$\langle !login, ?login, !tryAgain, ?tryAgain,$$
$$!newAttempt, ?newAttempt, !ok, ?ok\rangle, \ldots\}$$

The first trace corresponds to zero iterations in the loop. The next corresponds to one iteration and there will be infinitely many traces since the loop has no upper bound. The subsequence ?*tryAgain*, ?*tryAgain*, !*newAttempt*, ?*newAttempt* can be repeated an arbitrary number of times.

To avoid some problems in the matching process described later, we will extend the STAIRS syntax and semantics models by assigning a unique identifier (*id*) to each message in a trace. The same identifier is shared between the corresponding transmission event (!) and reception event (?) of the message:

$$\langle Message\rangle \rightarrow (Signal, \langle Transmitter\rangle, \langle Receiver\rangle, \mathbf{id}).$$

sdef.1 ⟦skip⟧ = { <> }
sdef.2 ⟦e: Event⟧ = { <e> }
sdef.3 ⟦seq [i: Interaction]⟧ = ⟦i⟧
sdef.4 ⟦seq [il: InteractionList, i: Interaction]⟧ = ⟦seq [il]⟧ $\succsim$ ⟦i⟧
sdef.5 ⟦alt [i_1, ...i_n: Interaction]⟧ = {⟦i_1⟧, .. ⟦i_n⟧ }
sdef.6 ⟦loop [i: Interaction]⟧ = {<>} $\uplus_{\forall\ n>0}$ ⟦loop {n} [i]⟧
sdef.7 ⟦loop {n..m} [i: Interaction]⟧ = $\uplus_{k\in\{n..m\}}$ ⟦loop {k} [i]⟧
sdef.8 ⟦loop {n} [i: Interaction]⟧ = ⟦loop {n-1} [i]⟧ $\succsim$ ⟦i⟧, n > 0
sdef.9 ⟦loop {0} [i: Interaction]⟧ = { <> }

Here, t1: TraceSet $\succsim$ t2: TraceSet is defined as the TraceSet with one concatenated trace for each combination of one trace from t1 and one trace from t2 where:

- events on a lifeline l in t1 comes before events on the same lifeline l in t2
- events on different lifelines may come in any order

Figure 5: Semantics for sequence diagrams

3 Aspect Diagram

The aspect diagram language we propose is inspired by graph transformations where the left part, the pointcut diagram, defines a pattern for which we are looking for matches or morphisms in the base model. The right part, the advice diagram, instructs how to update the base model. Both the poincut and advice diagram are based upon sequence diagrams so that the modeller can think in terms of an already familiar notation which also maps directly to the base model.

Notice that we reuse the syntactic sequence diagrams for defining the pointcut and advice, instead of defining pointcut and advice using the formal model of traces. Still, by ensuring that the pointcut matching process is semantics-based we will avoid the syntactic problem in Figure 1. Although the pointcut is syntactically specified, all the semantic equivalent base model parts will be matched as we will explain in the following sections.

We have defined an aspect (Figure 6) that can affect our base model. The aspect expresses that whenever the message *newAttempt* is followed by *tryAgain*, then add another message *saveAttempt*, in between the two messages matched by the pointcut, to log the failed attempt. A syntactic-based pointcut matching will fail to find matches within our base model, since the two messages come in a different order syntactically. However they will occur in an execution involving two or more iterations. Another challenge for the weaving in this example is that there is no limit to the number of matches as the number of loop iterations in this case is unbounded.

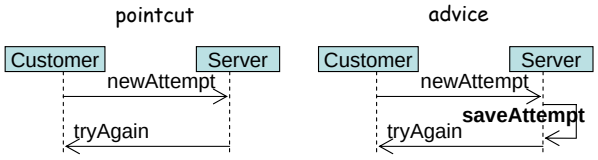

Figure 6: Logging Aspect

In this report we limit the approach to only consider `additive aspects`, that is aspects where deletions and replacements are not allowed. With plain additive aspects all the poincut elements of the pointcut diagram will be repeated in the advice diagram. In addition the advice contains new sequence diagram elements to be woven into the base model in certain places. These places are defined by the ordering relative to the pointcut diagram elements. In the logging example the added message shall occur directly after the *newAttempt* message, and directly before the *tryAgain* message of the base model.

A sequence diagram is `connected` if there is a path between every two pairs of lifelines in the diagram (for each lifeline that has at least one event in the same diagram). A `path` exists between two lifelines if there is a message between these two lifelines (the direction of the message is irrelevant in our path definition), and the path definition is transitive:

Definition 1 *The relation* $path(lifeline, lifeline)$ *and the boolean property* $connected(interaction)$*:*

$$\forall m : Message = (s, tr, re) \Rightarrow path(tr, re) \wedge path(re, tr)$$
$$path(l1, l2) \wedge path(l2, l3) \Rightarrow path(l1, l3) \quad (transitive)$$
$$connected(sd) \Leftrightarrow (\forall l1, l2 \in lifeLines(sd) \Rightarrow path(l1, l2))$$

We require that the pointcut diagram is connected to avoid the problem of intractable weaving as pointed out by Klein et al. [6]. In a disconnected diagram

where the pointcut uses independent lifelines, we may have an arbitrary number of potential matches relative to the number of loop iterations. In such cases we cannot express the weaving on a finite structure. The restriction is not crucial since a single sequence diagram normally is connected.

Figure 7 shows two example pointcut diagrams where the leftmost is connected and the rightmost is not connected. If any of the messages from the leftmost diagram is removed, then it will not be connected. The rightmost diagram is not connected since there is no path between any of the lifelines $L1$ and $L2$ to any of the lifelines $L3$ and $L4$. By adding a message that has $L1$ or $L2$ as transmitter lifeline and $L3$ or $L4$ as reception lifeline (or the opposite direction), the diagram becomes connected.

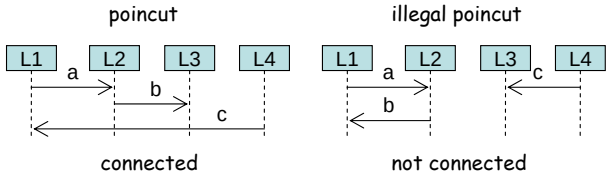

Figure 7: The pointcut shall be connected

The poincut diagram and the body of unbounded loops are restricted so that they can use only events and the *seq* operator. Whenever we would have liked to use an *alt* operator in the pointcut it can be replaced by a set of aspects, one for each operand of the *alt* operator (tool support could perform such replacements and allow the *alt* operator in the pointcut). The elements in the pointcut that are repeated in the advice diagram cannot be placed inside the advice diagrams combined fragments such as *alt* and *loop*. These restrictions will make the weaving process much simpler, while many typical aspects still can be expressed.

The full aspect diagram language for sequence diagrams however, will contain some more advanced constructs (e.g. wild card identifiers for lifelines and messages, replace/delete of messages). Some more investigation is needed before these parts are presented. The precise semantics of additive aspects is presented in the following sections where we cover the weaving process.

We have introduced some restrictions on the sequence diagrams used in pointcut, advice and base models. Our full approach explained in this report is only valid when these restrictions are not violated. Figure 8 shows a summary table of all the restrictions. Although we have only introduced the *seq*, *alt* and *loop* operators, we explain in section 12 that our results can be generalized to also apply for a number of other operators, except for the *strict* operator. The *strict* operator is included in the figure to stress this fact.

Pointcut	Advice	Base
Only *seq* + events	Only plain additive aspects	Only *seq* + events in unbounded loop body
Connected	No pointcut preserved elements in combined fragment operands	
	Unchanged partial order of pointcut preserved elements	*strict* operator not allowed
	strict operator not allowed	

Figure 8: Restrictions for poinctut, advice and base models

4 Semantics-based Matching

Both the pointut and the base model are translated to a set of traces by the $[\![\,]\!]$-operator. Then we look for matching pointcut traces within the base model traces. This way the pointcut matching becomes semantics-based. Whenever we find one entire pointcut trace contained within one of the base model traces, then we have identified a match. Any pointcut trace is by definition of even length (and at least two events in order to be meaningful) since the transmission and reception events of each message must be part of the trace. Formally,

Definition 2 *A match m must satisfy the following (all m_i and e_i are events, id() returns the id of an event, marked() returns true if the event is marked due to a previously handled match):*

$$
\begin{aligned}
& m = \langle m_1, ..m_r \rangle, \quad r \geq 2 \ \wedge \ r = 2 * n, \ n \in \mathbb{N} \\
& \exists t_p \in Trace : t_p \in [\![pointcut]\!] \ \wedge \ t_p = m \\
& \exists t_b \in Trace : t_b \in [\![base]\!] \ \wedge \ t_b = \langle e_1, \ldots, e_s \rangle \\
& \qquad \wedge \langle mapId(e_i), \ldots, mapId(e_j) \rangle = m \ \wedge \ 1 \leq i < j \leq s \\
& \qquad \wedge \forall r \in [i..j] : marked(e_r) = false \\
& mapId(k(s, trans, rec, bId)) = k(s, trans, rec, mapId(bId)) \\
& \forall bId \in \{id(e_i), \ldots id(e_j)\} : mapId(bId) = pId \\
& \qquad \wedge \exists e_p : Event \in t_p : pId = id(e_p)
\end{aligned}
$$

$mapId(Event) : Event$ is an injective function which takes a base trace event as input and returns a pointcut trace event. The kind (k), signal (s), transmitter($trans$), receiver (rec) are not changed by the mapping function, only the message identifier.

We need to iterate through all the pointcut traces and search for matches within the base traces. When a match is found we produce a morphism table where the id of a pointcut element gets a corresponding entry for the id of the base model element. This is important since the same message (signal, transmitter and receiver) may occur more than once within an execution trace. This is not necessary in the traditional sequence diagram setting where a partial trace has no meaning. However, in the setting of pointcut matching we accept partial traces as a match, which leads to inconsistencies if we did not use the

identifiers and the injective *mapId* function. Consider the incorrect match of Figure 9, which could occur if we did not require message identifiers. The left part shows the pointcut and one of its associated traces, and the right part shows the base model and one its associated traces with a mistaken match.

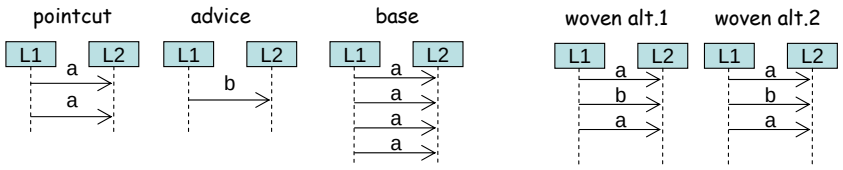

Figure 9: Incorrect match when not using identifiers

After a match is found a weaving according to the advice is carried out, and we continue the search for new matches. All the individual base match events are marked so that we avoid further matches at the same point. Otherwise, our matching and weaving approach would never terminate. We need to continue until there are no further unmarked matches in the base model traces.

Let us calculate the pointcut traces for the logging aspect,

$$[\![pointcut]\!] = \{\langle !newAttempt, ?newAttempt, !tryAgain, ?tryAgain\rangle\}$$

Our example pointcut gives only a single trace, while there in general will be a set of pointcut traces that we need to iterate over in order to find matches and perform a weaving. The single pointcut trace has no matches in the first two base model traces (corresponding to zero and one iterations in the loop). All the rest of the (infinite) set of traces have matches in the base model. In the next section we investigate how to update the base model with the advice for each match.

Figure 10 shows the matching process on an example where we have crossing messages in the base models. The two base models contain the same two messages, a and b and in the same order as within the pointcut. In addition the base models have a crossing c message that crosses differently. The first base model will never result in a match within any of the pointcut traces since the $?c$ event always will split the $?a$ and $?b$ events of the pointcut, and thus prevents a match. In the second base model however, the crossing results in matches since both the $!c$ and $?c$ events will before and after all the events associated with the pointcut messages.

The *loop* operator and especially unbounded loops (as in our login example) will complicate the weaving process since their presence in the base model results in infinite traces. Therefore we explain the weaving first without the loop operator available, and cover the loop operator in detail in section 8 instead.

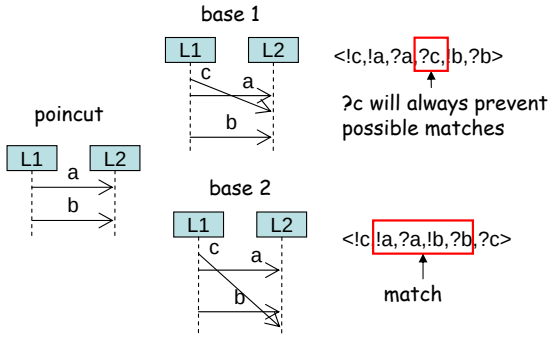

Figure 10: Matching

5 Weave Instructions

We will now describe the semantics of the aspect diagram and preprocess the aspect diagram into weaving instructions. It is beneficial to represent the advice diagram in a syntactic form in which the existing parts from the pointcut are separated from the additive parts:

Definition 3 *The advice syntax, s, is* ***weaveSorted*** *if it satisfies the following conditions ($P = Pointcut$, all a_i and p_i are interactions):*

$$
\begin{array}{l}
s = seq\,[a_0, p_1, a_1, \ldots, p_n, a_n]\,,\ n \geq 1 \\
[\![\ seq\,[p_1, \ldots, p_n]]\!] = [\![P]\!] \\
\forall i \in [1..n]: \quad p_i \neq skip\ \wedge\ p_i \in P \\
\forall i \in [1..n-1]: \quad a_i \neq skip \\
\forall i \in [0..n]: \quad a_i \neq skip\ \Rightarrow\ events(a_i) \cap events(P) = \emptyset
\end{array}
$$

The second line ensures that only additive aspects are defined. It also ensures that the weak sequence ordering from the pointcut is preserved for the pointcut parts of the advice. Only a_0 and a_n are allowed to be empty ($skip$). The final clause states that none of the events within a_i can be part of the pointcut.

For the logging aspect, one possible weaveSorted advice representation is:

$$
\begin{array}{lll}
seq[& skip, & \mathrm{a}_0 \\
 & seq\,[!newAttempt, ?newAttempt]\,, & \mathrm{p}_1 \\
 & seq\,[!saveAttempt, ?saveAttempt]\,, & \mathrm{a}_1 \\
 & seq\,[!tryAgain, ?tryAgain] & \mathrm{p}_2 \\
 & skip \quad] & \mathrm{a}_2
\end{array}
$$

We claim that any valid advice diagram as defined in section 3 can be expressed in a weaveSorted form. The advice diagram can be represented in many ways using the textual syntax defined in section 2. The alternatives are defined by the equivalence relation over $[\![\ ]\!]$. Due to definition 3 in Figure 5 we may

always add a *seq* operator as a wrapping around an interaction. This allows for grouping different parts of the diagram. Remember also that the pointcut can only use the *seq* operator and no combined fragment operators (e.g. *alt*,*loop*), and the pointcut elements repeated in the advice cannot be placed inside combined fragments. From these restrictions and definition 3 in Figure 5, it follows that it is possible to provide a semantically correct weaveSorted advice.

The top part of Figure 11 shows an aspect definition to insert the messages $ad1$ and $ad2$ in between the existing messages $m1$ and $m2$. Since none of these two additional messages are crossing the $m1$ nor the $m2$ messages, but may be ordered completely inside, then they may be grouped as one advice part $a_1 = seq\,[!ad1, ?ad1, !ad2, ?ad2]$. It will always be the case for *alt* or *loop* operators in the advice, that they can get a single insert instruction, since they will not cross with messages or interaction parts defined outside of the operator. In this case there are four ordering requirements for the two lifelines. With respect to lifeline $L1$, a_1 must be placed after $!m1$ and before $!m2$, while for lifeline $L2$, a_1 must be placed after $?m1$ and before $?m2$.

Definition 4 *The function $weaveInstrSet()$ produces a set of weave instructions based on a weaveSorted advice as input:*

$$
\begin{array}{l}
weaveInstrSet(seq\,[a_0, p_1, a_1, \ldots, p_n, a_n]) = \{ \\
a_0 \neq skip \Rightarrow insert(a_0), \\
insert(a_1), \ldots, insert(a_{n-1}), \\
a_n \neq skip \Rightarrow insert(a_n)\},\ where
\end{array}
$$

1. $a_0 \neq skip \Rightarrow (\forall l \in lifelines(a_0) :$
   $before(firstEvt(seq\,[p_1, \ldots, p_n], l)) \in insert(a_0))$
2. $\forall i \in [1..n-1] : \forall l \in lifelines(a_i) :$
   $after(lastEvt(seq\,[p_1, \ldots, p_i], l)) \in insert(a_i) \wedge$
   $before(firstEvt(seq\,[p_{i+1}, \ldots, p_n], l)) \in insert(a_i)$
3. $a_n \neq skip \Rightarrow (\forall l \in lifelines(a_n) :$
   $after(lastEvt(seq\,[p_1, \ldots, p_n], l)) \in insert(a_n))$

The weave instruction set, $weaveInstrSet()$, will contain one *insert* element for each of the advice parts a_i (except if *skip* elements are used for a_0 and/or a_n). Each *insert* element will contain another set of elements which are either $before$ or $after$ elements that defines *where* in the matching base trace to insert the respective advice part.

$firstEvt/lastEvt(interaction, lifeline)$ retrieves the first/last event on the $lifeline$ with respect to the given *interaction*. These two functions are trivial when the *interaction* only uses the *seq* operator and events, as we have within the pointcut part. If there is no event in the *interaction* for a specific $lifeline$, then $firstEvt$ and $lastEvt$ return the empty sequence $\langle\rangle$. $before(\langle\rangle)$ and $after(\langle\rangle)$ in $insert(a_i)$ can be removed since they imply no requirement on the placement of the a_i.

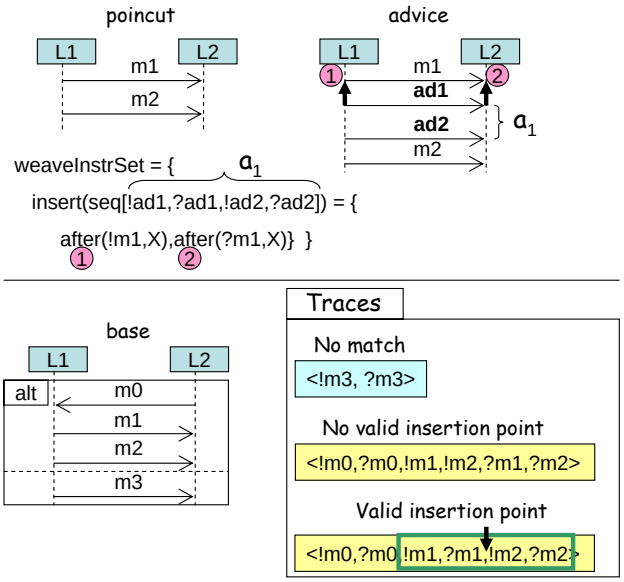

Figure 11: Weave instructions

For the logging aspect we get a set with a single weave instruction. The weave instruction contains four conditions on where to weave the advice part:

$$\{insert(seq\,[!saveAttempt, ?saveAttempt]) = \\ \{after(!newAttempt), after(?newAttempt), \\ before(!tryAgain), before(?tryAgain)\}\}$$

Notice that we are now combining syntax elements with semantic traces. An insert instruction has a syntax element that is to be inserted into a position defined relative to a set of trace events. Only base model traces with matches are candidates for such insertions, and all the explicit placement requirements ($before$ and $after$) have to be fulfilled. In addition we require that the ordering between the a_i elements is maintained as they are inserted into the trace, i.e. any a_i comes before a_{i+1} (not necessarily immediately before).

We will see all the details on how the base model traces are affected by the weave instructions in section 7. The syntactic advice parts (a_i) are inserted at a valid insertion point (not necessarily unique location) in a matching base model trace. This insertion results in an intermediate *hybrid trace*, which is a sequence of trace events and syntax elements:

$$\langle h_1, \ldots, h_n \rangle,\ \forall i \in [1..n] : h_i \in Event \vee h_i \in Interaction$$

The next section describes how the hybrid trace can be transformed back into regular traces so that we can continue working on the semantic level in the further weaving process.

6 Woven Sequence Diagrams

The goal of the weaving process is to produce new models where the advice instructions are woven at the correct places. We assume now that we have found a matching base model trace and inserted the syntactic advice parts, resulting in a hybrid trace. Before the matching process continues we need to calculate the full effect from our insertions upon the set of base model traces.

To achieve this, we propose to reverse-engineer the hybrid trace back to a sequence diagram that reflects the semantics of the woven hybrid trace. The woven diagrams can then be used to calculate the woven semantics by using the $[\![\,]\!]$ function from the STAIRS formal model.

The reverse-engineered sequence diagrams we produce are not intended for human comprehension. Thus, we do not try to make these diagrams close to the original ones, and we do not intend to make them nicely structured with optimal usage of combined fragments. That would also be a very complicated task.

Informally a sequence diagram is produced by making one outermost *alt* operator with one operand for each trace. Each trace is wrapped inside a *seq* operator that keeps the ordering within the trace. The trace items can be used directly inside the *seq* operator since they are either events or woven syntax elements from the advice.

Definition 5 *The function, makeSD(), makes a sequence diagram from a hybrid trace set $T = \{t_1, \ldots, t_n\}$ or from a single trace $t = \langle x_1, \ldots, x_n \rangle$:*

$$\begin{aligned} makeSD(T) &= alt\,[makeSD(t_1), \ldots, makeSD(t_n)] \\ makeSD(t) &= seq\,[x_1, \ldots, x_n] \end{aligned}$$

The following lemma shows that the $makeSD()$ definition is appropriate, since making a sequence diagram from a trace set and then going back to traces with the $[\![]\!]$ operator, will produce the original trace set:

Lemma 1 *Let T be the trace set for a sequence diagram sd ($T = [\![sd]\!]$), where sd uses only seq, alt and loop and sd is well-formed (sdef. in the proof refers to Figure 5):*

$$[\![makeSD(T)]\!] = T.$$

Proof: By induction on the number of traces. one trace

$$[\![makeSD(\{\langle e_1, \ldots, e_n \rangle) = alt\,[seq\,[\langle e_1, \ldots, e_n]] \quad \text{(def. 5)}$$

New induction on the length n of the trace. two events

$$= [\![alt\,[seq\,[\langle !m, ?m]]]\!] = \{\langle !m, ?m \rangle\}\ \square \quad \text{(sdef.5,4,3, } \succsim\text{)}$$

Inner induction step. Holds for length n. trace length n+2

$$\begin{aligned} &= [\![alt\,[seq\,[\langle e_1, \ldots, e_{n+2}]]]\!] \\ &= [\![alt\,[seq\,[\langle e_1, \ldots, e_n]]]\!] \succsim [\![e_{n+1}]\!] \succsim [\![e_{n+2}]\!] \quad \text{(sdef.5,4,3,5)} \\ &= \{\langle e_1, \ldots, e_n \rangle\} \succsim \{\langle e_{n+1} \rangle\} \succsim \{\langle e_{n+2} \rangle\} \text{ (ind.hyp.,sdef.3)} \end{aligned}$$

Since the original diagram *sd* used only *seq*, *alt* or *loop* which all imply from sdef.1-9 that the $\succsim$ is used by $[\![]\!]$ to produce the ordering among all the trace events. In this step of the induction we have assumed a single trace which will only result from $\succsim$ if there is at most one ordering imposed by the partial ordering of the events. Just adding $!m, ?m$ to the end of the trace must be one possible order, and thus also the only one. Thus the one trace case is proven. Now, we assume it holds for n traces. $\underline{n+1 \text{ traces}}$

$$
\begin{aligned}
&= [\![makeSD(\{t_1, \ldots, t_{n+1}\})]\!] \\
&= [\![alt\,[makeSD(t_1), \ldots, makeSD(t_{n+1})]]\!] \quad \text{(def. 5)} \\
&= \{t_1, \ldots, t_{n+1}\} \quad \text{(sdef.5,ind.proof of } \underline{\text{one trace}}\text{)}\square
\end{aligned}
$$

7 The Weave Algorithm

This section presents the full weaving algorithm (as pseudocode) which takes a base model sequence diagram and an aspect diagram as input and returns a woven sequence diagram. First, we explain the matching and weaving process.

There are a set of traces that can be considered equivalent with respect to the partial order of the events. The weave algorithm takes advantage of this and weaves only one of these partial order equivalent traces, while it ensures that the woven result is propagated to all of the partial order equivalent traces.

Definition 6 *We say that two traces* $t^A = \langle t_1^A, \ldots, t_n^A \rangle$ *and* $t^B = \langle t_1^B, \ldots, t_n^B \rangle$ *are* ***partial order equivalent (POE)*** *if and only if:*

$$[\![seq\,[t_1^A, \ldots, t_n^A]]\!] = [\![seq\,[t_1^B, \ldots, t_n^B]]\!]$$

Let $POE : Trace \rightarrow Traces$ be a function that calculates all the POE traces of a single trace. Then it follows from the definition of POE traces and the definition of $makeSD$ that $POE(t^A) = [\![makeSD(t^A)]\!]$.

For each pointcut trace we check for matches within each base trace. Each time a matching pointcut trace is found within a base trace, we will check the weave instructions set to see if there are valid insertion points for all the weave instructions.

If there is at least one insert instruction without a valid insertion point, then no weaving is performed on the base trace. We investigate the base model and the aspect in Figure 11. From the base model we get three different types of traces for which we will explain the action to be taken in the weaving process:

1. *Trace with match and valid insertion points.* In the example there is only one insert instruction, a_1, which has a valid insertion point. Thus, a_1 will be inserted at the corresponding valid insertion point within the base model trace which we now call $bTraceValid$. An arbitrary valid position can be chosen if there is more than one and as long as any a_i comes before a_{i+1}. We will also remove all the POE traces of $bTraceValid$ except $bTraceValid$ where we insert the advice parts. This suffices for all the POE traces since we next apply $[\![makeSD(weave(bTraceValid))]\!]$

to produce all deleted permutations again except that the woven result now holds additional constraints on the partial ordering of events.

2. *Trace with match and at least one invalid insertion point.* In our example there is no valid insertion point which satisfies both $after(?m1, a_1)$ and $before(!m2, a_1)$. No action shall be taken, and in general, none of the insert instructions will be performed when there is at least one invalid insertion point.

3. *Trace without match.* No action shall be taken.

Although we say that no action is taken for the two latter trace cases, they will be deleted as part of the action within the first trace case if they satisfy the test for deletion of permutated traces.

The *weave*() algorithm (7.1) shows pseudocode for the full approach. We first treat all the loops within the base model by a call to the *treatLoops*() algorithm (8.3) (treatment of loops is explained in the next section). Then we calculate the semantics, that is the trace sets of the pointcut and the base model. When calculating the base traces, we will ignore the unbounded (and marked) loops (loops that are treated will be marked and can then be ignored as we explain in the next section). The trace sets may be a hybrid structure consisting of some unbounded and marked loops which we know will not be of interest in the further weaving process. We also calculate the weave instructions by first preparing the advice to be in the weaveSorted representation. Then we exclude pointcut traces that does not have valid insertion points for all the insert instructions. Such pointcut traces will only match base traces of type two and three for which no action will be taken.

Notice that the advice may contain new, unbounded loops. They should not be matchRepetitive (to be defined in section 8.2) since this would be an aspect definition which will never terminate. If it has overlapping events with the pointcut, then it should be rewritten according to the *rewriteNonRep* algorithm (to be defined in section 8.2) before the weaving takes place (*treatLoops*(*aspect.getAdvice*) ensures such action is taken for all the loops).

After the initialization part, we iterate through all the pointcut traces and immediately enter a while loop which iterates through all the base traces that have matches for the current pointcut trace. The *getMatch* method will always choose the leftmost match within a base model trace. The base trace which contained the match (*bTrace*) is used to find all POE traces and delete these from the set of base traces.

basePart is a subsequence of the base model trace which matches the pointcut trace. The morphism table of the match holds the correspondance between the pointcut elements and the base model matched elements. The advice holds the weaving instructions which is relative to the pointcut identified elements. We will use the morphism table to replace those IDs with the base model element IDs, and the weave instructions will be updated with the base model IDs and stored in the *instrID* variable. Then we proceed with the weaving instructions

one at a time by inserting the advice parts into valid insertion points. Finally, we mark all the elements in the base trace which were part of the current match.

Algorithm 7.1: WEAVE($base, aspect$)

$pointcut = aspect.getPointcut$
$base = treatLoops(base, pointcut)$
$bTraces = [\![base]\!]$
$advice = treatLoops(aspect.getAdvice)$
$advice = aspect.getWeaveSortedAdvice$
$weaveInstrSet = advice.weaveInstrSet$
$pTraces = [\![pointcut]\!]$
for each $pTrace \in pTraces$
  do if ! $pTrace.hasValidIns(weaveInstrSet)$
  then $pTraces = pTraces - pTrace$
for each $pTrace \in pTraces$
  do while $bTraces.existMatch(pTrace)$
  do
    $match = bTraces.getMatch(pTrace)$
    $bTrace = match.getBTrace$
    $bTraces.removeAllPOETraces(bTrace)$
    $basePart = match.getbasePartTrace$
    $tbl = match.morphismTable$
    $instrIDSet = weaveInstrSet.replaceIDs(tbl)$
    for each $instr \in instrIDSet$
      do $basePart.weave(instr)$
    $bTrace.markAllElems(basePart)$
    $bTraces.replace(bTrace, [\![makeSD(bTrace)]\!])$
$wovenBase = makeSD(bTraces)$
return ($wovenBase$)

The *baseTraces* holds the value of the semantically woven traces. After each time an advice insertion has taken place, we have added syntactic elements into trace sequence of events. Before we continue the matching process, we need to we replace the updated base trace with the updated semantics for this trace($[\![makeSD(bTrace)]\!]$).

When there are no further matches the weave algorithm finally calls the *makeSD*() algorithm to produce the woven sequence diagrams. Notice that the whole weaving process has been described for a single aspect. If there are multiple aspects, then the idea would be to repeat the call to algorithm 7.1 for each aspect. Since one aspect may introduce new matches for other aspects, we need to recheck for matches in the other aspects each time an aspect has been applied. Multiple aspects raises a lot of new issues which are outside the scope of this report.

An overview of the weave algorithm is shown in Figure 12. The figure shows that the algorithm looks for matches based on the pointcut and base traces.

When a match is found, the advice parts are inserted to produce a hybrid trace. The hybrid trace is reverse-engineered into sequence diagram for which we compute the traces of the woven match base trace. The traces of the woven match base trace replaces the old unwoven match base trace and all its partial order equivalent traces. The process continues until there are no further matches. At the end, the base traces will contain the woven traces.

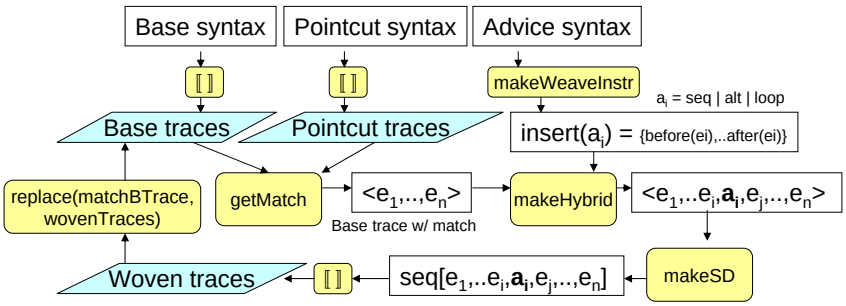

Figure 12: Overview of the weave algorithm

8 Weaving Unbounded Loops

This section will describe how we can do the weaving also for the *loop* operator. Loops without an upper bound are troublesome because they produce an infinite trace set, and loops with large upper bounds may give the weaving process a performance problem. We will however show that all unbounded loops may be restructured and fully woven on a finite structure for a specific aspect. The algorithm we present exploits that the number of relevant loop iterations are restricted with respect to the pointcut, when the pointcut is connected (definition 1).

We say that a loop is `matchRepetitive` if we can infinitely increase the number of matches as the number of loop iterations increase. Now we present a lemma that helps us to determine if a loop is matchRepetitive by investigating a finite trace structure. Remember that a loop body contains only the *seq* operator and events. The lemma proves it is sufficient to consider the loop with an upper bound equal to the number of messages in the pointcut.

Lemma 2 *An unbounded loop $lp = loop\,[body]$ is **matchRepetitive** for a pointcut pd if and only if there exist at least one match in the bounded loop $lp' = loop\{numMess(pd)\}\,[body]$ ($numMess()$ returns the number of messages within a sequence diagram).*

Proof:

- *if-direction.* If there exists a match within lp', then we may repeat these iterations within lp infinitely many times.

- *only if-direction.* Let n be the fewest number of iterations for lp which gives a match. Such a matching trace must involve at least one event from all iterations, otherwise we could exclude iterations not contributing to the match and get a match within fewer iterations than n. The pointcut trace which equals the match involves a number of messages which all have two events. Since the match need to involve both the transmission and reception events of a message, we know that each iteration contributes with at least two events in the matching trace. The length of the pointcut trace is twice the number of messages, which means that n cannot be larger than the number of messages within the pointcut. □

We cannot calculate the entire trace of unbounded loops since the result will be infinitely long traces. Instead we do a preprocessing of all unbounded loops and then produce hybrid traces consisting of normal trace events intermixed with unbounded, but preprocessed loops. In the preprocessed loops the necessary weaving is already handled, and will thus be ignored in the rest of the weaving process.

We only consider unbounded loops with cardinality $0..*$, where $*$ is unbounded. Other unbounded loops may easily be translated into finite loops combined with $0..*$ loops by rewrites from $loop\{n..*\}\,[body]$ to $seq[loop\{n\}\ [\ body\ ],\ loop\ [\ body\ ]\ ]$, and using the *alt* operator to split loops of the form $loop\{2, 10, 20..*\}$. First we cover the simplest case where the loop is unrelated to the pointcut, the we cover non-matchRepetitive loops, and finally we cover loops which are matchRepetitive.

When we produce traces to be matched we produce hybrid traces with ordinary traces for all bounded loops, while unbounded, marked loops are simply dumped to the trace so that the trace consists of either events or $loop'$ items. Ordinary marked events are handled by the $[\![\ ]\!]$ operator as before, while we have to adjust this operator to work also for $loop'$ items. By definition $[\![loop'\,[\ldots]]\!] = \{\langle loop'\,[\ldots]\rangle\}$. We also extend the definition of $\succsim$ in Figure 5 so that events are replaced by the new term *hybridEvent*. A *hybridEvent* is either ordinary events or $loop'$. Furthermore, we say that $loop'$ is a hybridEvent on a lifeline l if $\exists e \in events(loop') : lifelineAction(e) = l$, where a $lifelineAction$ returns the lifeline on which an event takes place. The $\succsim$ operator will now ensure that $loop'$ is placed only in positions which fulfill the partial order of the lifelines it involves.

The unbounded, marked loops are ignored in the remaining matching process and finally copied back to the plain syntax tree when the reverse-engineered sequence diagram is produced. For a hybrid trace $\langle e_1, \ldots, e_i, loop', e_{i+1}, \ldots, e_n\rangle$ $(\forall j \in [1..n] : e_j \in Event)$, the match must either be entirely within the before-part $(e_1, \ldots, e_i)$, or entirely within the after-part $(e_{i+1}, \ldots, e_n)$.

8.1 Unrelated Loops

The simplest case of unbounded loops are those that are *unrelated* to the pointcut. *Unrelated* means that $events(loopBody) \cap events(Pointcut) = \emptyset$.

Definition 7 *An unrelated loop, loop [body], can be rewritten by a rewrite rule (**rewriteUnrel**):*

$$alt\,[skip, loop'\{1..*\}\,[body]]$$

The *skip* operand corresponds to zero iterations and the other operand corresponds to 1^+ iterations. The remaining unbounded loop is marked since it will not be involved in any matches, and we will not produce any traces for this loop in the matching process. Since *skip* does not have a graphical notation, we will use the *opt* operand to display *alt* operators with *skip* operands in diagrams. The rewrite result above is equivalent to $opt\,[loop'\{1..*\}\,[body]]$.

8.2 Non-matchRepetitive Loops

In an unbounded, non-matchRepetitive loop, we know that possible matches will include preceding or succeeding traces (of the loop) or both. This is because matches only containing loop traces leads to match repetition. Potential matches in the preceding traces may be completed by loop traces, and potential matches in the loop traces may be completed by succeeding traces. By looking at the proof of lemma 2 we deduce that the maximum number of loop iterations involved in the match, is $numMess(pd) - 1$.

We translate the syntactic representation of the loop into a semantically equivalent form where we separate the unbounded loop part which cannot be part of any matches and the part that may contain matches as loops with upper bounds. We introduce an *alt* operator with two operands, one with only an upper bound on the old loop. The second *alt* operand contains the unbounded loop with a prefixed and postfixed loop of an upperbound depending on the length of the pointcut.

An unbounded, non-matchRepetitive loop, $loop\,[body]$, has the following rewrite rule (**rewriteNonRep**):

$$\begin{array}{lll} alt[& loop & \{0..((numMess(pd)-1)*2-1)\}\,[body]\,, \\ & seq[& loop\{numMess(pd)-1\}\,[body]\,, \\ & & loop'\,[body]\,, \\ & & loop\{numMess(pd)-1\}\,[body]\,]] \end{array}$$

The unbounded loop is always preceeded by $numMess(pd)$ - 1 loop iterations, which prevents any completion matches in the unbounded loop. Similarly the unbounded loop is always followed by $numMess(pd) - 1$ loop iterations, which prevents that any potential matches start in the unbounded loop. These arguments hold since the loop body can only contain the *seq* operator and since the pointcut is connected. This implies that all the events that could be part of matches in the unbounded loop always will be treated in the preceding or succeeding loop iterations that surrounds it and that have an upper bound. Notice that the unbounded loop is marked with a prime to indicate that no further matching or weaving shall be performed on that loop. The same rewriting process is shown by using diagrams in Figure 13.

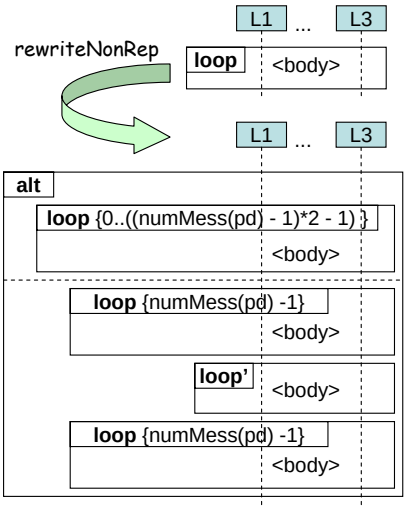

Figure 13: Rewriting a non-MatchRepetitive Loop

8.3 MatchRepetitive Loops

We will now check if our base model loop example is matchRepetitive($tA = tryAgain$ and $nA = NewAttempt$). As a consequence of lemma 2 it is sufficient to check the traces of the loop with exactly $numMess(pd)$ iterations:

$$numMess(pd) = 2$$
$$[\![pointcut]\!] = \{\langle !nA, ?nA, !tA, ?tA\rangle\}$$
$$[\![loop\{2\}\ []]\!] = \{\langle !tA, ?tA, \underbrace{!nA, ?nA, !tA, ?tA}_{match}, !nA, ?nA\rangle\}$$

This bounded loop has a single trace with a match, meaning that our base model loop is matchRepetitive. For such loops we will, as with the non-matchRepetitive loops, treat them in isolation from the rest of the base model. The treatment is again by translating the unbounded loop into bounded loops for which all traces are generated and a preprocessed unbounded loop that is not part of the further trace weaving process. As opposed to the non-matchRepetitive loops, the preprocessed unbounded loop will have matches for which we have added the aspect advice.

Algorithm 8.1 shows pseudocode to transform a matchRepetitive loop into a new syntax expression containing a bounded loop and an unbounded loop which is not matchRepetitive. This new structure is semantically equivalent except that a partial weaving is performed. The remaining weaving will continue working on the reduced structure in which there are no matchRepetitive loops.

Algorithm 8.1: REWRITEREP($loop\,[body]\,, aspect$)

$pd = aspect.getPointcut$
$r = numMess(pd)$
$boundedLoop = loop\{0..r-1\}\,[body]$
$loopTraces = [\![loop\{r\}\,[body]]\!]$
$pTraces = [\![pointcut]\!]$
$match = get1stMatch(loopTraces, pTraces)$
$matchTrace = match.trace$
$before = matchTrace.beforeMatch(match)$
$matchPart = matchTrace.matchPart(match)$
$after = matchTrace.afterMatch(match)$

comment: Weaving 3 trace parts

$wovenBefore = weave(seq\,[before]\,, aspect$
$wovenMatch = weave(seq\,[matchPart]\,, aspect$
$wovenAfter = weave(seq\,[after]\,, aspect$

$afterBefore = seq\,[wovenAfter, wovenBefore]$
if $!matchExist(afterBefore, aspect)$

then
- **comment:** No permutation
- $body = seq[wovenBefore, wovenMatch, wovenAfter]$
- $wovenBody = weave(body, aspect)$
- $wovenLoop = loop\,[wovenBody]$
- $wovenLoop.treatLoop$
- **return** $(seq\,[wovenLoop, boundedLoop])$

else
- **comment:** Permutation
- $body = seq\,[wovenMatch, afterBefore]$
- $wovenBody = weave(body, aspect)$
- $permLoop = loop\,[wovenBody]$
- $permLoop.treatLoop$
- $newAfter = seq\,[wovenMatch, wovenAfter]$
- $retVal = seq[\quad opt[wovenBefore, permLoop, newAfter],$ $boundedLoop]$
- **return** $(retVal)$

The algorithm prepares the matching process by calculating the traces for the pointcut and the loop with $numMess(pd)$ iterations. Then we retrieve the first match ($get1stMatch$). The loop trace where the first match is found consists of three parts: an event list in the match part ($matchPart$), a list of events preceding the match events ($before$), and a list of events succeeding the match events ($after$). The three parts are woven individually to obtain $wovenBefore$, $wovenMatch$, and $wovenAfter$.

Since we have only additive aspects, the $wovenMatch$ will contain all the

match events as marked, and these marked events will separate possible further matches to come only prior to or after this marked match. The only way for further matches to involve more than one iteration is that $afterBefore = seq\,[wovenAfter, wovenBefore]$ contains matches.

If there are no matches in $afterBefore$, then no permutation of the loop is needed. We rejoin the three previously split parts using their original order, *seq* [$wovenBefore$, $wovenMatch$, $wovenAfter$], and the result becomes the new loop body (*body*). We weave this body and we call on the *treatLoop* to handle this loop by either the *rewriteUnrel* or *rewriteNonRep* rewrite rules.

It is important to note that the loop trace we have used to construct the new loop body comes from $numMess(pd)$ iterations. Thus, the rewrite will only support $n{*}numMess(pd)$ (2,4,6 etc. in our example) iterations from the original loop. To cope with this, we add the *boundedLoop* with zero to $numMess(pd)-1$ (0..1 in our example) iterations at the end. In our example we will then support also 1,3,5 etc. iterations of the original loop.

If there are no matches in $afterBefore$, then permutation of the loop is needed. We shift the loop so that we get the sequence *seq* [$wovenMatch$, $afterBefore$]. To make the full syntax model identical to the original loop, we will produce an outer *seq* operator to ensure the $wovenBefore$ is inserted before we enter the loop and the $seq\,[wovenMatch, wovenAfter]$ is inserted after we exit the loop.

As in the non-permutation case, we need to call on the *treatLoop* to handle the new loop by either the *rewriteUnrel* or *rewriteNonRep* rewrite rules. There are still parts of the final result that may involve matches. But these parts can be treated by the normal weaving apparatus for finite traces. From the pseudocode we see that the *rewriteRep* algorithm to treat a matchRepetitive loop, $loop\,[body]$, will always terminate, if the weaving of the finite traces $loop\{numMess(pd)\}\,[body]$, terminates. The latter termination criteria depends on the aspect. For plain additive aspects as we only consider in this report, it is sufficient that the events in the additive parts are disjoint from the events of the pointcut.

The algorithm above will translate our original example loop into the diagram shown in Figure 14 (sA = *saveAttemtp*, marked elements are displayed with a prime). The loop will be permuted since we get additional matches when combining the end of the the woven body ($!nA, ?nA$) with the start of the woven body ($!tA, ?tA$). There will be two nested *opt* operators where the innermost results from applying the *rewriteUnrel* rule to the permuted and woven loop body, and the outermost comes directly from the *rewriteRep* algorithm.

8.4 Loop Summary

We now define the general loop algorithms which use the algorithms defined in the previous two subsections. The *treatLoop* algorithm (8.2) takes a loop and an aspect as input and returns an *Interaction* as output. The goal of this algorithm is to produce a new *Interaction* structure (if needed) to replace the loop, where all weaving in unbounded loops are finished.

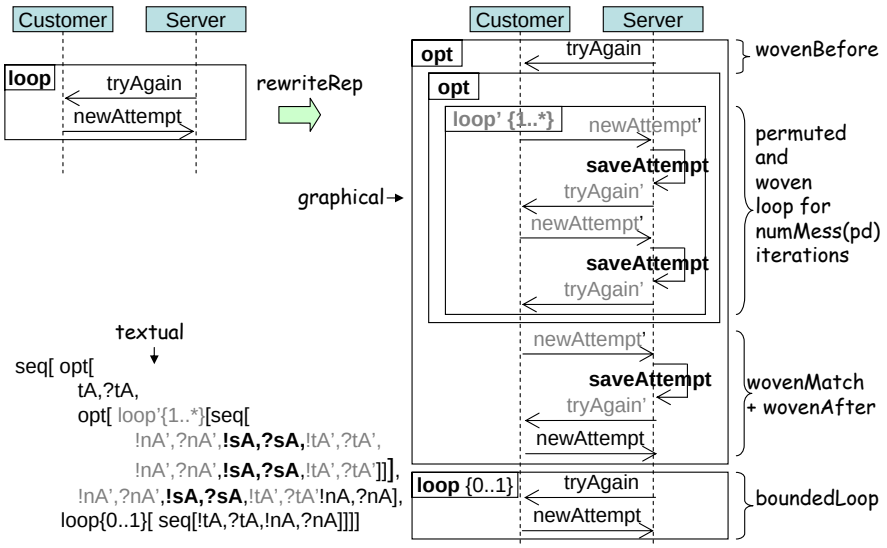

Figure 14: Rewriting the matchRepetitive loop of the login example

For loops with an upper bound, we generate full traces as with the other syntactic operators *seq* and *alt*. The rewrite rule *rewriteUnrel* is used if the loop is unrelated to the pointcut. Otherwise, unbounded loops are rewritten by following either the non-matchRepetitive (*rewriteNonRep*) or matchRepetitive (*rewriteRep*) loop rewrite expression. Notice that the latter rewrite results in an interaction where further rewrites may be necessary. The *treatLoops*() algorithm (8.3) is called to take care of this.

Algorithm 8.2: TREATLOOP($loop, aspect$)

$pd = aspect.pointcut$
if $loop.hasUpperBound$
  then return ($loop$) ⟨make all traces⟩
  else if $events(loop) \cap events(pd) = \emptyset$
  then return ($loop.rewriteUnrel$)
  else
$max = numMess(pd)$
$isNonRep = hasMatch(loop\{max\}, pd)$
if $isNonRep$
  then return ($loop.rewriteNonRep(pd)$)
  else
return ($treatLoops((loop.rewriteRep(pd))))$

The goal of our next algorithm is the same as for the previous algorithm, but

works for general interactions consisting of an arbitrary number of loops. When the latter algorithm is finished, all the unbounded loops have been marked so that they will be ignored in the continued matching process.

Algorithm 8.3: TREATLOOPS($diagram, aspect$)

for each $ld : Loop \in diagram$
  do $diagram.replace(ld, TreatLoop(ld, aspect))$
return $(diagram)$

9 Examples

This section provides some examples to illustrate how the weaving works in practice. We will show the woven result as sequence diagrams for easier comprehension. However, we do not propose that the woven sequence diagrams should be used other than for validation purposes, since the woven diagrams will typically have a poor layout in our approach. Here is a summary of the examples:

- **Consecutive *alt* operators**. This shows that we do solve the Figure 1 example properly with our semantics-based solution. The example has a pointcut that should match with the base model which has parts of the pointcut in one *alt* operand, and parts of the pointcut in another *alt* operand.

- **MatchRepetitive loop**. This example shows how to treat loops that are match repetitive. The example illustrates that the weaving of even a single aspect may produce different woven results and still be correct in a sense with respect to the aspect definition. This non-deterministic behaviour may be seen as a limitation of our current approach.

- **Crossing messages**. We illustrate by this example that our approach handles crossing messages. Such crossing messages may be used within any of the three models pointcut, advice or base model (as long as the pointcut preserved events have the same order in the advice).

- **Impossible automata weaving**. This is a loop example which is an unbounded sequence diagram leading to non-regular trace expressions, which cannot be handled by automata-based approaches. Our approach, on the other hand, handles this case easily by comparing the loop body with the pointcut and discovering that the *rewriteUnrel* rule can be used. The example is adopted from Klein et al. [6] who have presented an automata-based solution that fails in this case.

- **Disconnected pointcut**. This example shows why our approach fails to handle aspects with disconnected pointcuts. This problem is also adopted from Klein et al. [6] who have presented an automata-based solution that also fails in this case.

9.1 Example: Consecutive *alt* operators

We now go back to the Figure 1 example which illustrated the syntactic-based weaving problem. We have introduced an advice and will now show the full weaving process starting with calculating base traces, $weaveInstrSet$ and base traces (Figure 15).

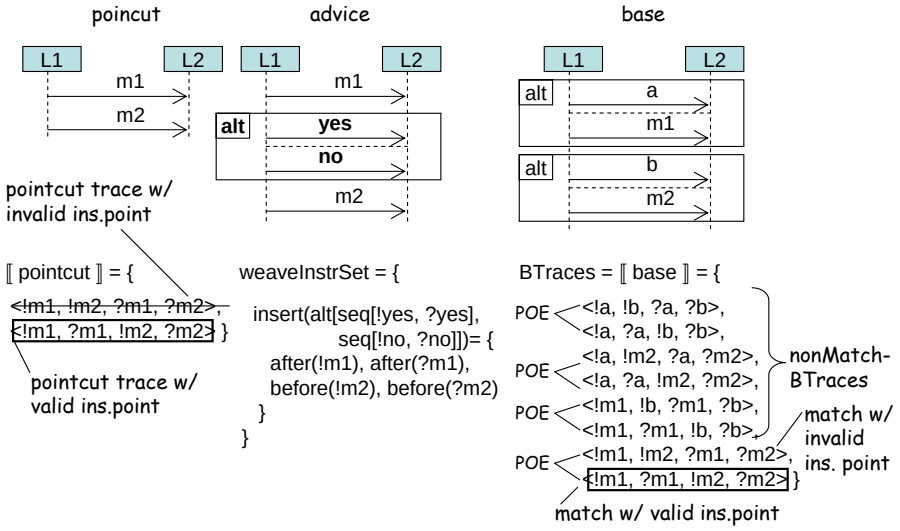

Figure 15: Example: Consecutive alt operators

The $weaveInstrSet$ defines where the advice parts shall be inserted, in this case the single advice part $alt\,[\ldots]$. There are two pointcut traces, where only one has valid insertion points for the weave instruction. The other pointcut trace are thus excluded from the matching process since there is no valid position to insert the advice part in matching traces.

There are eight base model traces of which two partial order equivalent (POE) traces exist for each operand combination. We find only one match of the remaining pointcut trace within the base model traces.

We will now perform weaving on the identified match. $BTraces$ holds all the base traces initially and will hold the final woven trace result at the end. We use the variable $nonMatchBTraces$ to denote the first six traces that are not related to the pointcut traces. First, we delete the other POE-equivalent base traces of the matching base trace, in this case one trace, and the result is:

$$BTraces = nonMatchBTraces \cup \{\langle !m1, ?m1, !m2, ?m2 \rangle\}$$

Then, we insert the advice part in a valid position, which in this case is only one allowed position, and we also mark all the events in the match to prevent further matches for the same events. The result is:

$$\begin{aligned} BTraces \quad = \quad & nonMatchBTraces \cup \\ & \{\langle !m1', ?m1', alt\,[seq\,[!yes, ?yes]\,, seq\,[!no, ?no]]\,, !m2', ?m2' \rangle\} \end{aligned}$$

The resulting hybrid trace is reverse-engineered by the *makeSD* function. For single traces this only means that an outer *seq* is used on the trace event sequence. The hybrid trace is replaced by the semantics of the reverse-engineered hybrid trace:

$$\begin{aligned} BTraces \quad = \quad & nonMatchBTraces \cup \\ & [\![seq\,[!m1', ?m1', alt\,[seq\,[!yes, ?yes]\,, seq\,[!no, ?no]]\,, !m2', ?m2']]\!] \end{aligned}$$

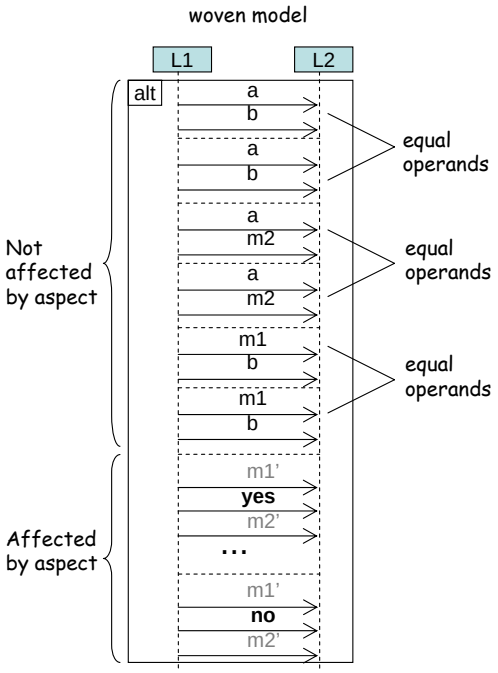

Figure 16: Example: Consecutive alt operators - Woven model

There are no more matches in the base model after one application of the aspect, and the weaving will terminate. We may use *makeSD* to investigate the result as a sequence diagram to verify that the woven result is correct. We get the woven sequence diagram shown in Figure 16. Notice that we get duplicate operands for each set of POE traces. A simple enhancement of the *makeSD* function could remove all but one trace from each set of POE traces. We also see that the original two *alt* operators are not preserved in the result. In this example the weaving makes it impossible to preserve the combined fragment structure. Even for cases where it is possible to preserve the original structure of operators (such as parts not affected by the aspect), our weave approach will not preserve such a structure. Such preservation is not a target for our approach since we only attempt to weave a semantically correct result as represented by the woven traces.

9.2 Example: MatchRepetitive loop

Figure 17 shows an example with a matchRepetitive loop. The aspect defines that a b message shall be added after two consecutive a messages. The base model contains a single a message followed by an unbounded loop with three consecutive a messages in the body.

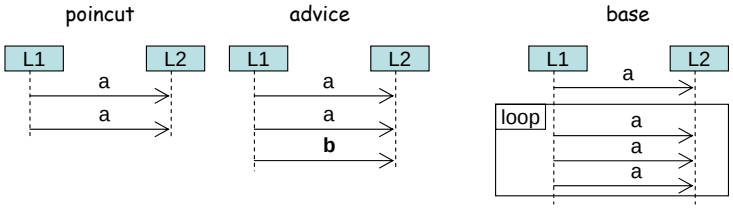

Figure 17: Example: MatchRepetitive Loop

We detect that the loop is matchRepetitive by finding a match within $loop\{numMess(pd) = 2\}$. Assume also that the match we find is the very first two a messages in the trace of $loop\{2\}$. The $before$ part is then empty and it is trivial that no permutation is needed according to the *rewriteRep* algorithm. The woven parts of the match part (two first a's) and the after part (last four a's) are joined together to produce the *wovenLoop* shown in the left part of Figure 18.

This loop is sent to the *TreatLoop* algorithm which detects that this is now an unrelated loop since all unmarked events (only b events since all a events are marked) are disjoint with the pointcut events. The result of *rewriteUnrel* is an alt with a skip operand. In order to display this graphically (skip has no defined graphical layout), we use the equivalent *opt* operator which means that we get one empty trace and one trace with the operand of the *opt*. The final part of the *rewriteRep* algorithm is to place the *boundedLoop* after the *rewriteUnrel* result, and the combined result is shown in the right part of Figure 18.

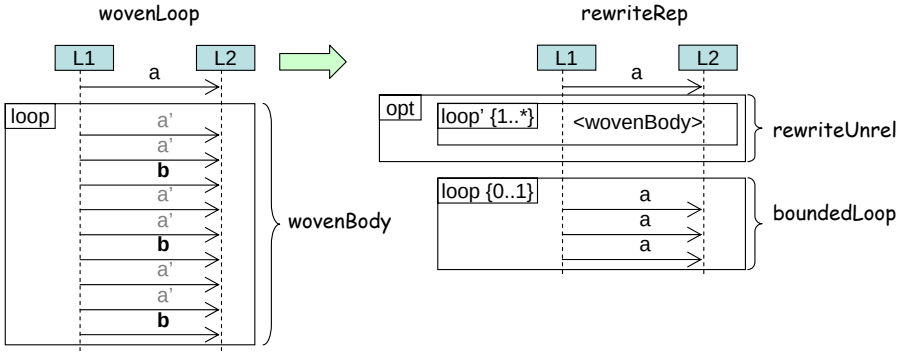

Figure 18: Example: MatchRepetitive Loop - Intermediate steps

All the unbounded loops are treated since there are no more unmarked and unbounded loops present in the partially woven model. We now continue by a call to the *weave* algorithm to handle the partially woven model. When we generate new traces for the partially woven model, we get four traces. These are the result of all combinations of empty *opt*/*opt* operand with zero or one iterations of the *boundedLoop*. After these four traces are woven and reverse-engineered back to a sequence diagram, we get the result shown in Figure 19.

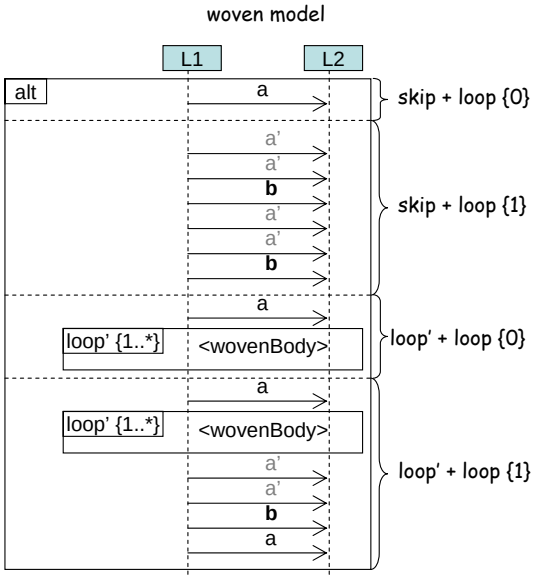

Figure 19: Example: MatchRepetitive Loop - Woven model

If we simply the traces a bit by only considering the message order and not events, $a(aaa)^*$ describes the possible traces before the weaving. After the weaving we get the following result:

$$\{a, a'a'ba'a'b, a(a'a'ba'a'ba'a'b)^+(a'a'ba)^?\}$$

This seems to be the correct result in the sense that there will never exist any sequence of two unmarked s's without an added b. In addition we see that by removing all the b's and the marking of the a's, then we get the same set of possible traces that we had before the weaving.

Notice that we get $a'a'ba'a'b$, while for all other woven traces (with the aspect applied) there will be three consecutive a's before the first b occurs. Currently, this is a limitation of our approach that we cannot ensure that the first two a's will be treated, or the last two a's are treated. Two different execution runs may also produce different results and still follow the approach explained in this report. It is possible to strengthen the match condition so that we always choose the first/last match of all the POE traces, when we perform the weaving. This should ensure that the result is always the same for every execution run or in any tool implementing our approach. However, more effort is needed to find a strategy that works in combination with the unbounded loops. This is because unbounded loops can then no longer be treated in isolation from the rest of the trace.

9.3 Example: Crossing messages

The examples in Figure 20 show how crossing messages influence the weaving. We have a simple aspect of two consecutive messages a and b with the advice to insert a message adv that crosses the matched a message.

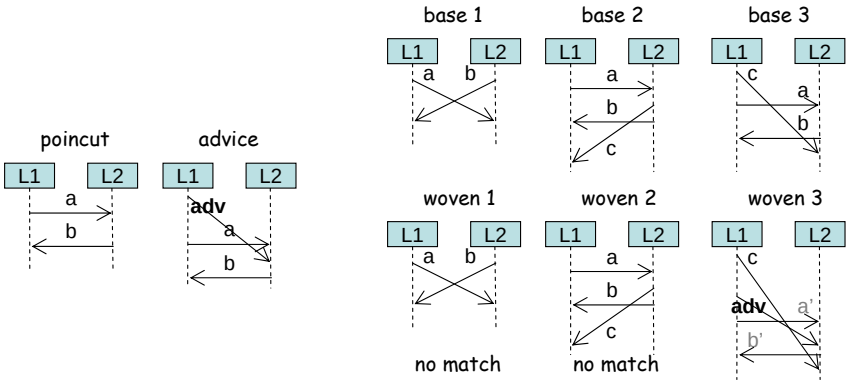

Figure 20: Crossing messages

First we calculate the pointcut traces and the weave instructions, which are independent of the base models. There is a single pointcut trace: $\langle !a, ?a, !b, ?b\rangle$.

One possible representation of the *weaveSorted* advice is:

$$\begin{array}{lll} seq[& !adv, & \mathrm{a}_0 \\ & seq\,[!a, ?a]\,, & \mathrm{p}_1 \\ & ?adv, & \mathrm{a}_1 \\ & seq\,[!b, ?b]\,, & \mathrm{p}_2, \\ & skip \quad] & \mathrm{a}_2 \end{array}$$

The corresponding $weaveInstrSet$ will then be:

$$\begin{array}{l} \{\ insert(!adv) = \{before(!a)\} \\ \quad insert(?adv) = \{after(?a), before(!b)\}\} \end{array}$$

We will investigate three different base models. For *base model 1* we have two crossing messages a and b, and there are two possible traces: $\langle !a, !b, ?a, ?b\rangle$ and $\langle !a, !b, ?b, ?a\rangle$. However, none of these two traces contain matches for the single pointcut trace. The weaving terminates with no aspect application.

For *base model 2* we have the same two messages a and b in addition to a message c which crosses over the b message. We easily see without calculating the traces that there will be no matches for this base model. The reason is that the $!c$ event will always prevent matches, since it will split the pointcut matched events from a and b.

For *base model 3* we again have the same two messages a and b in addition to a message c. The c message crosses over both a and b. However, this does not prevent matches since it does not necessarily split any events from a and b. Consider the trace: $\langle !c, !a, ?a, !b, ?b, ?c\rangle$. In this trace we identify a match and can insert the advice parts. $!adv$ is defined to be inserted $before(!a)$, which implicitly also means before any previous events of $!a$ on the same lifeline. Thus $!adv$ must be placed between $!c$ and $!a$, while $?adv$ is defined to be both $after(?a)$ and $before(!b)$.

Although the figure does not have crossing messages in the pointcut, this is also valid as long as the partial order of the pointcut preserved elements are maintained in the advice (which is always a requirement for the aspect).

9.4 Example: Impossible automata weaving

The example shown in Figure 21 is taken from Klein et al. (Figure 12 in [6]). We have translated from Message Sequence Chart to sequence diagram and also added an advice diagram not present in their example. In their automata-based approach the base model leads to a non-regular expression (called unbounded sequence diagrams), which thus cannot be expressed as an automata, and their approach fails to handle this case.

The pointcut consists of two consecutive messages $m1$ and $m2$. The advice model contains the same two messages and a new adv message. The base model contains the same two messages but with an unbounded loop in between. We will now show that this example works fine in our trace-based solution.

One possible representation of the *weaveSorted* advice is:

$$\begin{array}{lll} seq[& skip, & \mathrm{a}_0 \\ & seq\,[!m1, ?m1, !m2, ?m2]\,, & \mathrm{p}_1 \\ & seq\,[!adv, ?adv] \quad] & \mathrm{a}_1 \end{array}$$

The corresponding $weaveInstrSet$ will then be:

$$insert(seq\,[!adv, ?adv]) = \{after(!m1), after(?m2)\}$$

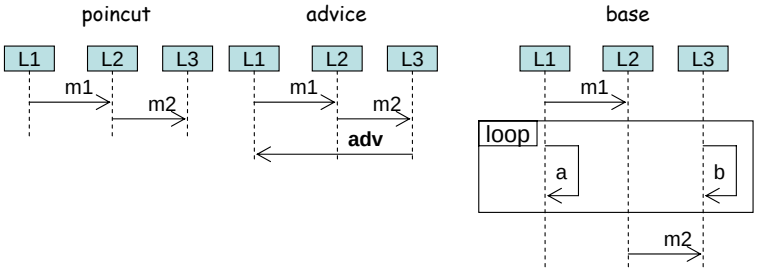

Figure 21: Impossible automata weaving: aspect and base

The unbounded loop in the base model will be sent to the $TreatLoops$ algorithm which finds out that this is an unrelated loop since it has no events in common with the pointcut. The $TreatLoops$ algorithm will call on the $rewriteUnrelated$ algorithm which simply introduces an *alt* operand with *skip* as the first operand and 1^+ iterations of the marked loop as the second operand. The marked loop, $loop'$, which will not be involved in any possible matches.

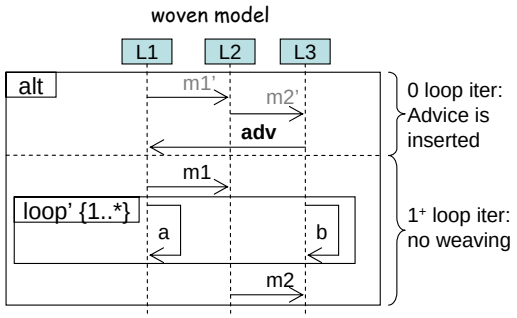

Figure 22: Impossible automata weaving: woven model

Figure 22 shows the final woven result. The marked, unbounded loop will be present in the final result as it will be part of the hybrid traces and in the reverse-engineered result from the hybrid traces. We end up with a woven model with

an outer *alt* operator. The first *alt* operand corresponds to zero loop iterations and will be affected by the advice. The second *alt* operand corresponds to 1^+ iterations of the loop and will not be affected by the advice.

9.5 Example: Disconnected pointcut

We do not allow for disconnected pointcuts. We will explore an example to see why our approach does not handle disconnected pointcuts properly. Figure 23 shows a pointcut with two messages that are placed on disjoint lifelines with no path from lifeline $L1$ to $L2$. The advice simply adds an *adv* message after the matched a and b messages. The base model consists of two consecutive and unbounded loops, where the first loop generates a messages, and the second generates b messages.

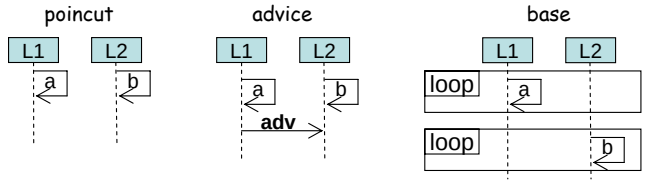

Figure 23: Disconnected pointcut is troublesome

The weave algorithm discovers that both these two loops are in the category non-MatchRepetitive loops, and thus they will be rewritten according to the *rewriteNonRep* algorithm. The result of this is shown in Figure 24. The traces with bounded loops are fairly trivial, so we will concentrate on the most complex part which is when we combine the two *alt* operands with unbounded loops. This part is shown in Figure 25 with two of the possible woven results. Remember that the marked, unbounded loops resulting from the rewriting will not be processed any further in the matching and weaving process.

In the first alternative, the advice has been added after the first two occurences of a and b messages, and after the last two occurences of a and b. This result is not correct since their will be unhandled matches for all loop iterations where both the loops iterate at least once.

In the second alternative, the advice has been added after the very last occurence of a and after the very first occurence of b. This woven result prevents all other possible matches and is thus correct in the sense that there will not be any unhandled matches in any of the traces generated from this woven result. This example both reveals that the woven results may be very different since the matching strategy is non-deterministic. Such a confluence problem may be dramatic in some cases, and acceptable in other cases. This issue should be investigated further.

The incorrect result which we may get (woven model 1) is unacceptable and is the reason why we do not allow for disconnected pointcuts. Even if we

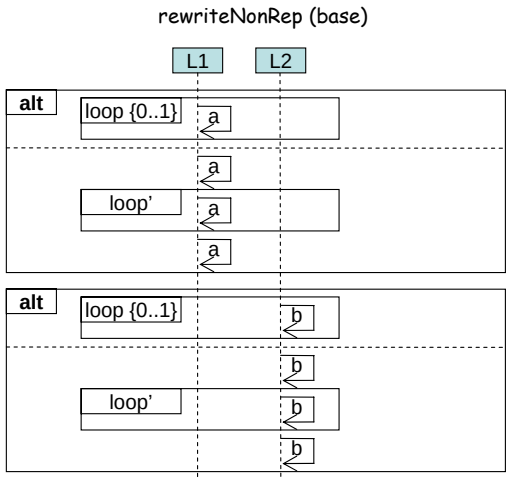

Figure 24: Disconnected pointcut: rewritten non-matchRepetitive loops

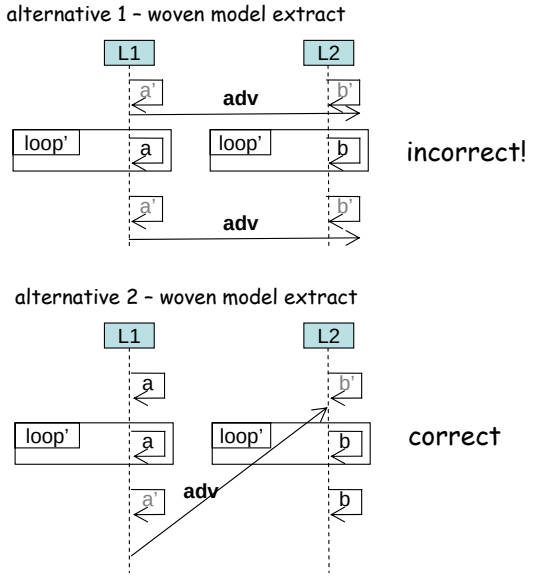

Figure 25: Disconnected pointcut: woven result may be incorrect

strengthened the approach to ensure that the woven model 2 would be chosen, this would not be good enough in general. Consider that the inserted advice in the example had transmitter and reception lifelines that were both different from $L1$ and $L2$, then we would only be able to produce incorrect woven results,

since the 1^+ loop iterations of the two unbounded loops will always produce unhandled matches.

The messages a and b have the same lifeline as transmission and reception. However, the weaving problems are the same if $L1$ was split into $L1_a$ and $L1_b$, and similar for $L2$.

10 Implementation

We are currently implementing a prototype tool to support the full approach described in this report. It reuses an Eclipse plugin SeDi sequence diagram editor v.1 [7] to define base, pointcut and advice diagrams. We build upon previous support for the STAIRS syntax model representation and the $[\![]\!]$ operator to calculate traces. Furthermore, we will be able to run the automated test programs by Lund and Stlen [8] since they take SeDi v.1 diagrams as input. Their test program may for instance compare if a diagram is a correct refinement of another diagram. The tool by Lund Stlen is just one example of tools that will behave similarly for semantically equivalent diagrams regardless of a poor structure and layout.

11 Related Work

Aspect-oriented behaviour modeling approaches so far have been dominated by UML sequence diagram attempts. This approach is supported by Clarke and Walker [1] with their Composition Patterns which are mapped to AspectJ.

Deubler et al. [3] and Solberg et al. [11] are examples of syntax-based approaches to aspects for sequence diagrams. Apart from being syntax-based, they also require special syntax for specifying join points, and this is accomplished by extensions to UML Sequence Diagrams.

J. Whittle and J. Arajo [13] has a little more advanced approach: It is still syntax based, but patterns of interactions (called Interaction Pattern Specifications - IPSs) are used in order to define point cuts. Patterns are defined in terms of specialized UML metamodel elements, and a matching interaction is an interaction with the same pattern, but whith some of the metamodel elements (the roles) bound to real model elements.

Klein et al. [6] allow for one outer hierarchical Message Sequence Chart (HMSC) consisting of arbitrary basic MSC (BMSC). Their approach is restricted to not use HMSCs inside an HMSC. Inside a BMSC only weak sequencing is allowed. We allow for arbitrary nesting of *alt*s which is not supported by their approach. The important restriction of allowing only connected pointcuts is adopted from their work to avoid some intractable weaving problems. However, with this assumption our algorithm will always terminate with a successful weaving (with the assumption of connected pointcuts) also in some cases with infinite loops where the Klein et al. approach fails (Figure 12 in [6]).

Klein et al [5] is a follow up of their work on semantics-based weaving in

that they allow for not just one aspect but several aspects. As they do not want the order of weaving of the aspects to have any significance, they have to allow events in between the events of a match. They present four different matching choices of which our matching definition corresponds to the *enclosed part*. Stein et al. [12] have developed the Join Point Designation Diagrams (JPDD) to capture pointcut expressions using mainly UML sequence diagrams and class diagrams. Their matching approach is expressive enough to allow events in between the events of a match.

Cottenier et al [2] is a quite different approach as it is based upon state machines, but the similarity is that pointcut matches are found in the implementation, while pointcuts are specified at the specification level. Many implementations may fulfill a specification, and as pointcuts are made at the specification level they become independent of evolving implementations. In our approach, many (different) base models may produce sets of traces where pointcut matches may be found.

12 Discussion

In this report we have restricted the base model to use only the *seq*, *alt* and *loop* operators. However, the results can be generalized to also be valid for several other sequence diagram operators. The *opt* (optional) and *par* (parallel) operators can be defined using an *alt* and is thus also supported in our approach. The *xalt* operator defines mandatory choice as opposed to the *alt* which defines potential choice. This difference is irrelevant to our approach and the *xalt* operator can be directly supported by treating it in the same way as with *alt*.

The *strict* operator is however, not supported since the reverse-engineering from traces back to sequence diagrams will fail. The reason is that the reverse-engineering handles each trace individually and has then no possibility to see the difference between weak sequencing and *strict*. We have simply assumed that weak sequencing has been applied originally. The reverse-engineering routine would be far more complex if we could not treat each trace isolated. We could however allow the base models to only use the *strict* operator and never use the weak sequencing, and easily adapt our approach to support this.

We have not stressed the performance and efficiency of the approach. The set of traces tend to increase dramatically with the number of messages in a sequence diagram. Also the use of the *par* operator and the *loops* have an exponential impact on the number of traces and the length of the traces. Although we have an approach that will always terminate (assuming the advice does not introduce additional matches) even for unbounded loops or loops with large upper bounds, there will be a practical limit on the kind of base models that our approach will handle within a reasonable time frame. We leave exploration of this issue to future work.

13 Conclusions

We have demonstrated that it is possible to do semantics-based aspect weaving for UML 2.0 sequence diagrams based upon a formal trace model for these. Semantics-based weaving implies that the pointcut matching is performed at the trace level ('what' the sequence diagrams really describe), and not at the syntactic level ('how' the sequence diagrams are described), but still with the convenience for the developer that the pointcut specifications can be done by means of syntactic elements of sequence diagrams.

Klein et al. [6] present another attempt to do semantics-based weaving of sequence diagrams which cover the same subset of secuence diagrams as we do, but while they cannot handle cases of infinite loops which leads to non-regular trace expressions, our approach also caters for support for general loops that are not finite. Our loop improvement is based on the observation that matches have a restricted length with respect to the poincut. We have established a systematic way to permutate and rewrite the original loop definition so that the weaving can be performed on a finite structure.

Acknowledgement

The work reported in this report has been funded by The Research Council of Norway, grant no. 167172/V30 (the SWAT project). We thank Mass Soldal Lund for valuable discussions and help with the implementation.

References

[1] S. Clarke and R. J. Walker. Composition Patterns: An Approach to Designing Reusable Aspects. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, Toronto, Ontario, Canada, 2001.

[2] T. Cottenier, A. van den Berg, and T. Elrad. Joinpoint Inference from Behavioral Specification to Implementation. In *ECOOP 2007 - Object-Oriented Programming, 21st European Conference*, Berlin, Germany, 2007.

[3] M. Deubler, M. Meisinger, S. Rittmann, and I. Krüger. Modeling Crosscutting Services with UML Sequence Diagrams. In *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2005)*, Montego Bay, Jamaica, 2005.

[4] R. Grosu and S. A. Smolka. Safety-Liveness Semantics for UML 2.0 Sequence Diagrams. In *Fifth International Conference on Application of Concurrency to System Design (ACSD 2005)*, St. Malo, France, 2005. IEEE Computer Society.

[5] J. Klein, F. Fleurey, and J.-M. Jézéquel. Weaving multiple aspects in sequence diagrams. *To appear in Trans. on Aspect Oriented Software Development*, 2007.

[6] J. Klein, L. Hélouët, and J.-M. Jézéquel. Semantic-based weaving of scenarios. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, Bonn, Germany, 2006.

[7] A. Limyr. Graphical editor for UML 2.0 sequence diagrams. Master's thesis, Department of Informatics, University of Oslo, 2005.

[8] M. S. Lund and K. Stølen. Deriving tests from UML 2.0 sequence diagrams with neg and assert. In *Proc. 1st International Workshop on Automation of Software Test (AST'06)*, 2006.

[9] O. M. G. (OMG). UML 2.0 Superstructure Specification, OMG Adopted Specification ptc/03-08-02, August 2003.

[10] R. K. Runde, Ø. Haugen, and K. Stølen. Refining UML interactions with underspecification and nondeterminism. *Nordic Journal of Computing*, 2(12), 2005.

[11] A. Solberg, D. Simmonds, R. Reddy, S. Ghosh, and R. B. France. Using Aspect Oriented Techniques to Support Separation of Concerns in Model Driven Development. In *29th Annual International Computer Software and Applications Conference (COMPSAC 2005)*, Edinburgh, Scotland, 2005.

[12] D. Stein, S. Hanenberg, and R. Unland. Join Point Designation Diagrams: a Graphical Representation of Join Point Selections. *International Journal of Software Engineering and Knowledge Engineering*, 16(3):317–346, 2006.

[13] J. Whittle and J. Araújo. Scenario modelling with aspects. *IEE Proceedings - Software*, 151(4):157–172, 2004.