A formal model for integrating multiple views

J. K. F. Bowles and B. Bordbar
School of Computer Science, The University of Birmingham
Edgbaston, Birmingham B15 2TT, UK
{J.Bowles| B.Bordbar}@cs.bham.ac.uk
Phone:+44-121-4148556
Fax:+44-121-4144281

Abstract

In this paper we show how to use labelled event structures as a unique mathematical representation for design models consisting of different UML 2.0 diagrams/notation. Each diagram is used to capture a particular aspect or view *of the system including structural and behavioural aspects. Our approach enables the analysis of complex systems designed in a combination of UML 2.0 notation, and serves as a means to detect inconsistencies in design.*

1 Introduction

Designing a system often starts by identifying a correct architecture for the system. This can be captured by a structural model which specifies components involved in the system and their interconnections. Structural models can be further enhanced by incorporating behavioural models. A behavioural model specifies the interaction between system components to fulfil various tasks of the system. The design and specification of structural and behavioral aspects of systems have received considerable attention, for example [14, 11, 18]. The inclusion of constraints to impose suitable restrictions and to ensure correct functioning of the system further refines structural and behavioural specifications. For example, constraints can be used to ensure correct synchronisation of interacting components [3], and to specify non-functional aspects such as Quality of Service [4], fault tolerance and policies [19]. As a result, models representing structure, behaviour and constraints are closer to better capture essential aspects of systems. However, for formal analysis and verification and thus to establish a clear understanding of the system, it is crucial to create a formal represention that embodies all views of the system.

In this paper, we use UML 2.0 to design systems consisting of a combination of class diagrams (structure), sequence diagrams (behaviour) and OCL (further constraints). In our approach, a design model in multiple views is translated into a unique mathematical model (labelled event structures) which can not only be used for analysis but also serves as a means to detect inconsistencies in design (for example, between an OCL constraint and a sequence diagram). We also expect our approach to allow us to identify missing behaviour (not all possible scenarios of behaviour have been identified in design), and indicate possible model optimisations (for example, a given OCL constraint can make a fragment within a sequence diagram redundant).

This paper is structured as follows. In Section 2 we show how to model systems using different UML 2.0 notation and introduce an example. Section 3 describes the mathematical model and the categorical construction used for combining view models. It also describes how the same model can be used for the two different views (structural and behavioural). In section 4 we define a *view synchronisation diagram* which allows us to use the categorical construction explained earlier to obtain the desired model. We illustrate how the diagram can also be used to identify inconsistencies or, as in our example, discard a sequence diagram that does not take into account the constraints on the structural model. We finish the paper with a discussion on related work and ideas for further research.

2 Modelling with UML 2.0: An Example

This section describes how we model systems using UML 2.0 [17]. We use two different UML diagrams to describe structural and behavioural aspects of a system. These different aspects which we capture through class diagrams (structure) and sequence diagrams (behaviour) can also be seen as giving different *views* on the system. In addition, we use the constraint language OCL [16] to capture further constraints on the model which cannot be captured diagrammatically. We illustrate the notation used by means of a simple example. We will come back to this example in later sections to explain our mathematical framework.

Consider the system described in Figure 1 of a sliding machine. The machine pushes incoming jobs into places A and B. In order to do so it uses two sliders. We want

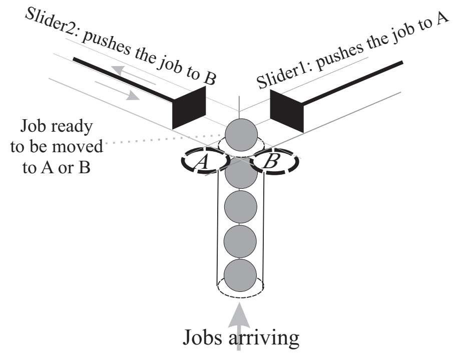

Slider2: pushes the job to B
Slider1: pushes the job to A
Job ready to be moved to A or B
A
B
Jobs arriving

Figure 1. A sliding machine.

to model the system in such a way that we guarantee the correct behaviour of the system. In particular, we want to prevent both sliders from attempting to push the same job into their respective places.

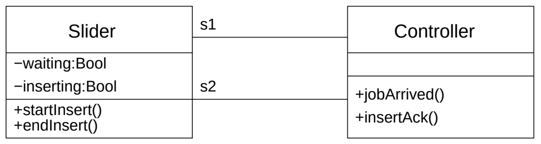


Figure 2. Class diagram for sliding machine.

Figure 2 shows the classes and their relationships for the example. The attributes of class `Slider` are used to indicate the state of a given slider (`inserting` or `waiting`). We assume the existence of a controller that knows when jobs arrive and allocates each job to a slider. The following OCL constraints restrict the possible behaviour of the objects in the system (and are intended to guarantee the correct *interleaved* behaviour of the sliders).

```
context c:Controller inv:
  not(c.s1.inserting=true and
      c.s2.inserting=true)
```

The first constraint states that both sliders (`s1` and `s2`) associated to a controller `c` cannot simultaneously be in a state where `inserting` is true. Notice, however, that it is possible for both sliders to be waiting for a job.

The next constraint indicates that a slider can either be waiting or inserting but not both.

```
context s:Slider inv:
  not(s.inserting=true and
      s.waiting=true)
```

The following constraint specifies the behaviour of the operation (method) `startInsert()`. It can only be executed provided the slider is in state `waiting` (precondition), and after execution it changes the value of the attributes (postcondition).

```
context s:Slider::startInsert()
  pre:  s.waiting=true
  post:  s.waiting = false and
      s.inserting=true
```

The final constraint describes the behaviour of the operation (method) `endInsert()`. The postcondition, in addition to changing the values of the attributes also states that the slider sends a message to its controller to acknowledge the insert.

```
context s:Slider::endInsert()
  pre:  s.inserting=true
  post:  s.waiting = true
      and s.inserting=false and
           s.Controller^insertAck()
```

OCL as defined in the standard specification document [16] cannot be used to describe temporal constraints such as liveness and fairness. In the context of our example, we cannot express the following additional temporal contraints using OCL.

1. When a controller receives an invocation of `jobArrived()` it will eventually send a message to a (waiting) slider (either `s1` or `s2`) to start an insert.

2. When a slider starts an insert it will eventually end it and send an acknowledgement, i.e., a `startInsert()` invocation will eventually be followed by `endInsert()` which in turn will eventually be followed by an `insertAck()`.

3. Two (independent) consecutive insert requests are always given to a different slider, i.e., for two consecutive parallel invocations of `jobArrived()`, the controller sends `startInsert()` messages in parallel to different sliders.

The OCL template introduced in [7] can be used to describe simple temporal (liveness) properties of the form **after** ⟨expression⟩ **eventually** ⟨expression⟩, but it cannot be used to impose sequences of occurrences (such as invocations) and fairness properties in general. Furthermore, it is accepted that designers prefer diagrammatic notations such as sequence diagrams to indicate sequences of execution. Consequently, we use UML 2.0 sequence diagrams to capture additional behaviour and/or dynamic constraints. Our example constraints are captured by the sequence diagrams in Figure 3, Figure 4 and Figure 5 respectively.

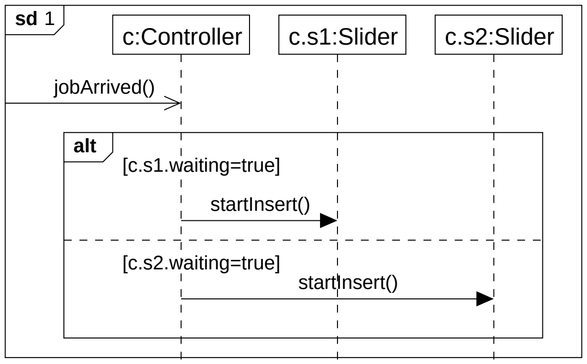


Figure 3. Modelling constraint 1.

Figure 3 shows controller `c` receiving a message `jobArrived()` (the sender is not specified and corresponds to what is called a *gate*). The arrowhead indicates that the message is sent asynchronously. The controller then sends synchronously a message to either `c.s1` or `c.s2`. This is modelled through the interaction fragment **alt** denoting alternative behaviour. An **alt** fragment contains several operands (alternatives) separated by horizontal dashed lines whereby each operand may have a guard. Only one of the operands, for which the guard evaluates to true, is executed. If more than one guard evaluates to true we have nonderterministic choice.

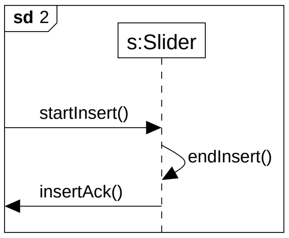


Figure 4. Modelling constraint 2.

Figure 4 shows the order of execution of the operations (methods) on a slider. Finally, Figure 5 shows a controller `c` receiving two independent messages `jobArrived()` (the square brackets along a lifeline indicate a concurrent region). The controller then enters a **par** fragment denoting parallel behaviour. A **par** fragment contains several operands (no guards) which execute in parallel. As we will see in the next section, we use an underlying true-concurrent model for interpreting sequence diagrams. Consequently, the messages sent in each operand can happen simultaneously or in any order. The idea here is that the controller tries to introduce some fairness in the allocation of jobs between sliders.

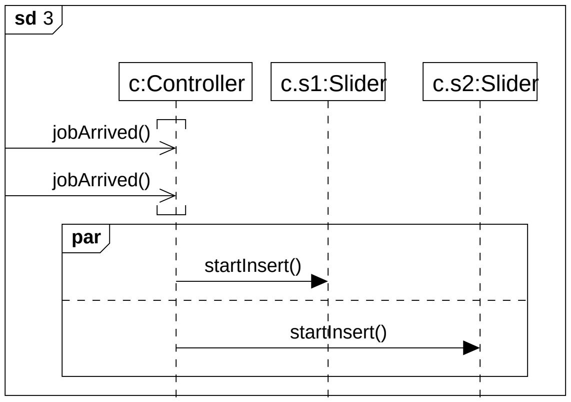


Figure 5. Modelling constraint 3.

Notice that the above model for the sliding system is not intended to be complete or correct and we shall return to such considerations later on.

3 Mathematical Model

We introduce a mathematical model that integrates the system views described in UML and whilst doing so can also be used to detect inconsistencies or incompleteness in the design. We have used labelled event structures as an underlying model for sequence diagrams in UML 2.0 [13, 5]. In this paper, we use labelled event structures for two purposes: 1) as a complete behavioural model for the objects and their relationships according to the structural view of the system (class diagram and OCL constraints); and 2) as a model that captures individual scenarios (sequence diagrams). If consistent, the models obtained in both views can be combined using the categorical construction of [5] and used for further analysis. Furthermore, our approach will give us information on completeness and/or consistency.

3.1 Event Structures: Basic Notions

We recall some basic notions on the model we use, namely *labelled prime event structures* [20].

Prime event structures, or event structures for short, allow the description of distributed computations as event occurrences together with relations for expressing causal dependency and nondeterminism. The first relation is designated *causality*, and the second *conflict*. The causality relation implies a (partial) order among event occurrences, while the conflict relation expresses how the occurrence of certain events excludes the occurrence of others. Consider the following definition of event structures.

Event Structure. An *event structure* is a triple $E = (Ev, \rightarrow^*, \#)$ where Ev is a set of events and $\rightarrow^*, \# \subseteq$

$Ev \times Ev$ are binary relations called *causality* and *conflict*, respectively. Causality $\rightarrow^*$ is a partial order. Conflict $\#$ is symmetric and irreflexive, and propagates over causality, i.e., $e\#e' \rightarrow^* e'' \Rightarrow e\#e''$ for all $e, e', e'' \in Ev$. Two events $e, e' \in Ev$ are *concurrent*, $e\ co\ e'$ iff $\neg(e \rightarrow^* e' \vee e' \rightarrow^* e \vee e\#e')$.

From the two relations defined on the set of events, a further relation is derived, namely the *concurrency* relation co. As stated, two events are concurrent if and only if they are completely unrelated, i.e., neither related by causality nor by conflict.

In our approach to inter-object behaviour specification, we will consider a restriction of event structures sometimes referred to as *discrete* event structures. An event structure is said to be *discrete* if the set of previous occurrences of an event is finite.

Discrete Event Structure. Let $E = (Ev, \rightarrow^*, \#)$ be an event structure. E is a *discrete event structure* iff for each event $e \in Ev$, the *local configuration* of e given by $\downarrow e = \{e' \mid e' \rightarrow^* e\}$ is finite.

The finiteness assumption of the so-called local configuration is motivated by the fact that system's computations always have a starting point, which means that any event in a computation can only have finitely many previous occurrences.

Consequently, we are able to talk about immediate causality in such structures. Two events are related by *immediate* causality if there are no other event occurrences in between. Formally, if $\forall_{e'' \in Ev}(e \rightarrow^* e'' \rightarrow^* e' \Rightarrow (e'' = e \vee e'' = e'))$ holds. If $e \rightarrow^* e'$ are related by immediate causality then e is said to be an *immediate predecessor* of e' and e' is said to be an *immediate successor* of e. We may write $e \rightarrow e'$ instead of $e \rightarrow^* e'$ to denote immediate causality. Furthermore, we also use the notation $e \rightarrow^+ e'$ whenever $e \rightarrow^* e'$ and $e \neq e'$.

Hereafter, discrete event structures are designated *event structures* for short.

Configuration. Let $E = (Ev, \rightarrow^*, \#)$ be an event structure and $C \subseteq Ev$. C is a *configuration* in E iff it is both (1) conflict free: for all $e, e' \in C$, $\neg(e\#e')$, and (2) downwards closed: for any $e \in C$ and $e' \in Ev$, if $e' \rightarrow^* e$ then $e' \in C$. A maximal configuration denotes a run. A run is sometimes called life cycle.

Finally, in order to use event structures to provide a denotational semantics to languages, it is necessary to link the event structures to the language they are supposed to describe. This is achieved by attaching a labelling function to the set of events. A generic labelling function is as defined next.

Labelling Function. Let $E = (Ev, \rightarrow^*, \#)$ be an event structure, and L be an arbitrary set. A *labelling function* for E is a total function $l : Ev \rightarrow L$ mapping each event into an element of the set L.

An event structure together with a labelling function defines a so-called labelled event structure.

Labelled Event Structure. Let $E = (Ev, \rightarrow^*, \#)$ be an event structure, L be a set of labels, and $l : Ev \rightarrow L$ be a labelling function for E. A *labelled event structure* is a pair $(E, l : Ev \rightarrow L)$.

Usually, events model the occurrence of actions, and a possible labelling function maps each event into an action symbol or a set of action symbols. In this paper, we use labelled event structures for two different purposes and as such we will need two labelling functions. When using event structures to model the possible behaviour of objects given a signature as defined in the class diagram, the labelling function maps each event to a logical formula indicating its state (enabled operations, occurring operations and values of attributes). When using event structures to model sequence diagrams in UML 2.0, the labelling function indicates whether an event represents sending or receiving a message, a condition, the beginning or end of an interaction fragment.

3.2 Categorical Construction

In this section we describe the main idea of a categorical construction for event structures which we use to combine different models. We omit details of categorical properties of event structures and refer the interested reader to [5, 20]. It suffices to understand that we consider two categories (**ev** and **cev**) over event structures (based on two notions of morphisms). For this paper, we only need to introduce the morphism notion for the category **ev** (from [20]).

Event Structure Morphism. Let $E_i = (Ev_i, \rightarrow_i^*, \#_i)$ for $i = 1, 2$ be event structures, and $C \subseteq Ev_1$ an arbitrary subset of events. A *morphism* from E_1 to E_2 consists of a partial function $h : Ev_1 \rightarrow Ev_2$ on events satisfying both (1) if C is a configuration in E_1 then $h(C)$ is a configuration in E_2, and (2) for all $e, e' \in C$, if $h(e), h(e')$ are defined and $h(e) = h(e')$ then $e = e'$.

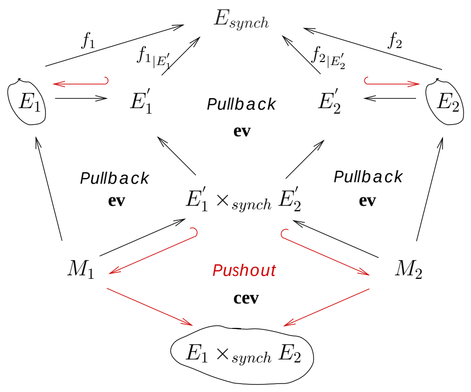
The notion of event structure morphism as given before preserves the concurrency relation, as has been proved in [20]. The intuition behind the morphism is that the occurrence of an event is matched (synchronised) with the occurrence of its image.

We want to be able to combine arbitrary (labelled) event structures by synchronising some of their events whilst leaving the remaining events and relations unchanged. There is no immediate categorical result on **ev** (or even **cev**) that gives us exactly what we need. It turns out that the model synchronisation as we need it can be obtained in several steps when using both categories. Before introducing the construction we define the diagram over which it is based.

Synchronisation Diagram Let E_1 and E_2 be two event structures. A *synchronisation diagram* for E_1 and E_2 is given by a triple $S = (E_{synch}, f_1, f_2)$ where E_{synch} is a nonempty event structure, and f_i with $i \in \{1, 2\}$ are two surjective event structure morphisms such that $f_i : Ev_i \rightarrow Ev_{synch}$, and satisfying $f_1(Ev_1) = f_2(Ev_2)$. Moreover, E_{synch} is called the *synchronisation event structure* of E_1 and E_2.

If a synchronisation diagram is not definable we say that the models are not composable. The synchronisation diagram tells us how the models relate.

Categorical Construction Let E_1 and E_2 be two event structures with a synchronisation diagram given by $S = (E_{synch}, f_1, f_2)$ where $f_i : Ev_i \rightarrow Ev_{synch}$ for $i \in \{1, 2\}$. Let $E_i^{'}$ be the maximal event substructure of E_i such that $f_{i|E_i^{'}}$ is a total morphism. Then doing the pullbacks in **ev** and the pushout in **cev** as depicted, we obtain the concurrent composition of E_1 and E_2, written $E_1 \times_{synch} E_2$, in accordance with the synchronisation diagram S.



The interesting aspect of the above construction is that it combines pullbacks in **ev** and one final pushout in **cev** in such a way that the pullbacks are done over fully synchronised event structures and we always obtain morphisms in **cev** satisfying the necessary conditions for the existence of the the final pushout.

3.3 Event Structures for Class Diagrams and OCL

In this paper, we are not interested in how to obtain an event structure model for a class diagram and additional OCL constraints. We are interested in how such a model can be combined with a model for a sequence diagram in order to obtain a more complete behavioural model or detect inconsistencies. We thus assume given a model for the structural view of the system. The labelling function of such a model is defined below and requires the notion of a class diagram signature.

First we recall the definition of a data signature $\Sigma_D = (S_D, \Omega_D)$, where S_D is a finite set of *data sorts*, and Ω_D is an $S_D^* \times S_D$-indexed family of sets of *data operations*. For each $o \in \Omega_{D w,s}$, w is the *parameter list* of the operation o and s the *result sort*. The elements of $\Omega_{D \varepsilon,s}$ are called *constant symbols* of sort s.

Let X be an S_D-indexed family of disjoint sets of *variables*. A data signature may be extended with variables by considering them as constant symbols of a given sort. A data signature with variables is sometimes written $\Sigma_D(X)$. From the symbols defined in the data signature and the variables we can construct *data terms* in the usual way. $T_{\Sigma_D,s}(X)$ denotes the set of data terms of sort s over $\Sigma_D(X)$. $T_{\Sigma_D}(\emptyset)$ is the family of closed terms, also written T_{Σ_D}. Terms denote a certain value, so they can be evaluated under a given interpretation. A data signature Σ is interpreted over Σ-algebras but we omit further details.

For describing a class diagram signature, apart from data sorts and data operations as above, we will need *object* sorts S_O and operations on them Ω_O. Intuitively, each class is equipped with an object sort.[1]

A class describes the *attributes* and *methods*[2] of its potential *instances*. Attributes, methods, and instances can be understood as special object sort operations, but we need to be able to distinguish them: we need to know whether a certain object operation is an attribute, method or else. Thus, we distinguish between: an *attribute object sort* (S_O^{at}), a *method object sort* (S_O^m), and an *instance object sort* (S_O^i).

The following defines a class diagram signature.

Class Diagram Signature A *class diagram signature* is a signature $\Sigma = (S, \Omega)$ such that:

- S is a finite set of sorts, *data* and *object* sorts, that is, $S = S_D \cup S_O$ where: $S_O = S_O^i \cup S_O^{at} \cup S_O^m$ is a disjoint union of sets of object sorts.

- Ω is an $S^* \times S$-indexed family of sets of operation symbols such that $\Omega_D \subseteq \Omega$. Let $S^i = S_O^i \cup S_D$. Further operations in Ω are
  - $\Omega_{x^i s^i}$ with $x^i \in S^{i*}$ and $s^i \in S_O^i$, object instance operations;
  - $\Omega_{s^i s^{at}, r^i}$ with $s^i \in S_O^i$, $s^{at} \in S_O^{at}$ and $r^i \in S^i$, attribute operations;
  - $\Omega_{s^i x^i, s^m}$ with $s^i \in S_O^i$, $x \in S^{i*}$ and $s^m \in S_O^m$, method operations;

Attribute and method operations are always associated with an object instance sort. For example, the attribute operation $a \in \Omega_{s^i s^{at}, r^i}$ is associated to an object instance sort given by s^i. (We antecipate that this is to be able to build

[1]Strictly speaking, to describe class diagrams we want to be able to capture inheritance which can be done by introducing a partial order over object sorts and using a so-called *order-sorted* data signature. We are not considering inheritance in this paper.

[2]We avoid the usual term operation in this context to avoid confusion.

terms in a special way.) Moreover, s^{at} indicates that the operation is an attribute and r^i is the sort of the attribute. For a method operation $m \in \Omega_{s^i x^i, s^m}$, x^i denotes the sorts of the arguments including possibly the result sort.

From a class diagram signature we can construct not only data terms (as usual) but instance, attribute and method terms for objects. For a given signature $\Sigma(X)$, we will denote $T_\Sigma(X)$ the family of sets of data and instance terms, $AT_\Sigma(X)$ the family of sets of attribute terms, and $ME_\Sigma(X)$ the family of sets of method terms.

Instance terms are constructed like data terms. Attribute terms have the form $t.a$ where t is an instance term and a an attribute operation. The sort of the term $t.a$ is given by the result sort of a, i.e., if $a \in \Omega_{s^i\ s^{at}, r}$ then the attribute term $t.a$ has sort r and t is a term of sort s^i. Method terms have the general form $t.c(t_1, \ldots, t_n)$ where t is an instance term, c denotes a method, and there is a list (possibly empty) of argument terms $t_1 \ldots t_n$. The sort of a method term $t.c(t_1, \ldots, t_n)$ is given by the method sort of c, i.e., if $c \in \Omega_{s^i x, s^m}$ then the method term is of sort s^m. Closed data and instance, attribute, and method terms are written T_Σ, AT_Σ and ME_Σ respectively.

The model for the structural view of a system with class diagram signature Σ is given by $M_\Sigma = (E, \mu_\Sigma)$ such that the labelling function is a total function defined over:

$$\mu_\Sigma : Ev \to 2^{\Phi_\Sigma}$$

where Φ_Σ are atomic formulae of a *state logic* defined over the signature Σ as follows.

$\Phi_\Sigma ::= AT_{\Sigma,s} = T_{\Sigma,s} \mid ME_\Sigma$

The syntax of Φ_Σ defines atomic formulae of a state logic (not shown) and is used to express the states of the system with class diagram signature Σ. The state of the system is given by the current values of the attributes and occurring methods of all objects in the system. A formula of Φ_Σ can be the predicate $=$ applied to a closed attribute term and a closed data term, or a closed method term. The labelling function associates an event to a set of atoms, namely pairs of attribute-value and occurring methods.

As we said earlier, in this paper we are not concerned with how to obtain a model for a class diagram and additional OCL constraints and we assume the model given. The idea is, however, that the model is obtained by concurrent composition of instance models. OCL constraints restrict the possible runs in the model. In our example, we obtain a model for the class diagram in Figure 2 by composing the models of instances of controller and slider, for example instances c, $c.s1$ and $c.s2$ ($s1$ and $s2$ for short). Let $M_\Sigma = (E, \mu_\Sigma)$ be a model for the class diagram Σ. We can also refer to the individual instance models M_c, M_{s1} and M_{s2}. Examples of event labels are for $e_1 \to e_2$:

$\mu_\Sigma(e_1) = \{s1.waiting = true, s1.inserting = false,$
$\quad s2.waiting = true, s2.inserting = false\}$

and for $e_2 \in Ev_{s1}$ a label could be:

$\mu_\Sigma(e_2) = \{s1.waiting = false, s1.inserting = true,$
$\quad s2.waiting = true, s2.inserting = false,$
$\quad\quad s1.startInsert(), c.startInsert()\}$

There is an infinite number of events with these labels. Because of the OCL constraints, we know that an event e_3 with label

$\mu_\Sigma(e_3) = \{s2.waiting = false, s2.inserting = true,$
$\quad s1.waiting = true, s1.inserting = false,$
$\quad\quad s2.startInsert(), c.startInsert()\}$

is such that e_2 and e_3 can be related by causality in either way or by conflict but not be concurrent.

3.4 Event Structures for Sequence Diagrams

In [13] we have shown how labelled event structures can be used to provide a model for sequence diagrams. Here we only provide the general idea.

To obtain the corresponding event structure model, we want to associate events to the *locations* of the diagram and determine the relations between those events to reflect the meaning of the diagram. Figure 6 shows the relation between the locations in a simple sequence diagram and the corresponding event structure model (where we depict immediate causality). Synchronous communication is captured as a shared event, hence the two locations for sending and receiving message `endInsert()` are captured by one event only. The labels become clearer later when we define the labelling function used.

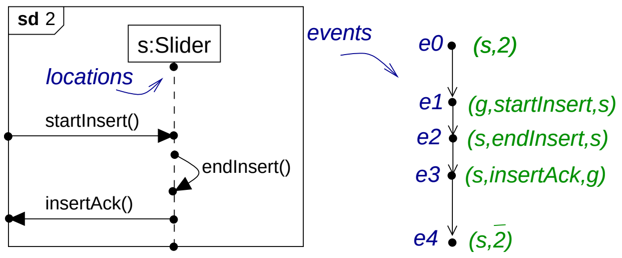


Figure 6. A simple sequence diagram and its corresponding model.

However, for more complex diagrams with fragments the correspondence between locations and events is not always so obvious.

The locations within different operands of an **alt** fragment are naturally associated to events in conflict. However, the end location of an **alt** fragment is problematic. If it corresponded to one event then this event would be in conflict with itself due to the fact that in a prime event structure conflict propagates over causality. This would, however, lead to an invalid model since conflict is irreflexive. We are therefore forced to copy events for locations marking the end of

alt fragments, as well as for all locations that follow. Events associated to locations that fall within a **par** fragment are concurrent. Synchronous communication is denoted by a shared event whereas asynchronous communication is captured by immediate causality between the send event and receive event.

As mentioned earlier, for representing sequence diagrams we use a labelling function to indicate whether an event represents sending or receiving a message, a condition, the beginning or end of an interaction fragment. The only considered fragments in this paper are **alt** and **par**. The fragment **ref** was treated in [5] and is not considered here to simplify the presentation.

Let D be a set of diagram names, I_d be the set of instances participating in the interaction described by $d \in D$, and g denote an unspecified instance or gate with $g \in I_d$ for all $d \in D$. Let $F_D = \{d, par, alt\}$ where $d \in D$, and $\overline{F_D} = \{\overline{d}, \overline{par}, \overline{alt}\}$. We use par (or $\overline{par}$) as a label of an event associated to the location marking the beginning (or end) of a **par** fragment. In particular, events associated to initial (or end) locations of a diagram d have labels d (or $\overline{d}$). Let Mes be a set of message labels. The labelling function for diagram d is a total function defined over:

$$\mu_d : Ev \to I_d \times (Mes \times \{s, r\} \cup F_D \cup \overline{F_D}) \cup I_d \times Mes \times I_d$$

The first part of the codomain is used to describe asynchronous messages or beginning/end of fragments, whilst the second part of the codomain deals with synchronous messages.

Finally, for a diagram $d \in D$, a model is a labelled event structure $M_d = (E_d, \mu_d)$.

4 Integrating Views

We have introduced our model, namely labelled event structures, and seen how it can be used with different labelling functions for the two views (structural and behavioural). In this section, we discuss how we can combine the different views using our categorical construction. This corresponds to defining synchronisation diagrams for the models of the views and applying the construction.

In our example, we have a model M_Σ for the structural view and three models for the sequence diagrams given M_{sd1}, M_{sd2} and M_{sd3}. The way to combine these individual models to obtain the final complete model for our system is arbitrary. We can either combine M_Σ with the sequence diagram models in a series of steps, or obtain first the model for all scenarios and then combine this with M_Σ. Here, we will see what happens if we try to combine M_Σ with the model of sequence diagrams 2 and 3.

Without giving the synchronisation diagram required for combining the models of sequence diagrams 2 and 3, the idea is that we duplicate the model of sequence diagram 2 for instances $s1$ and $s2$ and synchronise both models on the shared events labelled with $startInsert()$ to obtain the model in Figure 7 (for space reasons the names of the methods are abbreviated in the labels). Given that there is no

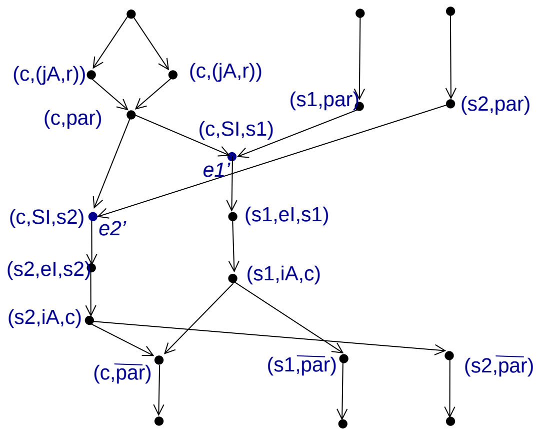


Figure 7. Combined model for sequence diagrams 2 and 3.

choice in behaviour in both diagrams, all events are either related by causality or concurrent. In particular, $e_1^{'}\ co\ e_2^{'}$. Let this model be M_{2+3}. In order to be able to combine it with the corresponding model for the structural view we first define in general the synchronisation diagram for different views.

View Synchronisation Diagram Let $M_\Sigma = (E_\Sigma, \mu_\Sigma)$ be a model for a structural diagram with signature Σ and set of instances I, and $M_d = (E_d, \mu_d)$ be a model for a sequence diagram d. A *view synchronisation diagram* for M_Σ and M_d is a synchronisation diagram $S = (E, f_1, f_2)$ with $f_1 : Ev_\Sigma \to Ev$, $f_2 : Ev_d \to Ev$ both surjective and such that for $e \in Ev_\Sigma$ and $e^{'} \in Ev_d$, $f_1(e)$ and $f_2(e^{'})$ are defined and $f_1(e) = f_2(e^{'})$ iff one of the following cases applies:

1. For $i \in I \cap I_d$, $i.a() \in \mu_\Sigma(e)$ and $\mu_d(e^{'}) = (i, (a, r))$.
2. For $i \in I \cap I_d$, $i.a(), j.a() \in \mu_\Sigma(e)$ and $\mu_d(e^{'}) = (i, a, j)$ or $\mu_d(e^{'}) = (j, a, i)$.

The synchronisation diagram identifies the events from both models which should be matched (synchronised). It only matches events labelled by the occurrence of a method (structural view) and the receipt of a message (sequence diagram) with the same signature. Similarly for cases of synchronous communication.

If there is a view synchronisation diagram for the models of two views we apply the categorical construction introduced earlier to obtain the final combined model. This model can be optimised to remove any events (from the sequence diagrams) corresponding to locations marking the beginning/end of fragments. If the view synchronisation diagram is not definable we can identify the events in both views that create a problem.

Consider the (partial) model for the structural view in Figure 8. with following labels (with shorthand notation

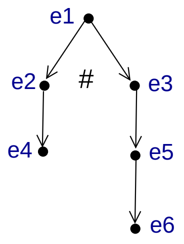


Figure 8. Partial model for structural view.

for attributes and methods):
$\mu(e_1) = \{s1.w = true, s1.i = false, s2.w = true, s2.i = false\}$
$\mu(e_2) = \{s1.w = true, s1.i = false, s2.w = false, s2.i = true, s2.sI(), c.sI()\}$
$\mu(e_3) = \{s1.w = false, s1.i = true, s2.w = true, s2.i = false, s1.sI(), c.sI()\}$
$\mu(e_5) = \{s1.w = true, s1.i = false, s2.w = true, s2.i = false, s1.eI()\}$
$\mu(e_6) = \{s1.w = true, s1.i = false, s2.w = false, s2.i = true, s2.sI(), c.sI()\}$

To combine the view models of Figure 8 and Figure 7 we would want to match events e_3 and $e1^{'}$, and events e_2 and $e2^{'}$. In this particular case, we can define a view synchronisation diagram with event structure morphisms f_1 and f_2 such that $f_1(e_3) = f_2(e_5^{'})$ and $f_1(e_2) = f_2(e_2^{'})$. However, in order for the event structure morphisms to be valid f_1 maps the conflicting events into two events in concurrency. Through the categorical construction we will ultimately loose the concurrency between events $e1^{'}$ and $e2^{'}$ in the final model, and effectively remain (as intended) in conflict as in model M_{Σ}. In this case, our approach serves to discard models from incorrect diagrams such as sd 3.

5 Related and Future Work

The main objective of this paper is to present a mathematical formalism which allows the integration of multiple UML 2.0 models. The formalisation of UML diagrams has received considerable attention in recent years, including approaches for defining a semantics to component diagrams [15] and sequence diagrams [8, 13] among many others. However, despite the vast range of approaches formalising individual UML diagrams, very little has been done on integrating several UML diagrams in a unique formalism. Ehrig et al [10] presents an algebraic framework for the composition of models for a component-based system. This work is further extended in [9] to apply to UML diagrams. However, their approach essentially deals with structural aspects of models representing various aspects of component design. In contrast our approach uses a single semantic domain, labelled event structures, for interpreting the structural and behavioural views of a system. In order to be able to obtain a unique mathematical model for a system their design models must consistent. Ensuring the consistency in design has recently received considerable attention. Among others, existing approaches use temporal logic [15], variants of automata [1], B and refinement calculus [12] and Petri nets [2].

In our approach consistency between models can be identified if we are able to define a view synchronisation diagram. If there is inconsistency between models we expect our contruction to identify configurations in either model that are inconsistent. On the other side, we showed that a sequence diagram which did not adequately describe intended behaviour for the system was in effect discarded when applying the categorical construction to the two view models. A clearer way to extract information from our mathematical construction in the form of feedback for designers needs to be further investigated.

The idea of multiple view modelling have been extensively studied by Open Distributed Processing (ODP) community. For example, refinement techniques has been successfully used to study the consistency of viewpoints [6]. These methods are different from our approach as they establish the consistency between viewpoints by showing existence of a common implementation.

The advantage of our approach lies in combining multiple view models whilst detecting possible inconsistencies. Moreover, we believe that it can be useful in identifying incomplete behaviour. When designers specify behaviour using scenario-based languages their obtained behaviour specifications are generally incomplete. In future work, we want to investigate how our approach can be used to identify missing scenarios of behaviour. Finally, we envisage the development of tools to support our formal approach. As mentioned earlier, the translation of formal results into feedback for the designer is essential. This should rely entirely on notation known to the designer, namely UML, and hide any technical details.

References

[1] L. Alfaro and T. Henzinger. Interface automata. In *9th Annyal Symposium on Foundations of Software Engineering*, pages 109–120, 2001.

[2] R. Bastide and E. Barboni. Component-based behavioural modelling with high-level Petri nets. In *MOCA 2004*, pages 37–46, 2004.

[3] A. Beugnard, J.-M. Jézéquel, and N. Plouzeau. Making components contract aware. *IEEE Computer*, 32:38–45, 1999.

[4] B. Bordbar, J. Derrick, and G. Waters. Using UML to specify QoS constraints in ODP. *Journal of Computer Network and ISDN systems*, pages 279–304, 2002.

[5] J. K. F. Bowles. Decomposing Interactions. In M. Johnson and V. Vene, editors, *Algebraic Methodology and Software Technology (AMAST 2006)*, volume 4019 of *LNCS*, pages 189–203. Springer, 2006.

[6] H. Bowman, M. Steen, E. Boiten, and J. Derrick. A formal framework for viewpoint consistency. *Formal methods in design*, 21:111–166, 2002.

[7] J. Bradfield, J. Küster-Filipe, and P. Stevens. Enriching OCL using observational mu-calculus. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering (FASE 2002)*, volume 2306 of *LNCS*, pages 203–217. Springer, 2002.

[8] M. Broy. A semantic and methodological essence of message sequence charts. *Science of Computer Programming*, pages 213–256, 2005.

[9] H. Ehrig, B. Braatz, M. Klein, F. Orejas, S. Perez, and E. Pino. Object-oriented connector-component architectures. In *FESCA'05*, volume 141 of *ENTCS*, pages 123–151, 2005.

[10] H. Ehrig, J. Padberg, B. Braatz, M. Klein, F. Orejas, S. Perez, and E. Pino. A generic framework for connector architectures based on components and transformations. In *FESCA'04*, volume 108 of *ENTCS*, pages 53–67, 2004.

[11] C. Kobryn. Modelling components and frameworks with uml. *Communications of the ACM*, 43:31–38, 2000.

[12] O. Kouchnarenko and A. Lanoix. Refinement and verification of synchronized component-based systems. In *Formal Methods Europe 2003*, volume 2805 of *LNCS*, pages 341–358. Springer, 2003.

[13] J. Küster-Filipe. Modelling Concurrent Interactions. *Theoretical Computer Science*, 351(2):203–220, February 2006.

[14] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *ESEC'95*, volume 989 of *LNCS*, pages 137–153. Springer, 1995.

[15] T. Nguyen and T. Katayama. Specification and verification of inter-component constraints in ctl. In *Specification and Verification of component-based systems*, 2005.

[16] OMG. *UML 2.0 OCL Specification, Version 1.6*. OMG document ad/03-01-07, available from www.uml.org, August 2003.

[17] OMG. *UML 2.0 Superstructure Specification*. OMG document ptc/05-07-04, available from www.uml.org, August 2005.

[18] F. Plasil and S. Visnovsky. Behaviour protocols for software components. *IEEE Transactions on Software Engineering*, 28:1056–1076, 2002.

[19] J. Putman. Model for fault-tolerance and policy from RM-ODP expressed in UML/OCL. In *ISORC 2000*. IEEE Computer Society, 2000.

[20] G. Winskel and M. Nielsen. Models for Concurrency. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. 4, Semantic Modelling*, pages 1–148. Oxford Science Publications, 1995.