UML 2 SEMANTICS
AND APPLICATIONS
If, as a well-known aphorism states, modelling is for reasoning, this chapter is an attempt to define and apply a formal semantics to interaction patterns captured by UML 2.0 sequence diagrams in order to enable rigorous 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 chapter argues for the use of coalgebra theory, as the mathematics of state-based designs, to give a precise semantics to sequence diagrams with coercions. In particular, it illustrates how an algebra for constructing and reasoning about them can be made explicit and useful for the working software architect.
1.1 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 rigorous reasoning either to establish models’ properties or to transform models towards effective implementations.

Recalling such an interconnection seems particularly appropriate with respect to the formalization attempts of UML 2.0. 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 formalizations 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.

This chapter introduces a new, coalgebraic semantics for UML 2.0 interaction models represented, as usual, by *sequence diagrams*. The proposed semantics was partially sketched in [25]. Moreover, a set of operators for such diagrams, informally described in [30], is formally characterized, settling the bases for 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 [41, 42, 4]. Those references introduce a semantics for class diagrams, use cases and statecharts based on *coalgebras* [38] taken as a suitable mathematical structure for expressing behaviour of state-based systems. 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 equivalence classes for the bisimilarity relation induced by the particular functor used, upon which properties of UML models can be formulated and checked.

Although the emphasis is placed on the formalisation of sequence diagrams, this chapter also intends to be an almost tutorial introduction to the use of coalgebraic techniques in semantics. Main concepts and tools are introduced in section 1.2, with (the formal description of) a UML class diagram acting as a running example.

Mathematically, coalgebras are the formal duals of algebras, exactly in the sense that makes *observation* and *construction* symmetric notions. Differently from familiar, inductive, data types which are completely defined by a set of constructors, the sort of computational structures that coalgebras can describe, admits only behavioural characterisations, as we will see soon. Typical examples of such structures are *processes*, *transition systems*, *objects*, *stream-like structures* used in lazy programming languages, ‘infinite’ or *non well-founded objects* arising in semantics,
and, as we want to argue along this chapter, *interaction models* for software systems, as seen from the point of view of the working, model-based software architect.

The rest of this chapter is organised as follows. Section 1.2 introduces coalgebras and related notions of behaviour and bisimulation. This is later applied to the construction of a semantic model for sequence diagrams, in section 1.3, and of their combinators, in section 1.4. Section 1.5 illustrates how UML 2.0 diagram annotations (such as typical neg or critical tags) can be incorporated within the model. Finally, section 1.6 shows the semantics ‘in action’, illustrating its use to prove properties of combinators and detailing the corresponding proof techniques. Some pointers for current and future work are collected in section 1.7.

### 1.2 WHY COALGEBRAS?

#### 1.2.1 Classes and Coalgebras

Our starting point is that any useful semantic framework for UML descriptions should be able to address both

- diagram *composition*, defining operators and investigating the laws which govern their behaviour, and
- diagram *refactoring*, in the sense of the original definition of this term given by Opdyke, almost two decades ago, i.e., understood as the *process of changing a software system in such a way that it does not alter the external behaviour of the code, yet improves its internal structure*, to quote [31].

In both cases a precise notion of *behaviour* and a calculational approach to *behavioural equivalence* and *refinement* is the key issue. Actually, such notions are at the kernel of coalgebra theory [15, 38], often suitably called the *mathematics of dynamical systems*. In particular, coalgebra theory provides a standard notion of systems’ behaviour in terms of the bisimilarity relation induced by the signature functor, a technical way to capture a suitable notion of system’s interface. Refinement, as explained below, corresponds to the ability of a diagram to simulate another in a quite precise way.

Experience seems to validate our claim that coalgebra theory may provide an expressive and powerful framework for studying the semantics of UML diagrams, as documented in [41, 42, 25, 4]. Thus, before delving into the details of a semantics for interaction designs, this section should be read as a tutorial introduction to the coalgebraic framework for the working systems architect. Our running example to introduce the main ideas and notation is a fragment of the class diagram depicted in Figure 1.1, corresponding to a simplified model of a video renting e-business.

The model is, certainly, self-explanatory. In any case we will just focus on class *Membership*. The aim of a class declaration is to introduce a signature of attributes and methods. As a representation of object families, a class can actually be regarded as a specification of *state-based* structures, encompassing the following basic elements:
• the presence of an internal state space which evolves and persists in time,

• and the possibility of interaction with other class instances through well-defined interfaces and during the overall computation.

This favors adoption of a behavioural semantics: class instances are inherently dynamic, possess an observable behaviour, but their internal configurations remain hidden and should be identified if not distinguishable by observation. The qualitative ‘state-based’ is, thus, used in the sense the word ‘state’ has in automata theory — the internal memory of the automaton which both constrains and is constrained by the execution of system operations.

Class Membership in Figure 1.1 introduces two attributes and a method over a state space, identified in the sequel by variable $U$, which is made observable exactly (and uniquely) by the attributes and methods it declares. Concretely,

$$\text{joined} : U \rightarrow \text{Date}$$

$$\text{lastHire} : U \rightarrow \text{Date}$$

$$\text{pay} : U \times \mathbb{R} \rightarrow U$$

An alternative, ‘black box’ view hides $U$ from the class environment and regards each operation as a pair of input/output ports. Such a ‘port’ signature of, e.g., the lastHire attribute is given by

$$\text{lastHire} : 1 \rightarrow \text{Date}$$

where 1 stands for the nullary (or unit) datatype, i.e., a representation of the singleton set. The intuition is that lastHire is activated with the simple pushing of a ‘button’ (its argument being the class instance private state space) whose effect is the production of a Date value in the corresponding output port. Similarly typing pay as

$$\text{pay} : \mathbb{R} \rightarrow 1$$
means that an external argument (a float value, modelled here as a real number) is required on activation but no visible output is produced, but for a trivial indication of successful termination. Such ‘port’ signatures are grouped together in the diagram below, in which all occurrences of 1 were dropped.

$$\begin{align*}
\text{joined} & : 1 \rightarrow Date \\
\text{lastHire} & : 1 \rightarrow Date \\
\text{pay} & : \mathbb{R} \rightarrow 1
\end{align*}$$

The diagram represents the class input interface (in the upper part) and its output interface (in the lower part). The behaviour of class Membership instances is given solely in terms of these interface types. Let us detail how and why, making the state space explicit once again.

First, note that the three declarations can be grouped into one through a split construction

$$(\text{joined, lastHire, pay}) : U \rightarrow Date \times Date \times \mathbb{R}$$

where notation $\text{pay}$ denotes the currying of $\text{pay}$. Therefore, we write,

$$[[\text{Membership}]] = (\text{joined, lastHire, pay})$$

I.e., the semantics of each instance of class Membership is given by function (1.1) which describes how it reacts to input stimuli, makes its attributes available and changes state. I.e., as a coalgebra $U \rightarrow T U$ for datatype transformer $T X = Date \times Date \times X\mathbb{R}$, as explained in the sequel.

Actually, the basic insight in coalgebraic modelling is that, for an arbitrary $T$, a state-based system can be represented by a function

$$p : U \rightarrow TU$$

which, for every state $u \in U$, describes the observable effects of an elementary step in the evolution of the system (i.e., a state transition). The possible outcomes of such a step are captured by notation $TU$. Technically, $T$ is a functor. Intuitively, a shape for the allowed observations.

1In order to emphasize the dependency of the possible observations $X$ from the input, we resort to the standard mathematical notation $X^I$ for functional dependency, instead of the equivalent $I \rightarrow X$ more familiar in computing.

2Note that our semantic constructions ‘live’ in a space of typed functions, something one could model as a graph with sets as nodes and set-theoretic functions as arrows. As functions (with the right types) can be composed and, for each set $S$, one may single out a function $\text{id}_S$ (the identity on $S$) acting as the neutral element for composition, this working universe has the structure of a (partial) monoid, i.e., a category. In
Let us consider a few possible alternatives for $T$. An extreme case is the ‘opaque’ shape $T = 1$: no matter what one tries to observe through it, the outcome is always the same. A slightly more interesting case is $T = 2$ which has the ability to classify states into two different classes (say, ‘black’ or ‘white’) and, therefore, to identify subsets of $U$. Should an arbitrary set $O$ be chosen the possible observations become more discriminating. Naturally, the same ‘universe’ can be observed through different attributes and, furthermore, such observations can be carried out in parallel, as in, for example $T = O \times O'$.

The case of a ‘transparent’ $T$, i.e., $T U = U$, is not particularly useful: any function $p : U \rightarrow U$ is a coalgebra for such a functor. But this also means that, by using $p$, the values in the state space $U$ can indeed be modified. On the other hand, the absence of attributes makes any meaningful observation impossible. More interesting, however, are interfaces able to model, e.g., computational partiality ($T U = U + 1$), non determinism ($T U = \mathcal{P}U$), for $\mathcal{P}U$ the finite powerset of $U$, or input triggering ($T U = U I$), among many others. Technically,

**Definition 1.2.1** The pair $\langle U, p : U \rightarrow TU \rangle$ constitutes a coalgebra for functor $T$ over carrier $U$. A morphism connecting two such coalgebras is a function between their carriers making the following diagram to commute:

$$
\begin{array}{ccc}
U & \xrightarrow{p} & TU \\
h \downarrow & & \downarrow \sigma \\
U' & \xrightarrow{p'} & TU' \\
\end{array}
$$

(1.4)

$T$-coalgebras and the corresponding morphisms form a category whose both composition and identities are inherited from $\text{Set}$, the usual category of sets and functions.

Back to our class diagram, note that, in general, the semantics $\llbracket c \rrbracket$ of a class $c$ is given by a specification of a coalgebra

$$
\langle \text{at}, \mathbf{m} \rangle : U \rightarrow A \times U^I
$$

(1.5)

where $A$ is the attribute domain ($\text{Date} \times \text{Date}$ in the example above), and each method accepts a parameter, of type $I$ ($\mathbb{R}$, above), and delivers a state change. I.e., a coalgebra for functor$^3$

$$
T : X \rightarrow A \times X^I
$$

(1.6)

this setting, a *functor* is simply a function $T$ over this universe which preserves the graph and monoidal structure, i.e., for each function $f : A \rightarrow B$, $Tf$ is typed as $TA \rightarrow TB$ and verifies:

$$
T \text{id}_X = \text{id}_T \text{id}_X \quad \text{and} \quad T(f \cdot g) = Tf \cdot Tg
$$

As most conceptual structures used in mathematics and computer science, this notion is borrowed from category theory [22], where it can be appreciated in its full generality.  

$^3$The general case also considers methods producing visible outputs, in which case the relevant functor becomes $T : X \rightarrow A \times (O \times X)^I$, where $O$ denotes the method output type, which was trivially $1$ in the Membership example; see [25] for details.
Typically, I is a sum type, aggregating the input-output parameters of each declared method. On its turn, A is usually a product type joining all attribute outputs in a way which emphasises that each of them is available independently of the others, and therefore always able to be accessed in parallel.

1.2.2 Behaviour and Bisimulation

By now one may ask what is a convenient functor for coalgebraic models of interactions and sequence diagrams and what notion of system behaviour would such a choice enforce. These questions will be discussed in detail in section 1.3. For the moment, however, let us stick to a few variants of an elementary, deterministic, model in order to introduce the basic ideas of coalgebraic modelling applied to software systems.

The simplest model of a class instance one could think of is that of systems inspected by an attribute \( \text{at} : U \rightarrow O \) and reacting (deterministically) to a method (or action) \( m : U \rightarrow U \) with no external influence (but for, say, pushing a button). Those two functions can be ‘glued’ together, as in declaration (1.2), leading to coalgebra

\[
p = \langle \text{at}, m \rangle : U \rightarrow O \times U \quad (1.7)
\]

Successive observations of (or experiments with) system \( p \) reveal its behavioural patterns. For each state value \( u \in U \), the behaviour of \( p \) at \( u \) (more precisely, from \( u \) onwards) — represented by \( [p] u \) — is an infinite sequence of values of type \( O \) computed by observing the successive state configurations, i.e.,

\[
[p] u = \langle \text{at} u, \text{at} (m u), \text{at} (m (m u)), \ldots \rangle \quad (1.8)
\]

Thus, the space of all behaviours, for this sort of systems, is the set of streams (infinite sequences) of \( O \), i.e., \( O^\omega \).

Bringing input information into the scene leads to a mild sophistication of this model. The result is known as a Moore transducer, a classical notion in automata theory [28], where each state is associated to an output symbol. Generalisations of Moore machines play a fundamental role in the semantics of UML diagrams. The semantics of classes, as discussed above, is an example. The semantics of sequence diagrams, as we will see in section 1.3, is another one. Thus, it pays to take them as our running example in the sequel.

Consider, then, an elementary Moore transducer

\[
p = \langle \text{at}, \overline{m} \rangle : U \rightarrow A \times U^I \quad (1.9)
\]

Its dynamics can be decomposed in the following transition relations:

\[
\begin{align*}
  u \xrightarrow{i} p u' & \iff \overline{m} u i = u' \quad (1.10) \\
  u \downarrow_p b & \iff \text{at} u = b \quad (1.11)
\end{align*}
\]
On the other hand, the behaviour of \( p \) at (from) a state \( u \in U \) is revealed by successive observations (experiments) triggered on input of different values \( i \in I \):

\[
\textstyle{\{ p \} u = \langle \text{at} \, u, \text{at} (\text{at} \, u \, i_0), \text{at} (\text{at} \, (\text{at} \, u \, i_0) \, i_1), ... \rangle} \quad (1.12)
\]

\[
\text{or, in a recursive definition,}
\]

\[
\textstyle{\{ p \} u \, \text{nil} = \text{at} \, u} \quad (1.14)
\]

\[
\textstyle{\{ p \} u \, (i : t) = \{ p \} (\text{at} \, u \, i) \, t} \quad (1.15)
\]

Behaviours of Moore transducers organise themselves into tree-like structures, because they depend on the sequences of input processed. Such trees, whose arcs are labelled with \( I \) values and nodes with \( A \) values, can be represented by functions from sequences of input type \( I \) to the attribute type \( A \). In other words, the space of behaviours of Moore machines (on \( I \) and \( A \)) is the set \( A^I \).

Instantiating diagram (1.4) for functor (1.6) defines the corresponding notion of a morphism \( h : \langle \text{at}, \text{m} \rangle \rightarrow \langle \text{at}', \text{m}' \rangle \) as a function connecting their state spaces which satisfies the following equations:

\[
\text{at}' \cdot h = \text{at} \quad (1.16)
\]

\[
\text{m}' \cdot (h \times \text{id}) = h \cdot \text{m} \quad (1.17)
\]

Clearly,

**Lemma 1.2.1** Morphisms preserve attributes and transitions.

*Proof.*

\[
\begin{align*}
\text{at} \rightarrow_{i} u' \text{ and } u \downarrow_{p} a & \quad \Leftrightarrow \quad \{ \text{definition} \} \\
\text{m}(u, i) = u' \text{ and } \text{at} \, u = a & \quad \Leftrightarrow \quad \{ \text{Liebniz} \} \\
h \cdot \text{m}(u, i) = h \, u' \text{ and } \text{at} \, u = a & \quad \Leftrightarrow \quad \{ h \text{ is a morphism: (1.16) and (1.17)} \} \\
\text{m}'(h \, u, i) = h' \, u' \text{ and } \text{at}' \, h \, u = a & \quad \Leftrightarrow \quad \{ \text{definition} \} \\
h \, u \rightarrow_{q} h' \, u' \text{ and } h \, u \downarrow_{q} a
\end{align*}
\]

\[\square\]

Observe, now, that set \( A^I \) of behaviours can itself be equipped with the structure of a Moore transducer as well. Actually, define

\[
\omega = (\text{at}_{\omega}, \text{at}_{\omega}) : A^I \rightarrow A \times (A^I)^I \quad (1.18)
\]
where
\[ \text{at}_\omega f = f \text{ nil} \quad \text{ie, the attribute value before any input is received} \]
\[ \text{m}_\omega fi = \lambda s. f (i : s) \quad \text{ie, every input determines its evolution} \]

Note that a state in \( \omega \) is a function \( f \). Therefore, the attribute is computed by function application, whereas the method gives a new function which reacts to a sequence \( s \) of inputs exactly as \( f \) would react to the appending of \( i \) to \( s \).

Having turned the set of observations \( A^* \) into a coalgebra itself, it is not surprising that every state of a machine \( p \) can be mapped into its behaviours in a ‘well-behaved’ way. In other words,

**Lemma 1.2.2** The behaviour \( \langle p \rangle \) of a coalgebra \( p \) can be singled out as a morphism from \( p \) to \( \omega \).

*Proof.* For \( \langle p \rangle : p \to \omega \), we check the corresponding instances of conditions (1.16) and (1.17)
\[ \text{at}_\omega \cdot \langle p \rangle = \text{at} \quad \text{and} \quad \text{m}_\omega \cdot (\langle p \rangle \times \text{id}) = \langle p \rangle \cdot \text{m} \]  \hfill (1.19)

Thus,
\[
\text{at}_\omega \cdot \langle p \rangle = \text{at} \\
\iff \quad \{ \text{introduction of variables} \} \\
\text{at}_\omega (\langle p \rangle u) = \text{at} u \\
\iff \quad \{ \text{definition of at}_\omega \} \\
(\langle p \rangle u) \text{ nil} = \text{at} u \\
\iff \quad \{ \text{definition of } \langle p \rangle \} \\
\text{true}
\]

and, similarly,
\[
\text{m}_\omega \cdot (\langle p \rangle \times \text{id}) = \langle p \rangle \cdot \text{m} \\
\iff \quad \{ \text{introduction of variables and application} \} \\
\text{m}_\omega (\langle p \rangle u, i) = \langle p \rangle \text{ m} (u, i) \\
\iff \quad \{ \text{definition of m}_\omega \} \\
\text{A s.} \quad (\langle p \rangle u) (i : s) = \langle p \rangle \text{ m} (u, i) \\
\iff \quad \{ \text{introduction of variables and application} \} \\
(\langle p \rangle u) (i : t) = (\langle p \rangle \text{ m} (u, i)) t \\
\iff \quad \{ \text{definition of } \langle p \rangle \} \\
\text{true}
\]

\( \square \)

Note that, a fundamental result on coalgebra morphisms is behaviour *preservation.* Formally, given two coalgebras \( p \) and \( q \) and a morphism \( h : p \to q \) between
them,
\[ \|p\| u = \|q\| h u \]  \hspace{1cm} (1.20)

This leads to a precise and generic notion of behaviour: any two states generate the same behaviour if they can be identified by a coalgebra morphism.

By induction on \( I^* \), it can be proved that there is always a morphism \( \|p\| \) from any \( p \) to \( \omega \) and, as morphisms preserve behaviour, such a morphism is unique. This makes \( \omega \) a very special Moore transducer: it is the only such coalgebra to which, from any other, there is one and only one morphism. We say \( \omega \) is the final Moore machine. Finality is an example of a universal property which, up to isomorphism, provides a complete characterisation of \( \omega \).

Actually, suppose finality is shared by two Moore coalgebras \( \omega \) and \( \omega' \). The existence component of the property gives rise to two morphisms \( h \) and \( h' \) connecting both machines in reverse directions. On the other hand, uniqueness implies \( h \cdot h' = \text{id} \) and \( h' \cdot h = \text{id} \), thus establishing \( h \) and \( h' \) as isomorphisms. These two aspects of finality provide both a definition scheme and a proof principle, generically known as coinduction, upon which coalgebraic reasoning is based. In general,

**Definition 1.2.2** Whenever the space of behaviours of a class of \( T \)-coalgebras can be turned into a \( T \)-coalgebra itself (written as \( \nu_T : \nu_T \longrightarrow T\nu_T \)), this is the final coalgebra: from any other \( T \)-coalgebra \( p \) there is a unique morphism \( \|p\| \) making the following diagram to commute:

\[
\begin{array}{c}
\nu_T \\
\|p\| \\
U \\
\end{array} \hspace{1cm} \begin{array}{c}
\longrightarrow \nu_T \\
\downarrow \uparrow T \nu_T \\
\downarrow \uparrow T \nu_T \\
\langle p \rangle \hspace{1cm} \langle T p \rangle \\
\end{array}
\]

The universal property is, equivalently, captured by the following law:
\[ k = \|p\| \Leftrightarrow \omega_T \cdot k = T k \cdot p \]  \hspace{1cm} (1.21)

Morphism \( \|p\| \) applied to a state value \( u \) gives, of course, the (observable) behaviour of a sequence of \( p \) transitions starting at \( u \). It is called the coinductive extension of \( p \) \([44]\) or the anamorphism generated by \( p \) \([23]\). Coalgebra \( p \) is referred to as its gene. In this context, equation (1.21) is the basic tool for calculating with behaviours. Being an universal property, it asserts, for each gene coalgebra \( p \), the existence and uniqueness of its coinductive extension \( \|p\| \).

As we have already remarked, the existence part of this universal property provides a definition principle for functions to spaces of behaviours (technically, carriers of final coalgebras). This is called definition by co-recursion and boils down to equipping the source of the function to be defined with a coalgebra to capture the ‘one-step’ dynamics in the behaviour generation process. This is exactly the way

\[ \text{Because, roughly speaking, it singles out an entity (} \omega \text{) among a family of ‘similar’ entities to which every other member of the family can be reduced or traced back. The study of universal properties is the ‘essence’ of category theory.} \]
combinators for sequence diagrams will be defined in Section 1.4. Then the corresponding anamorphism gives the rest.

Behavioural equivalence can also be defined in terms of anamorphisms:

**Definition 1.2.3** Two states $u$ and $v$ in the carriers of coalgebras $(U, p)$ and $(V, q)$, respectively, are behaviourally equivalent, represented by $u \sim v$, iff $\|p\| u = \|q\| v$.

Therefore the final coalgebra can be alternatively characterized as a coalgebra whose carrier is composed by all equivalence classes of behavioural equivalence.

Equality (1.20) entails a simpler way of establishing *behavioural equivalence*, which moreover has the advantage of not depending on the existence of final coalgebras: to look for a morphism $h$ such that one of the states is the $h$-image of the other.

Once conjectured, $h$ determines a relation $R \subseteq U \times V$ such that

$$(u, v) \in R \implies u \sim v \quad (1.22)$$

Such a relation is, of course, the *graph* of $h$, i.e., $\{(x, h x) \mid x \in U\}$. Can this idea be generalised? More precisely, what properties must a relation $R$ have so that one can conclude $u \sim v$ simply by checking whether $(u, v)$ is in $R$? Actually such a relation can be characterised and is called a $T$-*bisimulation*. Formally,

**Definition 1.2.4** A $(T)$-bisimulation relating coalgebras $p$ and $q$ is a relation over their carriers which is closed for their dynamics, i.e.

$$(x, y) \in R \implies (p x, q y) \in TR \quad (1.23)$$

Getting rid of variables, (1.23) becomes the following inequality in the language of the (pointfree) calculus of binary relations [2]:

$$R \subseteq p^\circ \cdot (TR) \cdot q \quad (1.24)$$

where $p^\circ$ stands for the relational converse of $p$. Applying the *shunting* rule of the calculus on $p^\circ$, this simplifies to

$$p \cdot R \subseteq (TR) \cdot q \quad (1.25)$$

Informally, two states of a $T$-coalgebra (or of two different $T$-coalgebras) are related by a bisimulation if their observation produces equal results and this is maintained along all possible transitions. I.e., each one can mimic the other’s evolution. Originally the notion was introduced in a functional formulation by [39] and in a relational one by [7]. Park’s landmark paper [32] made bisimulation a basic tool in the context of process calculi. Later [1] gave a categorical definition which applies, not only to the kind of transition systems underlying the operational semantics of process calculi, but also to arbitrary coalgebras. Bisimulation *acquired a shape*: the shape of the chosen observation interface $T$. 

---

**WHY COALGEBRAS?**
1.2.3 Properties and Invariants

Coalgebra theory also provides a way to express and reason about properties of systems. For example, to give semantics to the whole of a UML Class Diagram, like the one depicted in Figure 1.1, both constraints and associations must be taken into account. The former are typically attached to class specifications and their semantic effect is to constraint what coalgebras count as valid implementation for the class. Such is the case, for example, of constraint

\[ \text{balance} > 0 \]

attached to class Membership in our running example. Associations can also be interpreted as constraints, with respect to the sub-diagram formed by the relevant classes, as discussed in [4].

In general, constraints are predicates which are preserved along the system’s lifetime. Formally, they are incorporated in the semantics as invariants. Following the approach recently proposed in [5], such a predicate \( P \), once encoded as a coreflexive relation, i.e., a fragment of the identity, according to

\[ y \mathrel{\Phi_P} x \iff y = x \land P x \]

can be specified as an invariant for a coalgebra \( q \) as follows

\[ q \cdot \Phi_P \subseteq T \Phi_P \cdot q \quad (1.26) \]

When reasoning about diagram transformations constraints entail for proof obligations. For example,

\[ [[\text{balance} > 0]] = [[\text{Membership}]] \cdot \Phi_{\text{balance} > 0} \subseteq T \Phi_{\text{balance} > 0} \cdot [[\text{Membership}]] \]

needs to be discarded whenever justifying a diagram transformation, for example a refactoring, involving class Membership.

1.2.4 Further Reading

Only recently coalgebra theory emerged as a common framework to describe ‘state-based’, dynamical systems. Its study along the lines of Universal Algebra, was initiated by J. Rutten in [36] and [38]. There is a number of tutorials and lecture notes (see, e.g., [15], [10] or [19]) to which the interested reader can be referred to. The proceedings of the Coalgebraic Methods in Computer Science workshop series, initiated in 1998, document current research ranging from the study of concrete coalgebras over different base categories [44, 27] to the development of Set-independent, i.e., purely categorical, presentations of coalgebra theory (see, among others [44, 33, 9]), from coalgebraic logic (e.g., [29, 20]) to applications. Application examples range from automata [37] to objects [34, 13], from process semantics [21, 45, 3] to hybrid transition systems [12]. B. Jacobs and his group, following earlier work by H. Reichel [34, 11] have coined the term coalgebraic specification [14, 16, 35] to denote a style of axiomatic specification involving equations up to bisimilarity acting as constraints on the observable behaviour.
1.3 A SEMANTICS FOR UML SEQUENCE DIAGRAMS

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 a communication discipline (synchrony or asynchrony) and the occurrence of the events associated to message sending and receiving. Events situated on the same lifeline are ordered in time from top to down.

Figure 1.2 An annotated Sequence Diagram

Figure 1.2 is an example of a sequence diagram which describes the interactions in the login phase of an on-line banking scenario. As shown in the example, a UML sequence diagram is represented inside a rectangular frame labeled by the keyword `sd` followed by the interaction name. The vertical lines represent lifelines for the individual participants in the interaction.

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. Formally,

**Definition 1.3.1** A sequence diagram $sd$ is given by a tuple

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

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;
- $Mes$ is a set of message labels;
- $Loc_{ini} \subseteq Loc$ is a set of initial locations;
- $loc : I \rightarrow 2^{Loc}$ associates to each instance a set of locations. The function satisfies the following conditions expressing disjointness and conformity with the initial constraints, respectively,

$$\forall i, j \in I, i \neq j. \text{loc}(i) \cap \text{loc}(j) = \emptyset \quad (1.27)$$

$$\forall i \in I. \text{card}(\text{loc}(i) \cap Loc_{ini}) = 1 \quad (1.28)$$

where $\text{card}(S)$ is the cardinality of $S$.
- $E \subseteq Loc \times Mes \times Loc$ 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 the same, i.e., the following property is assumed:

$$\forall (l_1, m, l_2) \in E. l_1 \neq l_2 \quad (1.29)$$

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 function $\text{next} : Loc \rightarrow Loc$ returns the next location in a particular lifeline,

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

where $<$ stands, as usual, for the total order derived from $\leq$. 
Let $l_1, l_2$ range over $\text{Loc}$, and $\Sigma_m$ be the set of communication events relative to messages exchanged in a sequence diagram $sd$. Each such event has one of the following forms:

1. $\langle l_1 \xrightarrow{-} l_2, m \rangle$ - $l_1$ sends asynchronously message $m$ to $l_2$,
2. $\langle l_1 \xleftarrow{-} l_2, m \rangle$ - $l_1$ receives asynchronously message $m$ from $l_2$,
3. $\langle l_1 \xrightarrow{\circ} l_2, m \rangle$ - $l_1$ sends synchronously message $m$ to $l_2$, and
4. $\langle l_1 \xleftarrow{\circ} l_2, m \rangle$ - $l_1$ receives synchronously message $m$ from $l_2$.

Note that a lost (respectively, a found) message corresponds to an asynchronous sending (respectively, receiving) event to (from) an unknown location. Thus it can be represented by replacing $l_2$ by $\bullet$ in the corresponding clauses above. The type of an arbitrary event $e \in \Sigma_m$ is $\text{type}(e) \in \{/\circ\rightarrow, /\circ\leftarrow, /\square\rightarrow, /\square\leftarrow, /\bullet\rightarrow, /\bullet\leftarrow\}$.

Since the occurrence of events $e = \langle l_1 \xrightarrow{-} l_2, m \rangle$ and $\overline{e} = \langle l_2 \xleftarrow{-} l_1, m \rangle$, denoting, respectively, the sending and receiving of a synchronous message $m$, is simultaneous, we resort to notation $\langle e, \overline{e} \rangle$ to denote their joint occurrence. Finally, let $\Sigma_\tau$ denote the set of local actions in a sequence diagram. Such actions have the form $\langle l \xrightarrow{\bullet} a \rangle$ which means local action $a$ happens at location $l$. We use $\Sigma$ as an abbreviation of $\Sigma_m \cup \Sigma_\tau$.

The set of all event occurrences in a sequence diagram is denoted by $\Sigma$ and defined as

$$\Sigma = (\overline{\Sigma} \setminus \{ e \mid \text{type}(e) = /\rightarrow \lor \text{type}(e) = /\leftarrow \}) \cup \{ \langle e, \overline{e} \rangle \mid \text{type}(e) = /\rightarrow \} \quad (1.30)$$

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

$$\pi\Sigma' = \{ \pi(e) \mid e \in \Sigma' \}$$

and $\pi(\langle e, \overline{e} \rangle) = \{ \pi(e), \pi(\overline{e}) \}$.

A configuration for a sequence diagram denotes a global state, joining together all participants’ local states. For every configuration, there is a set of active events that may happen in that configuration.

**Definition 1.3.2** A configuration $c$ of a sequence diagram is a tuple of the local states (locations) of its participants.

Suppose $C$ denotes the set of all possible configurations, and let function $\varepsilon : C \rightarrow \mathcal{P}(\Sigma)$ return the set of active events on a given configuration. A configuration $c$ is called final if $\varepsilon(c) = \emptyset$. For a configuration $c$ and event $e$, $\pi(e) \in c$ means $\pi(e)$ is a location in $c$.

In this context, the semantics of a sequence diagram $sd$ can be given by a split of two functions over $C$:

$$\langle e, \overline{e} \rangle \quad (1.31)$$
where \( \epsilon \) was defined above and \( \overline{\alpha} : C \rightarrow C^\Sigma \) captures the diagram’s state transition relation triggered by event occurrence. The pair

\[
(C, (\epsilon, \overline{\alpha}))
\]  

(1.32)

together with an initial configuration \( c_0 \), i.e., a tuple of initial locations, one per diagram column, is a pointed coalgebra for functor

\[
\mathcal{T} X = \mathcal{P}(\Sigma) \times X^\Sigma
\]  

(1.33)

which records the set of enabled events as a state attribute.

A fundamental observation is that (1.33) is an instance of functor (1.6). Therefore, the semantics of a sequence diagram can be regarded as yet another instance of a Moore transducer.

One of the advantages of this semantics is to make explicit that a set of enabled events is present in the initial state, i.e., before any interaction occurs. Another advantage is the quite simple form taken by the carrier of the corresponding final coalgebra, as discussed in section 1.2.2:

\[
\nu = \mathcal{P}(\Sigma)^{\Sigma^*}
\]  

(1.34)

i.e., functions which relates each \( \Sigma \)-trace to the set of enabled events upon completion of its execution. As the empty trace \( \langle \rangle \) is a valid trace, one can talk of an initial set of enabled events. To capture the intended semantics for sequence diagrams this is defined as

\[
\epsilon(c_0) = \{ e \mid \pi(e) \in \text{Loc}_{\text{ini}} \land \text{type}(e) \neq \leftrightarrow \}
\]  

(1.35)

Finally, note there is a well-formedness condition on \( \mathcal{T} \)-coalgebras suitable as models of sequence diagrams: in a given configuration \( c \) only events enabled in \( c \) can be triggered. Formally,

\[
\langle \forall e, c : e \in \Sigma, c \in C : e \notin \epsilon(c) \Rightarrow \alpha(c, e) = c \rangle
\]  

(1.36)

Clearly, if at a given \( c \) the set of enabled events becomes empty, i.e., \( \epsilon(c) = \emptyset \), the diagram will remain indefinitely in the same state.

To define functions \( \epsilon \) and \( \alpha \), we will proceed by enumerating all possible transition schemes. First of all, note that, if an event \( e \) is not active in a configuration \( c \), i.e., either \( e \notin \epsilon(c) \) or \( \pi(e) \notin c \), it will not be executed until \( e \) is added to the set of active events as a result of some other event occurrence. This case is captured by a trivial transition

\[
\alpha(c, e) = c
\]

and

\[
\epsilon(\alpha(c, e)) = \epsilon(c)
\]

When a local action \( a \) happens at location \( l \in \text{loc}(i) \), the current location of participant \( i \) is changed to \( \text{next}(l) \). Thus, for \( e = \langle l \cup a \rangle \) where \( l \in c \),

\[
\alpha(c, e) = c[\text{next}(l)/l]
\]
and
\[\epsilon(a(c, e)) = \epsilon(c) \setminus \{e\} \cup \{e' \mid \pi(e') = \text{next}(l) \land \text{type}(e') \neq \leftarrow\}\]

where tuple update notation \(e[l'/l]\) means component \(l\) of tuple \(c\) is replaced by \(l'\).

Events modelling sending and receiving of a synchronous message occur simultaneously (i.e., in an atomic, non interruptible way): no other event can occur in between. So if the current configuration is \(c\) and both the sending event \(e = \langle l_1 \rightarrow \rightarrow l_2, m \rangle\) and the corresponding receiving event \(\overline{e} = \langle l_2 \leftrightarrow l_1, m \rangle\) are active, i.e., \(e \in \epsilon(c), \overline{e} \in \epsilon(c)\), then we have

\[a(c, \langle e, \overline{e} \rangle) = c[\text{next}(l_1)/l_1, \text{next}(l_2)/l_2]\]

and

\[\epsilon(a(c, \langle e, \overline{e} \rangle)) = \epsilon(c) \setminus \{e, \overline{e}\} \cup \{e' \mid \pi(e') = \text{next}(l_1) \land \text{type}(e') \neq \leftarrow\}\]

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 \(e = \langle l_1 \rightarrow \rightarrow l_2, m \rangle\) is active in configuration \(c\), we have

\[a(c, e) = c[\text{next}(l_1)/l_1]\]

and

\[\epsilon(a(c, e)) = \epsilon(c) \setminus \{e\} \cup \{(l_2 \leftrightarrow l_1, m)\} \cup \{e' \mid \pi(e') = \text{next}(l_1) \land \text{type}(e') \neq \leftarrow\}\]

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 \(e = \langle l_1 \leftrightarrow l_2, m \rangle\) is active in configuration \(c\), we have

\[a(c, e) = c[\text{next}(l_1)/l_1]\]

and

\[\epsilon(a(c, e)) = \epsilon(c) \setminus \{e\} \cup \{e' \mid \pi(e') = \text{next}(l_1) \land \text{type}(e') \neq \leftarrow\}\]

The case of a lost message, represented by event \(e = \langle l \leftarrow, m \rangle\), 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 \(e = \langle l \rightarrow, m \rangle\) 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

\[a(c, e) = c[\text{next}(l)/l]\]

and

\[\epsilon(a(c, e)) = \epsilon(c) \setminus \{e\} \cup \{e' \mid \pi(e') = \text{next}(l) \land \text{type}(e') \neq \leftarrow\}\]

assuming the corresponding events are enabled in configuration \(c\).
1.4 NEW SEQUENCE DIAGRAMS FROM OLD

In the previous section the semantics of an arbitrary sequence diagram $sd$ was defined by a pointed coalgebra

$$\sem{sd} = (C, \langle \epsilon, \alpha \rangle : C \to \powerset(\Sigma) \times C^\Sigma, c_0)$$

over the set $C$ of $sd$ configurations.

UML 2.0 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 [30] proposed one such set and gave an informal characterisation of the associated behaviours as follows:

- The operator $\text{alt}$ offers a choice of behavior alternatives represented by its two operands. The chosen sequence diagram must have an explicit or implicit guard expression that evaluates to true at this point in the interaction.

- The operator $\text{opt}$ designates a choice between the its (sole) operand or a idle behaviour.

- The operator $\text{par}$ stands for the parallel merge of the behaviors of the sequence diagram acting as its operands. Event occurrences in the different operands can be interleaved in any way as long as the ordering imposed inside each of them is preserved.

- The operator $\text{seq}$ represents a weak sequencing of behaviours of its 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 $\text{strict}$ represents a strict sequencing of the behaviors: all events in the first operand are made to occur before any event in the second.

- The operator $\text{loop}$ operator specifies an iteration of sequential composition: the execution of its operand repeats itself on completion.

The denotation of these operators in the envisaged semantic model formalises an algebra for building new sequence diagrams from old.

In the sequel, the semantics of $sd_i = (I_i, Loc_i, Mes_i, Loc^i_{ins}, loc_i, E_i, \leq_i)$ will be denoted by

$$\sem{sd_i} = (C_i, \langle \epsilon_i, \alpha_i \rangle : C_i \to \powerset(\Sigma_i) \times C_i^\Sigma_i, c_i^0)$$

where for any $c \in C_i$, i.e. the tuple of local states of participants in $sd_i$, $\Sigma_i^c = \epsilon_i(c)$ returns the set of events that are active in $c$. Moreover, we let $\epsilon_i(c^0_i) = \Sigma^0_i$. For a tuple
of elements \( t = (e_1, e_2, \ldots, e_m) \), we resort to projection function \( \pi_i \), for \( i = 1, \ldots, m \), to return its \( i \)-th element \( e_i \). With such notational conventions we are prepared to give the semantics of operators for combining interaction fragments.

**Choice:** \( \text{alt}(sd_1, sd_2) \).

Composition by \( \text{alt} \) requires \( c_0^1 = c_0^2 \) and that all events in both \( \Sigma_0^1 \) and \( \Sigma_0^2 \) become active in the initial configuration \( c_0 \). Therefore, \( c_0 = c_0^1 \) and \( C = \{c_0\} \cup (C_1 \setminus \{c_0^1\}) \cup (C_2 \setminus \{c_0^2\}) \). Formally\(^5\),

\[
\llbracket \text{alt}(sd_1, sd_2) \rrbracket = (C, (\text{alt}(e_1, e_2), \overline{\text{alt}(\alpha_1, \alpha_2)}), c_0)
\]

with

\[
\text{alt}(\alpha_1, \alpha_2)(x, e) = \begin{cases} 
\alpha_i(c_0, e) & \text{for } i = 1, 2 \\
\alpha_1(x, e) & x \in C_1 \setminus \{c_0^1\} \land e \in \Sigma_i \\
\alpha_2(x, e) & x \in C_2 \setminus \{c_0^2\} \land e \in \Sigma_i \\
\text{otherwise} & x
\end{cases}
\]

where \( x \) is a configuration in \( C \), and \( e \) is an event in either \( \Sigma_1 \) or \( \Sigma_2 \).

**Option:** \( \text{opt}(sd_l) \).

The purpose of \( \text{opt}(sd_l) \) is to offer an alternative between an empty scenario (in which ‘nothing happens’) and the activation of its (sole) operand, \( sd_l \). Its formalisation requires the introduction of a new event — \( \text{skip} \) — into the set of events to capture absence of effective behaviour. Then

\[
\llbracket \text{opt}(sd_l) \rrbracket = (C, (\overline{\text{opt}(e_1)}), c_0)
\]

where \( C = C_1 \) and \( c_0 = c_0^1 \). The transition structure is defined as

\[
\text{opt}(\alpha_1)(x, e) = \begin{cases} 
\alpha_1(x, e) & x \neq c_0 \land e \in \Sigma_1 \\
\text{let } x' = c_0^1 \text{ in } e x' = \emptyset \Rightarrow x' & x = c_0 \land e = \text{skip} \\
\alpha_1(c_0, e) & x = c_0 \land e \in \Sigma_1 \\
\text{otherwise} & x = c_0
\end{cases}
\]

\(^5\)To avoid an excessive notational burden, we use the same syntax for the combinator over sequence diagrams and its denotation in the proposed semantics.
Parallel: $\text{par}(sd_1, sd_2)$.
As one would expect, the state space for parallel composition is a Cartesian product, i.e., $C = C_1 \times C_2$, with $c_0 = (c_0^1, c_0^2)$. Then,

$$ \llbracket \text{par}(sd_1, sd_2) \rrbracket = (C, (\text{par}(e_1, e_2), \text{par}(\alpha_1, \alpha_2)), c_0) $$

where the transition structure is specified by

$$ \text{par}(e_1, e_2)(x) = e_1(\pi_1 x) \cup e_2(\pi_2 x) $$

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

Strict sequential composition: $\text{strict}(sd_1, sd_2)$.
For sequential composition, the transition structure in

$$ \llbracket \text{strict}(sd_1, sd_2) \rrbracket = (C, (\text{strict}(e_1, e_2), \text{strict}(\alpha_1, \alpha_2)), c_0) $$

is defined over $C = C_1 \cup C_2 \setminus \{c \mid c \in C_1 \wedge e_1(c) = \emptyset\}$ and $c_0 = c_0^1$ as follows

$$ \text{strict}(e_1, e_2)(x) = \begin{cases} x \in C_1 \wedge e_1(x) \neq \emptyset & \Rightarrow e_1(x) \\
\text{otherwise} & x \in C_2 \Rightarrow e_2(x) 
\end{cases} $$

$$ \text{strict}(\alpha_1, \alpha_2)(x, e) = \begin{cases} x \in C_1 & \text{let } x' = \alpha_1(x, e) \text{ in } e_1 x' = \emptyset \Rightarrow c_0^2 \\
\text{otherwise} & x' \alpha_2(x, e) 
\end{cases} $$

Weak sequential composition: $\text{seq}(sd_1, sd_2)$.
The case for weak sequential composition $\text{seq}(sd_1, sd_2)$ for $sd_i, i = 1, 2$ 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, weak sequencing reduces to a parallel merge.

Assume an identifier $s$, such that $I_1 \cap I_2 = \{s\}$, and functions $\text{loc}_1$ and $\text{loc}_2$ assigning locations to instances in $sd_1$ and $sd_2$, respectively. Let $\text{loc}(s) = \text{loc}_1(s) \cup \text{loc}_2(s)$. Furthermore, and without loss of generality, let $C_1 = \text{loc}_1(s) \times L$ and $C_2 = \text{loc}_2(s) \times K$ be the set of configurations for $sd_1$ and $sd_2$ respectively, where $L = \prod_{i \in I_1} \text{loc}_1(i)$ and $K = \prod_{j \in I_2} \text{loc}_2(j)$. Then, define

$$ \llbracket \text{seq}(sd_1, sd_2) \rrbracket = (C, (\text{seq}(e_1, e_2), \text{seq}(\alpha_1, \alpha_2)), c_0) $$

with $C = \text{loc}(s) \times L \times K$ and $c_0 = (x_1, x_2)_{x_1, x_2}^0$. We use $e$ as an abbreviation for $\text{seq}(e_1, e_2)$, and for any $c \in C$,

$$ e(c) = \bigcup_{(a_i, \pi_i) \in C, i = 1, 2} e((a_i, \pi_i), c) \setminus \{e | \pi(e) \in \text{loc}(s) \wedge \pi(e) \neq \pi_1 c\} $$
where $a_i \in \text{loc}(s)$, $i = 1, 2$ are two locations such that 

\[
(a_1, \pi_2 c) \in C_1 \land (a_2, \pi_3 c) \in C_2 \land (\pi_1 c = a_1 \lor \pi_1 c = a_2)
\]

The transition structure is given by

\[
\text{seq}(a_1, a_2)(x, e) = \begin{cases}
\pi(e) \cap \text{loc}(s) \neq \emptyset \Rightarrow \\
\pi(e) \cap \text{loc}(s) = \emptyset \Rightarrow \\
\text{let } x' = a_1((\pi_1, \pi_2)x, e), c_0^0 = (l, l) \text{ in } \\
\{ \\
\forall l' \in \text{loc}(s), l' \leq \pi(e) \cap \text{loc}(s) \Rightarrow (l, \pi_2 x', \pi_3 x) \\
\text{otherwise } (\pi_1 x', \pi_2 x', \pi_3 x) \\
\} \\
\text{let } x' = a_2((\pi_1, \pi_3)x, e) \text{ in } (\pi_1 x', \pi_2 x, \pi_2 x') \\
\pi(e) \cap \text{loc}(s) = \emptyset \Rightarrow \\
\{ \\
e \in \Sigma_1 \Rightarrow \\
\text{let } x' = a_1((\pi_1, \pi_2)x, e) \text{ in } (\pi_1 x', \pi_2 x', \pi_3 x) \\
e \in \Sigma_2 \Rightarrow \\
\text{let } x' = a_2((\pi_1, \pi_3)x, e) \text{ in } (\pi_1 x', \pi_2 x, \pi_2 x') \\
\} \\
\end{cases}
\]

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

\[
\text{seq}(sd_1, sd_2) = \text{par}(sd_1, sd_2) \tag{1.38}
\]

Furthermore, whenever $I_1 = I_2$, we have

\[
\text{seq}(sd_1, sd_2) = \text{strict}(sd_1, sd_2) \tag{1.39}
\]

The above equalities are in fact bisimulation equations between the corresponding denotations, i.e., for example, for equation (1.38),

\[
\llbracket \text{seq}(sd_1, sd_2) \rrbracket \sim \llbracket \text{par}(sd_1, sd_2) \rrbracket
\]

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 1.6.

\textbf{Loop:} $\text{loop}(sd_1)$.

Finally, the semantics of the iteration combinator is given by

\[
\llbracket \text{loop}(sd_1) \rrbracket = (C, (\text{loop}(e_1), \text{loop}(a_1)), c_0)
\]
over \( C = C_1 \) and \( c_0 = c_1 \), and with the following transition structure

\[
\begin{align*}
\text{loop}(\epsilon_1)(x) &= \epsilon_1(x) \\
\text{loop}(\alpha_1)(x, e) &= \begin{cases} 
 e \in \text{loop}(\epsilon_1)(x) 
 & \Rightarrow \text{let } x' = \alpha_1(x, e) \text{ in } \\
 & \text{loop}(\epsilon_1)(x') = \emptyset \Rightarrow c_0 \\
 \text{otherwise} 
 & \text{otherwise } x'
\end{cases}
\end{align*}
\]

\[1.5 \text{ COERCIONS & DESIGNS}\]

The description of a sequence diagram in UML 2.0 can also be annotated with some sort of coercions which restricts or expands the underlying possible behaviours. In the UML tradition, such conditions are themselves specified as sequence diagrams (instead of, say, through formulae in a logic). An example is depicted in Figure 1.3. Note that, although annotations like, e.g., \textit{critical} or \textit{alt} are syntactically similar, their intended semantics is completely different: the former stands for a behavioural restriction in the diagram, the latter for a composition operator. Annotated diagrams will be called designs in the sequel and represented as \( \langle \text{coer}(p) \rangle \sd \).

This section illustrates how such coercions can be accommodated in our semantic framework. To be concise, we shall consider only the following, most common, cases of possible coercions on a diagram \( sd \):

- **Annotation \textit{neg}, parametric on a sequence diagram \( p \), which restricts the behaviour of \( sd \) to exclude all interactions specified by \( p \).** It is required that the set of events \( \Sigma_p \) of \( p \) is a subset of the corresponding set \( \Sigma \) in \( sd \). In the example of Figure 1.3 the \textit{neg} annotation rules out the possibility of a confirmation message immediately followed by the production of a receipt.

- **Annotation \textit{critical}, parametric on a sequence diagram \( p \), which requires that all interactions specified by \( p \) occur without interruption or interference of other events in \( sd \).** Such is the case of pin validation in the example of Figure 1.3.

- **Annotation \textit{ignore}, parametric on a message \( m \), which abstracts away the behaviour of \( sd \) of any occurrence of \( m \).**

We have already claimed that an advantage of adopting a coalgebraic framework for diagram semantics is that, once fixed the functor, a canonical characterisation of behaviour pops out as the carrier of the final coalgebra. As discussed above, for \( T \) given by (1.33), behaviours are elements of \( \mathcal{P}(\Sigma)^* \). Thus, we may define the semantics of an annotated diagram \( sd \) as a pair

\[
\langle \llbracket sd \rrbracket, \beta(sd) \rangle \quad (1.40)
\]
where \( \llbracket sd \rrbracket \) denotes the semantics of \( sd \), as in (1.41) and ignoring any annotation, and \( \beta(sd) \) is the behaviour of the annotated diagram. The latter is, typically, but not always, a restriction of the behaviour of \( \llbracket sd \rrbracket \). This, on the other hand, is, as you may recall, canonically given as the coinductive extension of coalgebra \( \llbracket sd \rrbracket \) applied to its initial configuration \( sd_0 \), i.e., \( \llbracket \llbracket sd \rrbracket \rrbracket(sd_0) \). For the sake of uniformity, we can also present the semantics of a non annotated diagram as a pair

\[
\langle \llbracket \llbracket \llbracket sd \rrbracket \rrbracket \rrbracket(sd_0) \rangle
\]

We shall now define the semantics of the three sorts of designs discussed in this paper. In all cases consider annotations over a diagram \( sd \) with \( sd_0 \) as the initial configuration.

Also note that, in the final model \( \nu \), we can rule out all sequences of events which lead to empty sets of observations (i.e., of enabled events). This entails the definition of the following function to compute the (allowed) traces of a sequence diagram \( sd \) with initial configuration \( sd_0 \):

\[
\text{traces} = \{ t \in \Sigma^* | \llbracket \llbracket sd \rrbracket \rrbracket((sd_0)(t)) \neq \emptyset \}
\]
Thus,

design: \( (\text{neg}(p)) \) sd

The intuition behind the definition below is that the set of enabled events after completion of a particular trace \( t \in \Sigma^* \) in the annotated diagram is purged of all events enabled by completion of \( t \) in (the coinductive extension of) \( p \). Eventually this can reduce to zero the set of enabled events after a particular trace, which, as discussed above, corresponds to rule out such a trace as a possible interaction in the design. Or it may simply eliminate a few elements of the event set, meaning that the completion of \( t \) still leads to a non deadlock state, if only a subset of interactions abstracted in \( t \) are represented in \( p \).

Formally,

\[
\llbracket (\text{neg}(p)) \rrbracket_{sd} = \langle \llbracket sd \rrbracket, \llbracket sd \rrbracket_{sd0} \triangleright \llbracket p \rrbracket_{p0} \rangle \tag{1.43}
\]

where, for \( f, g : \Sigma^* \rightarrow \mathcal{P}(\Sigma) \),

\[
(f \triangleright g) t = f(t) - g(t) \tag{1.44}
\]

Notice that, as behaviours are total functions, all possible interactions of \( p \), which correspond to traces on the domain of its behaviour, are taken in consideration.

design: \( (\text{critical}(p)) \) sd

\[
\llbracket (\text{critical}(p)) \rrbracket_{sd} = \langle \llbracket sd \rrbracket, \beta \rangle \tag{1.45}
\]

where

\[
\beta(t) = \begin{cases} 
\llbracket p \rrbracket_{p0}(t) \quad \Leftrightarrow t \in \text{traces}(p) \\
\llbracket sd \rrbracket_{sd0}(t) \quad \Leftrightarrow \text{otherwise} 
\end{cases} \tag{1.46}
\]

Notice that as all traces in \( p \) are taken into consideration, and that the prefix of a trace is also a trace, the dynamics of \( p \) will always overrides that of original sd, whenever the later involves events in the former. For example, suppose \( sd \) allows trace \( (a, z, b) \), but \( p \) has \( (a, b) \) as the unique trace starting with event \( a \), thus forcing it to occur with no interruption in the annotated diagram. Clearly, event \( z \) is enabled in \( sd \) after \( a \), i.e., \( z \in \llbracket sd \rrbracket_{sd0}(a) \) but such is not the case in \( p \), where \( \llbracket p \rrbracket_{p0}(a) = \{b\} \). Therefore, traces \( (a, z) \) and \( (a, z, b) \) will not be allowed in the semantics of the annotated diagram.

design: \( (\text{ignore}(m)) \) sd

In this design, the annotation is parametric on a single message \( m \in \Sigma_{sd} \) which is supposed to be ignored in any interaction specified by \( sd \). Thus the behaviour of this design is defined over \( \Sigma_{sd} - \{m\} \): for each original trace \( t \) ending in \( m \), the enabled events of \( t \) and of its maximal prefix are joined together. Formally,

\[
\llbracket (\text{ignore}(m)) \rrbracket_{sd} = \langle \llbracket sd \rrbracket, \gamma \rangle \tag{1.47}
\]
where

\[ \gamma(t) = \parallel p \parallel_1(p_0)(t) \cup \parallel p \parallel_1(p_0)(t^\sim(m)) \] (1.48)

The following result shows that compositional reasoning is still possible when dealing with annotated diagrams:

**Lemma 1.5.1** Annotations always sum up. i.e., the design resulting from composing with \( \theta \) two other designs corresponds to the composition of the underlying diagrams with \( \theta \) to which both coercions are added afterwards. Formally,

\[ ((\text{coer}_1(p)) \text{sd}_1) \theta ((\text{coer}_2(q)) \text{sd}_2) = (\text{coer}_1(p)) (\text{coer}_2(q)) (\text{sd}_1 \theta \text{sd}_2) \]

for coer ranging over neg, critical and ignore.

**Proof.** Let \( \theta \) be any of the sequence diagrams operators characterised in section 1.4. As annotations in a sequence diagram only affect disjoint sub-diagrams, it is trivial to check, from the semantics of neg, critical and ignore that restrictions act over disjoint sets of events. Thus, their effects manifest themselves cumulatively.

\[ \square \]

### 1.6 A CALCULUS FOR INTERACTIONS

#### 1.6.1 Towards a calculus of diagram composition

Equations (1.38) and (1.39) above were our first examples of properties which establish, under suitable conditions, behavioural equality 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.

For functor \( T \) given by (1.33), bisimulation definition (1.2.4) boils down to

\[ (c, d) \in R \Rightarrow \forall \epsilon \in \Sigma . \epsilon(c) = \epsilon(d) \land (\phi(c, e), \varphi(d, e)) \in R \] (1.49)

for every pair of configurations \((c, d)\), where \( p = (\epsilon, \phi) \) and \( q = (\epsilon, \varphi) \). This provides a 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,

**Lemma 1.6.1** Operators \( \text{alt}, \text{par} \) and \( \text{strict} \) are associative. Formally,

\[ \text{tensor}(\text{tensor}(\text{sd}_1, \text{sd}_2), \text{sd}_3) = \text{tensor}(\text{sd}_1, \text{tensor}(\text{sd}_2, \text{sd}_3)) \] (1.50)

for \( \text{tensor} = \text{alt}, \text{par}, \text{strict} \).

**Proof.** Let us consider the case for \( \text{alt} \), i.e.,

\[ [\text{alt}(\text{alt}(\text{sd}_1, \text{sd}_2), \text{sd}_3)] \sim [\text{alt}(\text{sd}_1, \text{alt}(\text{sd}_2, \text{sd}_3))] \]
The set of configurations for both sides of this equation is \( C = \{c_0\} \cup \bigcup_{i \geq 2} (C_i \setminus \{c_i'\}) \), and the initial configuration, also in both cases, is \( c_0 \). Let \( \varepsilon_i(c_i') = \sum_i \) for \( i = 1, 2, 3 \). For any \( x \in C \) and event \( e \) one gets, according to the definition,
\[
\text{alt}(\text{alt}(\varepsilon_1, \varepsilon_2), \varepsilon_3)(x) = \begin{cases} x = c_0 & \Rightarrow e_i(x) \\ x \in C_1 \setminus \{c_1'\} & \Rightarrow e_i(x) \\ x \in C_2 \setminus \{c_2'\} & \Rightarrow e_i(x) \\ x \in C_3 \setminus \{c_3'\} & \Rightarrow e_i(x) \end{cases}
\]

Similarly,

**Lemma 1.6.2** Operators \( \text{alt} \) and \( \text{par} \) are commutative. Formally,
\[
\text{tensor}(sd_1, sd_2) = \text{tensor}(sd_2, sd_1)
\]

for \( \text{tensor} = \text{alt}, \text{par} \).

**Proof.** Again our task is to verify the bisimulation equation
\[
\llbracket \text{par}(sd_1, sd_2) \rrbracket \sim \llbracket \text{par}(sd_2, sd_1) \rrbracket
\]

The sets of configurations for \( \llbracket \text{par}(sd_1, sd_2) \rrbracket \) and \( \llbracket \text{par}(sd_2, sd_1) \rrbracket \) are \( C = C_1 \times C_3 \) and \( D = C_2 \times C_1 \), respectively. Define \( h : C \rightarrow D \) as \( h = (\pi_2, \pi_1) \). To prove the bisimulation equation, we only need to show that \( h \) is a coalgebra morphism, i.e., to prove that the equations
\[
\text{par}(\varepsilon_1, \varepsilon_2)(x) = \text{par}(\varepsilon_2, \varepsilon_1)(h(x))
\]
\[
h \cdot \text{par}(\alpha_1, \alpha_2)(x, e) = \text{par}(\alpha_2, \alpha_1)(h(x), e)
\]

are satisfied for any configuration \( x \) and event \( e \). According to the definition of \( \text{par} \),
\[
\text{par}(\varepsilon_1, \varepsilon_2)(h(x)) = \text{par}(\varepsilon_2, \varepsilon_1)((\pi_2, \pi_1) x)
\]
\[
= \varepsilon_2(\pi_1 \cdot (\pi_2, \pi_1) x) \cup \varepsilon_1(\pi_2 \cdot (\pi_2, \pi_1) x)
\]
\[
= e_2(\pi_2 x) \cup e_1(\pi_1 x)
\]
\[
= e_1(\pi_1 x) \cup e_2(\pi_2 x)
\]
\[
= \text{par}(\varepsilon_1, \varepsilon_2)(x)
\]
and for $e \in \Sigma_1$,

$$\par(\alpha_2, \alpha_1)(h(x), e)$$

$$= \par(\alpha_2, \alpha_1)((\pi_2, \pi_1)(x), e)$$

$$= \let x' = \alpha_1(\pi_2 \cdot (\pi_2, \pi_1)x, e) \in (\pi_1 \cdot (\pi_2, \pi_1)x, x')$$

$$= (\pi_2 x, \alpha_1(\pi_1 x, e))$$

$$= (\pi_2, \pi_1) \cdot (\alpha_1(\pi_1 x, e), \pi_2 x)$$

$$= h \cdot \par(\alpha_1, \alpha_2)(x, e)$$

Similarly, for $e \in \Sigma_2$, we also get $\par(\alpha_2, \alpha_1)(h(x), e) = h \cdot \par(\alpha_1, \alpha_2)(x, e)$. And for $e \notin \Sigma_1 \cup \Sigma_2$, the result is obvious: $h(x) = h(x)$. The proof is complete noting that $h(c_1^3) = c_2^3$.

This second proof resorted to a quite handy technique of coinductive reasoning: to establish bisimilarity it is enough to define a coalgebra morphism connecting the two coalgebras. Such a technique is based on the fact that coalgebra morphisms entail bisimulation, a direct consequence of (1.20).

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,

$$\alt(sd, sd) = sd$$

$$\alt(sd, \emptyset_{I_{sd}}) = \opt(sd)$$

$$\strict(\alt(sd_1, sd_2), sd_3) = \alt(\strict(sd_1, sd_3), \strict(sd_2, sd_3))$$

$$\strict(sd_1, \alt(sd_2, sd_3)) = \alt(\strict(sd_1, sd_2), \strict(sd_1, sd_3))$$

$$\par(\alt(sd_1, sd_2), sd_3) = \alt(\par(sd_1, sd_3), \par(sd_2, sd_3))$$

$$\par(sd_1, \alt(sd_2, sd_3)) = \alt(\par(sd_1, sd_2), \par(sd_1, sd_3))$$

In equation (1.53) we use $\emptyset_{I_{sd}}$ to denote the empty sequence diagram with the same set of participants as $sd$, but no events (i.e., $\emptyset_{I_{sd}} = (I, Loc_{ini}, \emptyset, Loc_{ini}, i \mapsto \emptyset_0, \emptyset, =)$, for $sd = (I, Loc, Mes, Loc_{ini}, loc, E, \leq))$.

### 1.6.2 Refactoring

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

Originally introduced by Opdyke in [31] in the context of OO programming, refactoring has been widely used in modern software development processes such as Rational Unified Process [18] and eXtreme Programming [6] to support iterative software development and improve the quality of software artifacts. More recently, expression ‘model refactoring’ was coined to witness the shift of emphasis from code to the design level [40, 43, 46].
In any case, 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,

**Lemma 1.6.3** A new lifeline can be introduced into a sequence diagram.

*Proof.* Suppose \( sd = (I, Loc, Mes, Loc_{ini}, loc, \leq) \) 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 \( l_0 \). So the resulting diagram is \( sd' = (I \cup \{i\}, Loc \cup \{l_0\}, Mes, Loc_{ini} \cup \{l_0\}, loc \cup \{i \mapsto l_0\}, \leq) \). If \( c \) is a configuration for \( sd \), then \( (c, l_0) \) is a configuration for \( sd' \). Let \( h = \pi_1 \times \text{id} \). This morphism maps every configuration of \( sd' \) to a configuration of \( sd \), and forms a coalgebra morphism between them, which justifies the law.

\[ \square \]

A similar argument justifies the dual law for removing lifelines:

**Lemma 1.6.4** 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, require preservation of behaviour in a weaker sense. 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 [26, 24]. In brief, the idea is to relax the coalgebra morphism condition in (1.4) by

\[ T \cdot h \cdot p \leq p' \cdot h \]  

(1.58)

where \( \leq \) is a so-called refinement preorder [24]. Function \( h \) is said to be a forward morphism: it preserves transitions from coalgebra \( p \) but fails to reflect \( p' \) transitions back to \( p \). Relation \( \leq \), for functor \( T \) given in (1.33), is a preorder on functions from events to configurations. A possible example requires one of the diagrams to possess less active events than the other in related configurations, as captured by the following (in)equations:

\[ e_1(c_1) \subseteq e_2(h(c_1)) \]

\[ h \cdot \alpha_1(c_1, e) = \alpha_2(h(c_1), e) \]

The existence of a forward morphism \( \leq \) connecting two (coalgebraic denotations of) sequence diagrams witnesses a refinement situation captures by preorder \( \ll \). With respect to this preorder, designs discussed in section 1.5 can be related to the original diagram by forward refinement. In particular,
Lemma 1.6.5

\[
\langle \text{neg}(p) \rangle \, sd \preceq sd \\
\langle \text{critical}(p) \rangle \, sd \preceq sd \\
sd \preceq \langle \text{ignore}(m) \rangle \, sd
\]

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 several refactoring situations for sequence diagrams, as refinement results.

1.7 CONCLUDING

This chapter introduced a coalgebraic semantic framework for UML 2.0 sequence diagrams, including the formalisation of the recently proposed set of combinators for such diagrams. It was also illustrated how coinductive techniques can be used to prove properties of UML designs and develop a theory of sequence diagrams composition and refactoring. This piece of research is in line with the authors’ previous work on coalgebraic semantics for other UML models [41, 42, 4] and can be regarded as part of a major attempt to give a precise semantics to UML descriptions. Several proposals for formalising UML 2.0 sequence diagrams are known. Among them, we are particularly interested in approaches, such as [17], which are also based on translating sequence diagrams to a language of automata, therefore entailing, even if implicitly, a coalgebraic perspective. A proper comparison with the approach proposed in this chapter is in order.

For future work, we single out, as a main open issue, the need for a detailed classification of possible refactoring patterns and their formalization in this framework. Refactoring by (coalgebraic) refinement, as pointed out in section 1.6, is also an open field for further research.

Acknowledgements.

This work is partially funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-007254.

REFERENCES


REFERENCES


A COALGEBRAIC SEMANTIC FRAMEWORK FOR REASONING ABOUT INTERACTION DESIGNS


