A Coalgebraic Semantic Framework for Reasoning about UML Sequence Diagrams*

Sun Meng[1] and Luís S. Barbosa[2]
[1]CWI, Kruislaan 413, Amsterdam, The Netherlands
[2]Department of Informatics, Minho University, Portugal
M.Sun@cwi.nl, lsb@di.uminho.pt

Abstract

If, as a well-known aphorism states, modelling is for reasoning *, this paper is an attempt to define and apply a formal semantics to UML sequence diagrams in order to enable rigourous reasoning about them. Actually, model transformation plays a fundamental role in the process of software development, in general, and in model driven engineering in particular. Being a* de facto *standard in this area, UML is no exception, even if the number and diversity of diagrams expressing UML models makes it difficult to base its semantics on a single framework. This paper builds on previous attempts to base UML semantics in a coalgebraic setting and illustrates the application of the proposed framework to reason about composition and refactoring of sequence diagrams.*

1. Introduction

The aphorism *modelling is for reasoning* which, even if in an implicit way, underlies most research in Formal Methods, sums up the fundamental interconnection between *modelling* and *calculation*. The former is understood as the ability to choose the right abstractions for a problem domain. The latter, on the other hand, concerns the need for expressing such abstractions in a framework whose mathematical structure is sufficiently rich to enable rigourous reasoning either to establish models' properties or to transform models towards effective implementations.

Recalling such an interconnection seems particularly appropriate with respect to the formalisation attempts of UML. The number and diversity of diagrams expressing a UML model makes it difficult to base its semantics on a single framework. On the other hand, some of the formalisations proposed in the literature are essentially descriptive and difficult to use.

There are, at least, two levels at which the contribution of a formal semantics for the UML is deeply needed. One concerns model composition (their operators and the laws which govern their behaviour), the other model *refactoring*, i.e., model transformations which preserve external behaviour while improving their internal structure.

Originally introduced by Opdyke in [21] in the context of OO programming, refactoring has been widely used in modern software development processes such as Rational Unified Process [12] and eXtreme Programming [3] to support iterative software development and improve the quality of software artifacts. In [5] it is defined as "*the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure*". Later, interest in research shifted from the code level to model refactoring [25, 26], which is a rather new topic. A few references in the UML context include [15, 26, 29], although most of them restrict themselves to class diagrams.

This paper introduces a new, coalgebraic semantics for UML interaction models represented, as usual, by *sequence diagrams*. Moreover, a set of operators for such diagrams, which was proposed and informally described in [20], is formally characterised, therefore providing a calculus to reason about them. Finally, the paper discusses how both composition and refactoring laws for sequence diagrams can be dealt within the proposed framework. This extends previous work by the authors in seeking a unifying coalgebraic semantics for UML, as reported in [16, 19]. Those references introduced a semantics for class diagrams, use cases and statecharts based on coalgebras [23] taken as a suitable mathematical structure for expressing behaviour and state-based models. A similar approach is taken here for sequence diagrams. In all cases the coalgebraic point of view puts forward a well-defined notion of behaviour, as equiva-

* The work is partially supported by a grant from the GLANCE funding program of the Dutch National Organization for Scientific Research (NWO), through project CooPer (600.643.000.05N12).

1550-6002/08 $25.00 © 2008 IEEE
DOI 10.1109/QSIC.2008.13

lence classes for the bisimilarity relation induced by the particular functor used, upon which properties of UML models can be formulated and checked.

The organization of this paper is as follows. Section 2 provides a brief introduction to UML sequence diagrams. The coalgebraic semantic framework is given in section 3 and further developed in section 4 where an algebra of sequence diagrams' combinators is defined. The use of this semantics in reasoning about sequence diagrams is illustrated in section 5 through the discussion of some composition and refactoring results. Comparison with related work is made in Section 6. Section 7 concludes with a few remarks for future work.

2. UML sequence diagrams

This section provides a brief introduction to UML Sequence Diagrams, which are used to model the dynamic behavior of systems. Graphically, a UML sequence diagram, abbreviated to sd in the sequel, has two dimensions: a horizontal dimension representing the components participating in the scenario, and a vertical one representing time, i.e., the component temporal evolution or *lifeline*, represented by a vertical dashed line. Actually the focus of a sequence diagram lies on message interchange between a number of lifelines during a system run.

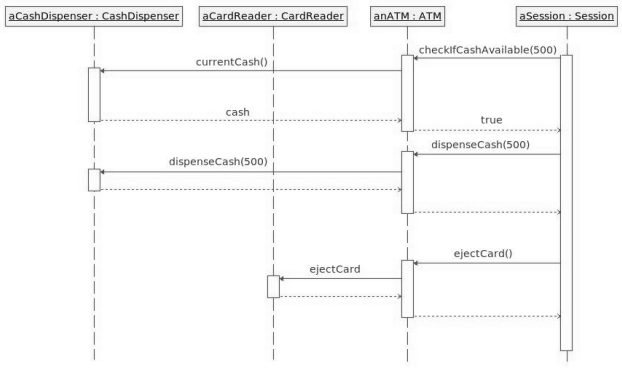

Figure 1. Sequence diagram for a cash withdraw scenario on an ATM

Figure 1 shows an example of a UML sequence diagram, which illustrates a cash withdraw scenario in an automatic teller machine (ATM). The diagram shows a possible interaction between instances of classes **CashDispenser**, **CardReader**, **ATM** and **Session**, when a user decides to make a withdraw and the amount of cash is dispensed successfully.

A message defines a particular communication between lifelines of an interaction. It can be either asynchronous (represented by an open arrow head) or synchronous (represented by a filled arrow head). Additionally, there are two special kinds of messages, *lost* and *found*, with the obvious meaning, which are described by a small black circle at the arrow head, or origin, respectively.

UML sequence diagrams may contain sub-interactions called *interaction fragments* that can be structured and combined using a number of so-called *interaction operators*. Although the semantics of an interaction fragment depends on the set of operators available, the precise definition of such a set is still an open topic in UML modelling. Recently, the UML superstructure specification [20] proposed one such set and gave an informal characterisation of the associated behaviours as follows:

- The operator **alt** offers a choice of behavior alternatives represented by its two operands. The chosen sd must have an explicit or implicit guard expression that evaluates to true at this point in the interaction.
- The operator **opt** designates a choice between the its (sole) operand or a idle behaviour.
- The operator **par** stands for the parallel merge of the behaviors of the sd acting as its operands. Event occurrences in the different operands can be interleaved in any way as long as the ordering imposed inside each sd is preserved.
- The operator **seq** represents a weak sequencing between the behaviors of the operands, i.e., the ordering of event occurrences within each of the operands is maintained in the result, whereas event occurrences on different lifelines in different operands may come in any order. Event occurrences on the same lifeline in different operands are ordered in such a way that an event occurrence of the first operand comes before that of the second operand.
- The operator **strict** represents a strict sequencing of the behaviors: all events in the first operand are made to occur before any event in the second.
- The **loop** operator specifies an iteration of strict sequential composition: the execution of its operand repeats itself on completion.

Such is the kernel of operators proposed in [20] for which the present paper seeks a suitable semantics. It should be stressed, however, that our purpose is essentially to show how a coalgebraic framework may provide an interesting semantic domain for this sort of combinators, rather than advocating the strict adoption of this OMG particular proposal. That includes, moreover, a few further operators that, by space limitation, will be omitted here. The most relevant

are a negation operator **neg** enforcing a diagram to represent invalid traces, an **ignore** operator allowing messages to appear anywhere along the lifelines of their participants, and a **critical** operator to represent a critical region in a diagram and therefore forbid interleaving of its events with any other event occurrences on the lifelines covered by the region.

3. A coalgebraic semantics

Graphically, a UML sequence diagram has two dimensions: a horizontal dimension representing the participants in the scenario, and a vertical dimension representing time. Participants evolve along *lifelines*, represented by vertical dashed lines. Interactions between participants are shown as horizontal arrows called *messages*. A message is a communication between two participants, and specifies both the type of communication (synchronous or asynchronous) and the associated sending and receiving event occurrences. Events situated on the same lifeline are ordered in time from top to down.

The signature of a UML sequence diagram is defined as follows:

Definition 3.1 *A sequence diagram sd is given by a tuple*

$$(I, Loc, Loc_{ini}, Mes, E, \leq)$$

where

- *I is a set of instance identifiers corresponding to the participants in the interaction described by the diagram;*
- *Loc is a set of locations;*
- *$Loc_{ini} \subseteq Loc$ is a set of initial locations;*
- *Mes is a set of message labels;*
- *$E \subseteq Loc \times Mes \times Loc$ is a relation such that tuple (l_1, m, l_2) represents a message m sent from location l_1 to location l_2.*
- *$\leq \subseteq Loc \times Loc$ is a partial order capturing the relative positions of locations within each of the diagram lifelines.*

Note that in general, for an edge to represent a communication between participants in a sequence diagram, its source and target locations can not be same, i.e., the following property is assumed:

$$\forall (l_1, m, l_2) \in E \ . \ l_1 \neq l_2$$

On the other hand, *local events*, which by definition are relative to a unique participant, are represented by reflexive edges at a particular location, e.g., (l, a, l).

Within this model the following functions return the set of locations of a particular participant and the next location in a particular lifeline. Formally,

- $loc : I \to 2^{Loc}$ associates to each instance a set of locations. The function satisfies the following conditions expressing disjointness and conformity with the initial constraints, respectively,

$$\forall i, j \in I, i \neq j \ . \ loc(i) \cap loc(j) = \emptyset \quad (1)$$
$$\forall i \in I \ . \ card(loc(i) \cap Loc_{ini}) = 1 \quad (2)$$

  where for a set S, $card(S)$ returns the cardinality of S.

- $next : Loc \to Loc$ is defined as

$$next(l) = l' \text{ iff } \exists i \in I \ . \ l, l' \in loc(i) \ \wedge \ l \leq l' \wedge \forall l'' \in loc(i) \ . \ l \leq l'' \Rightarrow l' \leq l''$$

Let l_1, l_2 range over Loc, and Σ_m be the set of communication events executed concerning messages exchanged in a sequence diagram sd. Such events have one of the following forms:

1. $\langle l_1! \to l_2, m \rangle$ - l_1 sends asynchronous message m to l_2,
2. $\langle l_1? \leftarrow l_2, m \rangle$ - l_1 receives asynchronous message m from l_2,
3. $\langle l_1! \blacktriangleright l_2, m \rangle$ - l_1 sends synchronous message m to l_2, and
4. $\langle l_1? \ \blacktriangleleft l_2, m \rangle$ - l_1 receives synchronous message m from l_2.

Note the cases for *lost* and *found* messages can be represented by replacing l_2 by $\bullet$ in the first two cases. Finally, let Σ_τ denote the set of local actions in a sequence diagram. Such actions have the form $\langle l \circlearrowleft a \rangle$ which means local action a happens at location l. The set of all events in a sequence diagram will be denoted by Σ and defined as $\Sigma = \Sigma_m \cup \Sigma_\tau$.

For any event $e \in \Sigma$, the location at which e happens is defined by $\pi(e) = l$ iff $e = \langle l(\cdots) \rangle$. This notation generalises for a set of events $\Sigma' \subseteq \Sigma$ as $\pi\Sigma' = \{\pi(e) \mid e \in \Sigma'\}$.

A configuration of a sequence diagram denotes a global state, composed of participants' local states, together with the current environment within which the system is supposed to interact. The first component describes which states are simultaneously active in the configuration. The second gives the environment specified as the set of active events.

Definition 3.2 *A configuration of a sequence diagram is a pair $\langle G, \Sigma_A \rangle$ where*

- *$G \in \prod_{i \in I} loc(i)$ is a tuple of participants' local states;*
- *$\Sigma_A \subseteq \Sigma$ denotes the current environment of active events.*

For an arbitrary configuration $\langle G, \Sigma_A \rangle$ and an event $e \in \Sigma_A$, if $\pi(e) \in G$, then e can occur in the configuration and make the system evolve [1]. Otherwise, e will stay in the set Σ_A until a new configuration forcing $\pi(e) \in G$ is reached or e is removed from the set Σ_A.

A semantics for sequence diagrams can then be defined in terms of coalgebras for functor

$$\mathsf{T}(X) = X^{\Sigma} \tag{3}$$

where notation X^{Σ} stands for the set of all functions from Σ to X.

The T-coalgebra corresponding to a particular sequence diagram sd is defined as $(C, \overline{\alpha} : C \longrightarrow C^{\Sigma})$, where C is the set of sd possible configurations, together with an *initial* configuration c_0. The latter given by the tuple of initial locations and the set of events initially active, i.e.,

$$c_0 \;=\; (\prod_{l \in Loc_{ini}} l, \{e \mid \pi(e) \in Loc_{ini}\})$$

Let us now define α, the curried version of $\overline{\alpha}$, by enumerating all possible transition schemes. First of all, note that, if an event e is not active in a configuration (G, Σ_A), i.e., $e \notin \Sigma_A$, it will not be executed until, by some other event occurrence, e is added to the set of active events. This case is captured by a trivial transition

$$\alpha((G, \Sigma_A), e) \;=\; (G, \Sigma_A)$$

When a local action a happens at location $l \in loc(i)$, the current location of participant i is changed to $next(l)$. Thus, for $e = \langle l \circlearrowright a \rangle$ where $l \in G$,

$$\begin{array}{l} \alpha((G, \Sigma_A), e) \;= \\ \quad (G[next(l)/l], \Sigma_A \setminus \{e\} \cup \{e' \mid \pi(e') = next(l)\}) \end{array}$$

For synchronous messages, the events modelling both sending and receiving occur simultaneously (i.e., in an atomic, non interruptible way): no other event can occur in between. So if the current configuration is (G, Σ_A) and both the sending event $e = \langle l_1! \blacktriangleright l_2, m \rangle$ and the corresponding receiving event $\langle l_2? \blacktriangleleft l_1, m \rangle$ are active, we have

$$\begin{array}{l} \alpha((G, \Sigma_A), e) \;= \\ \quad (G[next(l_1)/l_1, next(l_2)/l_2], \Sigma_A \setminus \{e, \langle l_2? \blacktriangleleft l_1, m \rangle\} \\ \quad \cup\; \{e' \mid \pi(e') = next(l_1) \vee \pi(e') = next(l_2)\}) \end{array}$$

For asynchronous messages, however, when the sending event occurs, the location of the sender will be updated to the next location in its lifeline, while locations of the other participants will remain unchanged. The sending event is therefore removed from the set of active events. On the other hand, the corresponding receiving event will be added to such set. Furthermore, the events at the next location of the sender's lifeline will become active in the new configuration. If $e = \langle l_1! \rightarrow l_2, m \rangle$ is active in configuration (G, Σ_A), we have

$$\begin{array}{l} \alpha((G, \Sigma_A), e) \;= \\ \quad (G[next(l_1)/l_1], \Sigma_A \setminus \{e\} \cup \{\langle l_2? \leftarrow l_1, m \rangle\} \\ \quad \cup\; \{e' \mid \pi(e') = next(l_1)\}) \end{array}$$

Dually, when an asynchronous message is received, the receiver will change to the next location in its lifeline, while locations of all other participants remain unchanged. Formally, if $e = \langle l_1? \leftarrow l_2, m \rangle$ is active in configuration (G, Σ_A), we have

$$\begin{array}{l} \alpha((G, \Sigma_A), e) \;= \\ \quad (G[next(l_1)/l_1], \Sigma_A \setminus \{e\} \cup \{e' \mid \pi(e') = next(l_1)\}) \end{array}$$

The case of a lost message, represented by event $e = \langle l! \rightarrow \bullet, m \rangle$, is similar to the asynchronous communication: the sender updates its location and e is removed from the set of active events. However, no corresponding receiving event becomes active. Similarly, for a found message, when a receiving event $e = \langle l? \leftarrow \bullet, m \rangle$ occurs, only the location of the receiver is updated and e is removed from the set of active events. Both cases are, therefore, handled by

$$\begin{array}{l} \alpha((G, \Sigma_A), e) \;= \\ \quad (G[next(l)/l], \Sigma_A \setminus \{e\} \cup \{e' \mid \pi(e') = next(l)\}) \end{array}$$

assuming the corresponding events are enabled in configuration (G, Σ_A).

4. An algebra of UML sequence diagrams

In the previous section the semantics of an arbitrary sequence diagram sd was defined by a triple

$$[\![sd]\!] = (C, \overline{\alpha} : C \to C^{\Sigma}, c_0)$$

where C is the set of configurations of sd. The next step consists of defining the denotations of the *interaction operators* proposed in [20] for combining UML sequence diagrams, as recalled in section 2. This formalises an algebra *to build new sequence diagrams from old*.

In the sequel, we assume, for a sequence diagram sd_i, that $[\![sd_i]\!] = (C_i, \overline{\alpha_i}, c_0^i)$, where $C_i = \{(G_i, \Sigma_A^i)\}$. Here G_i denotes the tuple of local states of the participants in sd_i, and Σ_A^i the set of current active events in sd_i. Moreover, we let $c_0^i = (G_i^0, \Sigma_0^i)$. For a tuple of elements $t = (e_1, e_2, \cdots, e_m)$, we resort to projection function π_i, for $i = 1, \cdots, m$, to return the i-th element e_i. With such notational conventions we are prepared to give the semantics of all operators considered in section 2.

1 Here $\pi(e) \in G$ means $\pi(e)$ is a location in the tuple G.

Choice: $\mathbf{alt}(sd_1, sd_2)$.
Denoting an alternative form of aggregation of sequence diagrams, it is required that $G_1^0 = G_2^0$, and that all events in both Σ_0^1 and Σ_0^2 become active in the initial configuration c_0. Therefore, $c_0 = (G_1^0, \Sigma_0^1 \cup \Sigma_0^2)$. Furthermore, $C = \{c_0\} \cup (C_1 \setminus \{c_0^1\}) \cup (C_2 \setminus \{c_0^2\})$. Formally,

$$[\![\mathbf{alt}(sd_1, sd_2)]\!] = (C, \overline{\mathbf{alt}(\alpha_1, \alpha_2)}, c_0)$$

with $\mathbf{alt}(\alpha_1, \alpha_2)$ given by[2]

$$\mathbf{alt}(\alpha_1, \alpha_2)(x, e) = \begin{cases} x = c_0 \wedge e \in \Sigma_i \Rightarrow & \alpha_i(c_0^i, e) \text{ for } i = 1, 2 \\ x \in C_1 \wedge e \in \Sigma_1 \Rightarrow & \alpha_1(x, e) \\ x \in C_2 \wedge e \in \Sigma_2 \Rightarrow & \alpha_2(x, e) \\ \text{otherwise} & x \end{cases}$$

where $x = (G, \Sigma_A)$ is a configuration in C, and e is an event in either Σ_1 or Σ_2.

Option: $\mathbf{opt}(sd_1)$.
The purpose of $\mathbf{opt}(sd_1)$ is to offer an alternative between an empty scenario (in which 'nothing happens') and the activation of its (sole) operand, sd_1. To formalise its meaning we need to introduce a new event — $skip$ — into the set of events to capture absence of effective behaviour. Then

$$[\![\mathbf{opt}(sd_1)]\!] = (C, \overline{\mathbf{opt}(\alpha_1)}, c_0)$$

where $C = C_1 \setminus \{c_0^1\} \cup \{(G_1^0, \Sigma_0^1 \cup \{skip\})\}$ and $c_0 = (G_1^0, \Sigma_0^1 \cup \{skip\})$. The transition structure is defined as

$$\mathbf{opt}(\alpha_1)(x, e) = \begin{cases} e \in \Sigma_1 \Rightarrow & \alpha_1(x, e) \\ x = c_0 \wedge e = skip \Rightarrow & (G_1^0, \emptyset) \\ \text{otherwise} & x \end{cases}$$

Parallel: $\mathbf{par}(sd_1, sd_2)$.
In this case we consider

$$C = \{(G_1 \times G_2, \Sigma_A^1 \cup \Sigma_A^2) \mid \text{for } i = 1, 2, (G_i, \Sigma_A^i) \in C_i\}$$

and

$$c_0 = (G_1^0 \times G_2^0, \Sigma_0^1 \cup \Sigma_0^2)$$

in

$$[\![\mathbf{par}(sd_1, sd_2)]\!] = (C, \overline{\mathbf{par}(\alpha_1, \alpha_2)}, c_0)$$

where the transition structure is defined as

$$\mathbf{par}(\alpha_1, \alpha_2)(x, e) = \begin{cases} e \in \Sigma_1 \Rightarrow & \textbf{let } x' = \alpha_1((\pi_1\pi_1 x, \pi_2 x \mid_{\Sigma_1}), e) \textbf{ in} \\ & ((\pi_1 x', \pi_2\pi_1 x), \pi_2 x' \cup \pi_2 x \mid_{\Sigma_2}) \\ e \in \Sigma_2 \Rightarrow & \textbf{let } x' = \alpha_2((\pi_2\pi_1 x, \pi_2 x \mid_{\Sigma_2}), e) \textbf{ in} \\ & ((\pi_1\pi_1 x, \pi_1 x'), \pi_2 x' \cup \pi_2 x \mid_{\Sigma_1}) \\ \text{otherwise} & x \end{cases}$$

Strict sequential composition: $\mathbf{strict}(sd_1, sd_2)$.
The transition structure in

$$[\![\mathbf{strict}(sd_1, sd_2)]\!] = (C, \overline{\mathbf{strict}(\alpha_1, \alpha_2)}, c_0)$$

is defined over $C = C_1 \cup C_2$ and $c_0 = c_0^1$ as follows

$$\mathbf{strict}(\alpha_1, \alpha_2)(x, e) = \begin{cases} x \in C_1 \Rightarrow & \textbf{let } x' = \alpha_1(x, e) \textbf{ in} \\ & \quad \pi_2 x' = \emptyset \Rightarrow c_0^2 \\ & \quad \text{otherwise } x' \\ \text{otherwise} & \alpha_2(x, e) \end{cases}$$

Weak sequential composition: $\mathbf{seq}(sd_1, sd_2)$.
The case for weak sequencing $\mathbf{seq}(sd_1, sd_2)$ for $sd_i = (I_i, Loc_i, Loc_{ini}^i, Mes_i, E_i, \leq_i), i = 1, 2$ is a bit more demanding because its definition depends on whether the operands share a number of lifelines. If such is the case, i.e., if an identifier, say s, exists in $I_1 \cap I_2$, then all the event occurrences on s in sd_1 should happen before those on s in sd_2. However, any other events on lifelines out of the scope of both sd_1 and sd_2, may occur in any order. Note that if the operands involve disjoint sets of participants, the weak sequencing reduces to a parallel merge.

Assume an identifier s, such that $I_1 \cap I_2 = \{s\}$, and function loc_1 and loc_2 assigning locations to instances in sd_1 and sd_2, respectively. Let $loc(s) = loc_1(s) \cup loc_2(s)$. Furthermore, and without loss of generality, let

$$C_1 = \{(G_1, \Sigma_A^1) \mid G_1 \in loc_1(s) \times L\}$$

and

$$C_2 = \{(G_2, \Sigma_A^2) \mid G_2 \in loc_2(s) \times K\}$$

be the set of configurations for sd_1 and sd_2 respectively, where $L = \prod_{i \in I_1 \setminus \{s\}} loc_1(i)$ and $K = \prod_{j \in I_2 \setminus \{s\}} loc_2(j)$. Then, define

$$[\![\mathbf{seq}(sd_1, sd_2)]\!] = (C, \overline{\mathbf{seq}(\alpha_1, \alpha_2)}, c_0)$$

with

$$C = \{(G, \Sigma_A) \mid G \in loc(s) \times L \times K$$

and

$$\Sigma_A = \Sigma_A^1 \cup \Sigma_A^2 \setminus \{e \mid \pi(e) \in loc(s) \wedge \pi(e) \neq \pi_1(G)\}\}$$

2 To avoid an excessive notational burden, we use the same syntax for the combinator over sequence diagrams and its denotation in the proposed semantics.

if there are two locations $a, b \in loc(s)$ such that $((a, \pi_2 G), \Sigma_A^1) \in C_1 \wedge ((b, \pi_3 G), \Sigma_A^2) \in C_2 \wedge (\pi_1 G = a \vee \pi_1 G = b)$. Finally, define $c_0 = ((\pi_1 c_0^1, \pi_2 \pi_1 c_0^2), \pi_2 c_0^1 \cup (\pi_2 c_0^2 \mid_{-\{s\}}))$. Notice the use of notation $\Sigma \mid_{-\{s\}}$ to denote the subset of Σ obtained by removing all the events occurring at some location in $loc(s)$, i.e.,

$$\Sigma \mid_{-\{s\}} = \{e \mid e \in \Sigma \wedge \pi(e) \notin loc(s)\}$$

The transition structure is given by

$$\begin{array}{l}
\mathbf{seq}(\alpha_1, \alpha_2)(x, e) = \\
\quad \mathbf{let}\ \{s\} = I_1 \cap I_2 \\
\quad \left\{ \begin{array}{l}
\pi(e) \in loc(s) \Rightarrow \\
\quad \left\{ \begin{array}{l}
\pi\pi_2(x) \cap loc_1(s) \neq \emptyset \Rightarrow \\
\quad \mathbf{let}\ x' = \alpha_1(x, e)\ \mathbf{in} \\
\quad \pi\pi_2 x' \cap loc_1(s) = \emptyset \Rightarrow \\
\quad \mathbf{let}\ c_0^2 = ((l, t), A)\ \mathbf{in} \\
\quad ((l, \pi_2\pi_1 x', t), \pi_2 x' \cup A) \\
\quad \text{otherwise } x' \\
\text{otherwise } \alpha_2(x, e)
\end{array} \right. \\
\pi(e) \notin loc(s) \Rightarrow \\
\quad \left\{ \begin{array}{l}
e \in \Sigma_1 \Rightarrow \\
\quad \mathbf{let}\ x' = \alpha_1((\langle \pi_1, \pi_2 \rangle \pi_1(x), \pi_2(x) \mid_{\Sigma_1}), e)\ \mathbf{in} \\
\quad ((\pi_1 x', \pi_3 \pi_1 x), \pi_2 x' \cup (\pi_2 x \mid_{\Sigma_2})) \\
e \in \Sigma_2 \Rightarrow \\
\quad \mathbf{let}\ x' = \alpha_2((\langle \pi_1, \pi_3 \rangle \pi_1(x), \pi_2(x) \mid_{\Sigma_2}), e)\ \mathbf{in} \\
\quad ((\pi_1 \pi_1 x', \pi_2 \pi_1 x, \pi_2 \pi_1 x'), \pi_2 x' \cup (\pi_2 x \mid_{\Sigma_1}))
\end{array} \right.
\end{array} \right.
\end{array}$$

The definition can be easily generalized to an arbitrary number of shared lifelines in sd_1 and sd_2.

On the other hand, if $I_1 \cap I_2 = \emptyset$, the definition of the transition structure reduces to the second branch of the case structure. By redefining the projection functions (since there is no s in the configurations), we can find that

$$\mathbf{seq}(sd_1, sd_2) = \mathbf{par}(sd_1, sd_2) \tag{4}$$

Furthermore, whenever $I_1 = I_2$, we have

$$\mathbf{seq}(sd_1, sd_2) = \mathbf{strict}(sd_1, sd_2) \tag{5}$$

The above equalities are in fact *bisimulation* equations between the corresponding denotations, i.e., for example, for equation (4),

$$[\![\mathbf{seq}(sd_1, sd_2)]\!] \sim [\![\mathbf{par}(sd_1, sd_2)]\!]$$

as such is the notion of equality in a coalgebraic setting.

They are, therefore, the first illustration of a *calculus* of sequence diagrams made possible by the semantic definition. The issue is further discussed in section 5.

Loop: $\mathbf{loop}(sd_1)$.

Finally, the semantics of the iteration combinator is given by

$$[\![\mathbf{loop}(sd_1)]\!] = (C, \overline{\mathbf{loop}(\alpha_1)}, c_0)$$

over $C = C_1$ and $c_0 = c_0^1$, and with the following transition structure

$$\begin{array}{l}
\mathbf{loop}(\alpha_1)(x, e) = \\
\quad \left\{ \begin{array}{ll}
e \in \pi_2 x \Rightarrow & \mathbf{let}\ x' = \alpha_1(x, e)\ \mathbf{in} \\
& \quad \pi_2 x' = \emptyset \Rightarrow c_0 \\
& \quad \text{otherwise } x' \\
\text{otherwise} & x
\end{array} \right.
\end{array}$$

5. Reasoning about sequence diagrams

5.1. Towards a calculus of diagram composition

Equations (4) and (5) above were our first examples of properties which establish, under suitable conditions, the equality of behaviour between expressions denoting arbitrary compositions of UML sequence diagrams. As mentioned there, such equalities are, in fact, bisimulation equations relating the coalgebras which represent the diagrams' semantics.

In coalgebra theory [23] a bisimulation between two coalgebras α and β is a relation R which preserves the transition structure, i.e., which is closed for the coalgebra dynamics. In general, for an arbitrary functor T, such a relation satisfies the following inequality

$$\alpha \cdot R \subseteq \mathsf{T}\, R \cdot \beta \tag{6}$$

where the dot denotes relational composition and $\mathsf{T}\, R$ is the image of relation R under functor T[3]. For the particular case of functor T defined in (3), however, definition (6) boils down to

$$(c, d) \in R \Rightarrow \forall_{e \in \Sigma} \,.\, (\alpha(c, e), \beta(d, e)) \in R \tag{7}$$

for every pair of configurations (c, d). This provides a rather simple way of testing behavioural equivalence for (the denotations of) UML sequence diagrams.

Not surprisingly some simple proofs, which proceed by the construction of a witnessing bisimulation, establish a number of algebraic laws relating different composition patterns. For example, one gets, commutativity and associativity for **alt**

$$\mathbf{alt}(sd_1, sd_2) = \mathbf{alt}(sd_2, sd_1) \tag{8}$$

$$\mathbf{alt}(\mathbf{alt}(sd_1, sd_2), sd_3) = \mathbf{alt}(sd_1, \mathbf{alt}(sd_2, sd_3)) \tag{9}$$

3 See [2] for a detailed, generic account of bisimulations and their calculational properties.

and, similarly, for the **par** and **strict** combinators (commutativity is satisfied only by **par**). For illustration purposes, let us prove now equation (9) and the commutativity result for parallel composition, i.e. the **par** version of equation (8):

$$\mathbf{par}(sd_1, sd_2) \;=\; \mathbf{par}(sd_2, sd_1) \tag{10}$$

Notice how in the second proof a quite handy technique of coinductive reasoning is used: to establish bisimilarity it is enough to define a coalgebra morphism connecting the two coalgebras. Such a technique, based on the fact that coalgebra morphisms entail bisimulation, is used extensively in, e.g., [1] to investigate the structure of a calculus for software components. Recall that a coalgebra morphism is a function h between their state spaces that preserves and reflects the transition structure, i.e. such that

$$\mathsf{T}\, h \cdot \beta \;=\; \alpha \cdot h \tag{11}$$

Proof: (of equation (9)).
We have to verify that

$$[\![\mathbf{alt}(\mathbf{alt}(sd_1, sd_2), sd_3)]\!] \sim [\![\mathbf{alt}(sd_1, \mathbf{alt}(sd_2, sd_3))]\!]$$

The set of configurations for both sides of this equation is $C = \{c_0\} \cup \bigcup_{1 \le i \le 3}(C_i \setminus \{c_0^i\})$, and the initial configuration, also in both cases, is c_0. For any $x \in C$ and event e one gets, according to the definition,

$$\begin{aligned}
&\mathbf{alt}(\mathbf{alt}(\alpha_1, \alpha_2), \alpha_3)(x, e) \\
&= \begin{cases}
x = c_0 \wedge e \in \Sigma_i \Rightarrow & \alpha_i(c_0^i, e) \text{ for } i = 1, 2, 3 \\
x \in C_1 \wedge e \in \Sigma_1 \Rightarrow & \alpha_1(x, e) \\
x \in C_2 \wedge e \in \Sigma_2 \Rightarrow & \alpha_2(x, e) \\
x \in C_3 \wedge e \in \Sigma_3 \Rightarrow & \alpha_3(x, e) \\
\text{otherwise} & x
\end{cases} \\
&= \mathbf{alt}(\alpha_1, \mathbf{alt}(\alpha_2, \alpha_3))(x, e)
\end{aligned}$$

□

and
Proof: (of equation (10)).
Again our task is to verify the bisimulation equation

$$[\![\mathbf{par}(sd_1, sd_2)]\!] \;\sim\; [\![\mathbf{par}(sd_2, sd_1)]\!]$$

The sets of configurations for $[\![\mathbf{par}(sd_1, sd_2)]\!]$ and $[\![\mathbf{par}(sd_2, sd_1)]\!]$ are $C_1 = \{(G_1 \times G_2, \Sigma_A^1 \cup \Sigma_A^2)\}$ and $C_2 = \{(G_2 \times G_1, \Sigma_A^2 \cup \Sigma_A^1)\}$ respectively, where (G_i, Σ_A^i) is a configuration of sd_i for $i = 1, 2$. Define $h : C_1 \to C_2$ as $h = \langle\langle \pi_2, \pi_1\rangle \cdot \pi_1, \pi_2\rangle$. To prove the bisimulation equation, we only need to show that h is a coalgebra morphism, i.e., $h \cdot \mathbf{par}(\alpha_1, \alpha_2)(x, e) = \mathbf{par}(\alpha_2, \alpha_1)(h(x), e)$ for any configuration x and event e. According to the definition of **par**, for $e \in \Sigma_1$,

$$\begin{aligned}
&\mathbf{par}(\alpha_2, \alpha_1)(h(x), e) \\
=&\mathbf{let}\ x' = \alpha_1((\pi_2\pi_1 h(x), \pi_2 x \mid_{\Sigma_1}), e)\ \mathbf{in} \\
&((\pi_1\pi_1 h(x), \pi_1 x'), \pi_2 x' \cup \pi_2 h(x) \mid_{\Sigma_2}) \\
=&\mathbf{let}\ x' = \alpha_1((\pi_1\pi_1 x, \pi_2 x \mid_{\Sigma_1}), e)\ \mathbf{in} \\
&((\pi_2\pi_1 x, \pi_1 x'), \pi_2 x' \cup \pi_2 x \mid_{\Sigma_2}) \\
=&\mathbf{let}\ x' = \alpha_1((\pi_1\pi_1 x, \pi_2 x \mid_{\Sigma_1}), e)\ \mathbf{in} \\
&h((\pi_1 x', \pi_2\pi_1 x), \pi_2 x' \cup \pi_2 x \mid_{\Sigma_2}) \\
=&h \cdot \mathbf{par}(\alpha_1, \alpha_2)(x, e)
\end{aligned}$$

Similarly, for $e \in \Sigma_2$, we also get $\mathbf{par}(\alpha_2, \alpha_1)(h(x), e) = h \cdot \mathbf{par}(\alpha_1, \alpha_2)(x, e)$. And for $e \notin \Sigma_1 \cup \Sigma_2$, the result is obvious: $h(x) = h(x)$. Furthermore, it is easy to obtain the result about initial configurations $h(c_0^1) = c_0^2$. Thus the law is proved. □

Following a similar strategy, one can prove, for example, idempotence results, reductions and, in particular, distribution of strict sequential and parallel composition over choice. Formally,

$$\mathbf{alt}(sd, sd) \;=\; \mathbf{sd} \tag{12}$$

$$\mathbf{alt}(sd, \emptyset_{I_{sd}}) \;=\; \mathbf{opt} \tag{13}$$

$$\begin{aligned}
&\mathbf{strict}(\mathbf{alt}(sd_1, sd_2), sd_3) \\
&\qquad = \mathbf{alt}(\mathbf{strict}(sd_1, sd_3), \mathbf{strict}(sd_2, sd_3))
\end{aligned} \tag{14}$$

$$\begin{aligned}
&\mathbf{strict}(sd_1, \mathbf{alt}(sd_2, sd_3)) \\
&\qquad = \mathbf{alt}(\mathbf{strict}(sd_1, sd_2), \mathbf{strict}(sd_1, sd_3))
\end{aligned} \tag{15}$$

$$\begin{aligned}
&\mathbf{par}(\mathbf{alt}(sd_1, sd_2), sd_3) \\
&\qquad = \mathbf{alt}(\mathbf{par}(sd_1, sd_3), \mathbf{par}(sd_2, sd_3))
\end{aligned} \tag{16}$$

$$\begin{aligned}
&\mathbf{par}(sd_1, \mathbf{alt}(sd_2, sd_3)) \\
&\qquad = \mathbf{alt}(\mathbf{par}(sd_1, sd_2), \mathbf{par}(sd_1, sd_3))
\end{aligned} \tag{17}$$

In equation (13) we use $\emptyset_{I_{sd}}$ to denote the empty sequence diagram with the same set of participants as sd, but no events. Suppose sd is given by $(I, Loc, Loc_{ini}, Mes, E, \leq)$, then $\emptyset_{I_{sd}} = (I, Loc_{ini}, Loc_{ini}, \emptyset, \emptyset, =)$.

5.2. Refactoring

If the previous sub-section intended to illustrate how a calculus of UML sequence diagrams operators can emerge from the proposed semantics, we shall focus now in the other kind of application mentioned in the Introduction to this paper: *refactoring*. Again we shall not be exhaustive, but rather suggest possible steps in this direction.

Actually, typical refactoring laws are supposed to preserve behaviour and therefore they boil down to bisimulation equations, as the ones considered above. Well-known examples are laws expressing fine grained refactoring steps such as adding, removing and moving elements in sequence diagrams. For example,

Law 5.1 *A new lifeline can be introduced into a sequence diagram.*

Proof:
Suppose $sd = (I, Loc, Loc_{ini}, Mes, E, \leq)$ is a sequence diagram. Adding a new lifeline to sd means that a new instance identifier i is added to I. Since there is no message exchanges between i and other participants in the diagram, it has only one location, i.e., the initial location l_0^i. So the resulting diagram is $sd' = (I \cup \{i\}, Loc \cup \{l_0^i\}, Loc_{ini} \cup \{l_0^i\}, Mes, E, \leq)$. If (G, Σ_A) is a configuration for sd, then $(\langle G, l_0^i \rangle, \Sigma_A)$ is a configuration for sd'. Let $h = \pi_1 \times \mathsf{id}$. This morphism maps every configuration of sd' to a configuration of sd, and forms a coalgebra morphism between them, which justifies the law. □

The very same argument justifies the corresponding law for removing lifelines:

Law 5.2 *A lifeline which does not interact with other participants and has no local actions can be removed from a sequence diagram.*

Other refactoring laws, however, may require a sort of weaker preservation of behaviour. Such is the case, for example, of refactorings involving the split of a lifeline into a set of independent lifelines representing sections of non-interfering execution and enforcing time constraints by specific message exchange.

In the semantic framework discussed here, such weak preservation of behaviour corresponds to relating (denotations of) sequence diagrams by refinement, instead of bisimilarity. Refinement for coalgebras has been studied by the authors in [17, 18]. In brief, the idea is to replace the coalgebra morphism condition in (11) by

$$\mathsf{T}\, h \cdot \beta \;\leq\; \alpha \cdot h \tag{18}$$

where $\leq$ is a so-called *refinement preorder* [18]. Function h is said to be a *forward morphism* which is intended to preserve transitions from the source coalgebra, but fails to reflect them back. Relation $\leq$, for functor T given in (3), is a preorder on functions from events to configurations. A possible example would require the images of the same event e under the semantics $[\![sd_i]\!] = (G_i, \Sigma_A^i)$, for $i = 1, 2$, of diagrams sd_1 and sd_2, respectively, being related by

$$G_1 = G_2 \;\wedge\; \Sigma_A^1 \subseteq \Sigma_A^2$$

which allows one of the diagrams to possess less active events than the other in some configurations.

In the references cited above, forward morphisms are shown to compose and enjoy a number of calculational properties. In particular they are powerful enough (more exactly, weak enough!) to capture all the refactoring situations for sequence diagrams one can think of, as refinement results.

6. Related Work

Sequence diagrams originate from message sequence charts (MSCs) [9]. Variants of MSCs, such as Live Sequence Charts (LSCs) [4], triggered MSCs [24] and template MSCs [7], are being widely used to capture behavioral requirements in Software Engineering. There is a number of approaches to formalize such scenario descriptions in order to facilitate the analysis of requirements or specifications. Starting from MSCs, Uchitel et al. [27, 28] specified semantics for HMSCs, and developed an approach to synthesise behavioural models in the form of labelled transition systems. Their approach aims at preserving the component structure of the system. This causes their models to allow for additional behaviours, which were not explicitly specified in the scenarios.

There is also a lot of research on providing semantics to the scenario descriptions. LSC has a well-defined operational semantics and a tool called Play-Engine [8] allows a user to construct LSCs by playing in scenarios and checking them through a play out mechanism. However, the semantics and tools do not consider concurrent interactions and verification. J. Küster-Filipe defines the semantics of UML 2.0 sequence diagrams by using labelled event structures, and presents a distributed concurrent logic for reasoning about interactions in [14]. A number of approaches for synthesis of state-based models from scenario descriptions has been developed. For example, the authors of [13] present a state-chart synthesis algorithm, but the approach does not support High-Level Message Sequence Charts (HMSC), which provide a composition mechanism very close to UML 2.0 sequence diagrams. A translation from UML 2.0 interactions into a special class of automata is presented in [11]. An algebraic approach for synthesizing statecharts from UML 2.0 sequence diagrams is also discussed in [30]. Both, however, omit a number of interaction fragments actually permitted in UML.

The semantics given here is novel and offers potential benefits concerning reasoning behavior of UML models. Together with our previous work, the coalgebraic framework offers a unifying semantics for different UML models. Furthermore, a coalgebraic semantics leads to a proof style (coinduction) which provides an elegant way to check the correctness of model transformation, i.e., behavior preservation. Both bisimulation and refinement between models can be easily established by using morphisms (and forward morphisms).

7. Conclusion and Future Work

This paper proposes a semantic framework for UML sequence diagrams which is consistent with the authors' previous work on coalgebraic semantics for other UML mod-

els. The recently proposed set of combinators for sequence diagrams is formalised in this framework. A number of laws describing the theory underlying such combinators is also discussed. It is argued that identical principles can be followed to establish model refactoring as (coalgebraic) refinements.

A detailed investigation and classification of possible refactoring patterns and their formalization in this framework, is our main plan for future work. Moreover, it would be interesting to study the relationship between UML model transformations and other sorts of refinement notions, either proposed within a coalgebraic setting (as, for example, in [10]), or emerging from research on software architecture (as in [22] or [6]). We also hope to extend the coalgebraic framework to deal with QoS aspects, like timing constraints, in UML models.

References

[1] L. S. Barbosa. Towards a calculus of state-based software components. *Journal of Universal Computer Science*, 9(8):891–909, August 2003.

[2] L. S. Barbosa, J. N. Oliveira, and A. M. Silva. Calculating invariants as coreflexive bisimulations. In *12th Int. Conf. Algebraic Methods and Software Technology (AMAST)*. Springer Lect. Notes Comp. Sci. (to appear), 2008.

[3] K. Beck. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley, 2004.

[4] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(0), 2001.

[5] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[6] D. Garlan. Formal modeling and analysis of software architecture: Components, connectors and events. In M. Bernardo and P. Inverardi, editors, *Third International Summer School on Formal Methods for the Design of Computer, Communication and Software Systems: Software Architectures (SFM 2003)*. Springer Lect. Notes Comp. Sci, Tutorial, (2004), Bertinoro, Italy, September 2003.

[7] B. Genest, M. Minea, A. Muscholl, and D. Peled. Specifying and verifying partial order properties using template mscs. In *Foundations of Software Science and Computation Structures, 7th International Conference, FOSSACS 2004*, volume 2987, pages 195–210. Springer, 2004.

[8] D. Harel and R. Marelly. *Come, Let's Play: Scenario-based Programming using LSCs and the Play-Engine*. Springer, 2003.

[9] ITU-TS. Recommendation Z.120(11/99) : MSC 2000, 1999. Geneva.

[10] B. Jacobs and H. Tews. Assertional and behavioural refinement in coalgebraic specification. In *Electronic Notes in Theoretical Computer Science*, volume 47. Elsevier Science Publishers, 2001.

[11] A. Knapp and J. Wuttke. Model checking of UML 2.0 interactions. In T. Kühne, editor, *Models in Software Engineering, Workshops and Symposia at MoDELS 2006, Genoa, Italy, October 1-6, 2006, Reports and Revised Selected Papers*, volume 4364 of *LNCS*, pages 42–51. Springer, 2007.

[12] P. Kruchten. *The Rational Unified Process: An Introduction (3rd edition)*. Addison-Wesley, 2003.

[13] I. Krüger, R. Grosu, P. Scholz, and M. Broy. From MSCs to statecharts. In *Distributed and Parallel Embedded Systems*, pages 61–72. Kluwer, 1999.

[14] J. Küster-Filipe. Modelling concurrent interactions. *Theoretical Computer Science*, 351(2):203–220, 2006.

[15] S. Marković and T. Baar. Refactoring ocl annotated UML class diagrams. In *Proceedings of MoDELS 2005*, number 3713 in LNCS, pages 280–294. Springer-Verlag, 2005.

[16] S. Meng, B. K. Aichernig, L. S. Barbosa, and Z. Naixiao. A coalgebraic semantic framework for component based development in UML. In *Proceedings of CTCS'04*, volume 122 of *ENTCS*, pages 229–245. Elsevier Science Publishers, 2005.

[17] S. Meng and L. S. Barbosa. On refinement of generic state-based software components. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Proceedings of AMAST'04*, volume 3116 of *LNCS*, pages 506–520. Springer, 2004.

[18] S. Meng and L. S. Barbosa. Components as coalgebras: The refinement dimension. *Theor. Comp. Sci.*, 351:276–294, 2006.

[19] S. Meng, Z. Naixiao, and L. S. Barbosa. On semantics and refinement of UML statecharts: A coalgebraic view. In J. R. Cuellar and Z. Liu, editors, *SEFM2004, 2nd International Conference on Software Engineering and Formal Methods*, pages 164–173. IEEE Computer Society, 2004.

[20] Object Management Group. *Unified Modeling Language: Superstructure - version 2.1.1*, 2007. `http://www.uml.org/`.

[21] W. F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Framewoks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[22] J. Philipps and B. Rumpe. Refinement of pipe-and-filter architectures. In J. M. Wing, J. Woodcock, and J. Davies, editors, *FM'99, Proceedings of the World Congress on Formal Methods in the Development of Computing System*, pages 96–115. Springer Lect. Notes Comp. Sci. (1708), 1999.

[23] J. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249:3–80, 2000.

[24] B. Sengupta and R. Cleaveland. Triggered Message Sequence Charts. In *ACM SINSOFT 2002, 10th International Symposium on the Foundations of Software Engineering*, pages 167–176, 2002.

[25] R. V. D. Straeten, V. Jonckers, and T. Mens. Supporting model refactoring through behaviour inheritance consistencies. In Thomas Baar et al., editor, *UML 2004 - The Unified Modeling Language*, volume 3273 of *LNCS*, pages 305–319. Springer, 2004.

[26] G. Sunyé, D. Pollet, Y. L. Traon, and J.-M. Jézéquel. Refactoring UML models. In M. Gogolla and C. Kobryn, editors, *Proceedings of UML 2001*, volume 2185 of *LNCS*, pages 134–148. Springer, 2001.

[27] S. Uchitel, J. Kramer, and J. Magee. Synthesis of behavioral models from scenarios. *IEEE Transactions on Software Engineering*, 29(2):99–115, 2003.

[28] S. Uchitel, J. Kramer, and J. Magee. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM Transactions on Software Engineering and Methodology*, 13(1):37–85, 2004.

[29] A. Zawlocki, G. Marczyński, and P. Kosiuczenko. Property preserving redesign of specifications. In J. L. Fiadeiro et al., editor, *CALCO 2005*, volume 3629 of *LNCS*, pages 439–455. Springer, 2005.

[30] T. Ziadi, L. Hélouët, and J.-M. Jézéquel. Revisiting statechart synthesis with an algebraic approach. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*. IEEE Computer Society, 2004.