Navigation Expressions in Object-Oriented Modelling

Ali Hamie, John Howse, Stuart Kent
Division of Computing,
University of Brighton, Lewes Rd., Brighton, UK.
http://www.biro.brighton.ac.uk/index.html
biro@brighton.ac.uk[1]

Abstract. In component-based development, object-oriented modelling notations such as UML are being proposed as a way of providing richer specifications of components. Much more so than in bespoke software development, this requires a high level of precision coupled with sufficient expressive power. Expressive power is delivered by adding textual annotations, such as invariants, pre & post conditions, to diagrams. Navigation expressions, which identify collections of objects by navigating associations, are central to making such annotations precise. We give a semantics to navigation expressions as they are used in recently proposed extensions to object-oriented modelling notations *in widespread use by practitioners*. The semantics is given using Larch (essentially FOPL), which makes it as accessible as possible while enabling some support for reasoning. The semantics helps to clarify some subtle issues to do with navigation expressions, including the meaning of navigating across collections (sets, bags and sequences) as opposed to just single objects, and the use of filters on collections within expressions.

1 Introduction

Modern object-oriented modelling notations, such as UML [19], [7], are based on graphical notations for expressing a wide variety of concepts which are relevant to a problem domain. While these notations are intuitive and easy to understand by users, they lack expressive power. Kent [15] shows that in order to write some constraints on the behaviour of a system it is necessary to step outside the diagrams and write them textually. Navigation expressions are critical to making these textual languages precise, which we argue is essential to enable the current advance of software engineering towards component-based development. Semantics of such languages, hence navigation expressions, will be required to (a) check the integrity of the language and (b) support the development of CASE tools.

For bespoke software development, it is possible in many cases to get by with informal, imprecise annotations. This is because models are often discarded at the end of a development, because short-term economic pressure mitigates against them being maintained and kept up to date as the code is developed and tested: Why spend a lot of time making these models precise if they are only going to be thrown away? Of course, putting the effort into making them more precise and then maintaining them would likely pay off in the long term, as precision early in the development cycle is likely to lead to cleaner designs and code and less need for testing. As documentation such models can be invaluable, as they avoid implementation detail, allowing maintainers to uncover the essence of the software design more quickly.

1. This research is partially funded by the UK EPSRC under grant number GR/K67304

In component-based development (CBD), however, the requirement for precision cannot be so lightly discarded. Object-oriented modelling notations are being proposed (e.g. [18], [1]) as an approach to documenting component interfaces in a more accessible and more detailed form than is the case in CBD technologies such as Microsoft's COM, the Object Management Group's (OMG) CORBA and SunSoft's Java Beans, which rely upon a list of operations with their signatures, accompanied with some informal, though not necessarily informative, descriptions. The need for precision and formality when using these notations, both diagrams and text, for CBD is elaborated in [18] and [16]. Precise, expressive specifications are required to facilitate searching and matching of components and component assembly. Precision is essential for the automation of these processes. A user of a component will require a certificate giving her confidence that the component does what is claimed. This is especially important when components are "black box", where the design and implementation is not supplied. Confidence in certificates will only be achieved if appropriate techniques are used. This means, for example, the use of precise models at all levels of the construction process, enabling the implementation to be traced back to the specification and conformance of the implementation against the specification to be checked. In other words a rigorous approach to refinement should be supported.

CBD also increases the importance of specification models. Components must be described in business (requirements) oriented terms rather than implementation oriented terms, as the focus is always on using the component rather than on how it works. Indeed, often there may be a considerable mismatch between the specification and implementation. Combined with a need for precision and expressiveness, this has led to the extension of diagrammatic notations such as UML with a precise textual language for expressing pre/post conditions and invariants, where the latter allow modellers to abstract away from implementation detail. Syntropy [4] extends OMT [17] with a Z-like textual language for adding invariants to class diagrams and annotating transitions on state diagrams with pre/post conditions. Catalysis [5], [6] does something very similar for UML, adopting an arguably simpler and more usable approach. The Catalysis and Syntropy notations have now been superseded with the Object Constraint Language (OCL) which has recently become part of the UML standard.

Semantics work [2], [3], [8] for OO modelling notations in widespread use, such as OMT or UML, is generally restricted to capturing the meaning of those notations, so navigation expressions are not considered, as they are not officially part of those notations at least as far as their use in pre/post conditions and invariants are concerned. The semantics of the extensions is at best rigorous. For example in [4] the semantics comprises six pages of informal text interleaved with examples.[1] The semantics given for Catalysis in [5] and [6] is similarly informal. A key result of this paper is to check this intuitive semantics. One specific result is to show that the introduction of so-called "flat sets" in the semantics, as suggested in [6], is not necessary.

The focus of this paper, then, is on the use of navigation expressions in writing invariants, pre and post conditions in UML extended with a precise textual language. As the work in this paper was completed before the publication of OCL in the UML 1.1. standard, we use the textual language of Catalysis, one of its immediate predecessors. However, the

1. Recently, a semantics has been given for a part of Syntropy ([2]) but only cursory coverage of navigation expressions is given.

work is applicable to any precise textual language which uses navigation expressions. Indeed, since the original submission of this paper we have been using the work presented here as a basis for the semantics of OCL [14].

The semantics is characterised in terms of the Larch Shared language (LSL) [11] which is essentially a syntax for writing and composing theories in many-sorted first order predicate logic with equality. This follows a well-rehearsed approach to semantics, dating back at least to Burstall and Goguen [9], whereby specifications are given a semantics in terms of (compositions of) theories of some logic.

Larch is a mature language which comes with a toolset including a sophisticated proof assistant. The choice of Larch was motivated by the desire not to be engaged in the design of logics and reasoning systems, but instead to focus on elaborating the meaning of the modelling notations themselves in a way that is widely accessible. It was also chosen because it supports a compositional approach to semantics [13].

Section 2 is an informal introduction to navigation in object-oriented modelling. Section 3 establishes the semantic framework by giving a semantics to class diagrams and associations. Section 4 defines the semantics of navigation expressions, considering in turn the use of navigation expressions in invariants, the semantics of filters, the use of navigation expressions in pre and post conditions, and the semantics of navigation expressions considering navigation over collections other than sets, in particular sequences and bags. Section 5 defines the general mapping of class diagrams in UML and terms in the Catalysis textual language to expressions in Larch.

2 Navigation in Object Oriented Modelling

In order to understand what is meant by navigation in object oriented modelling, it is necessary first to understand what is an object oriented model, which essentially comprises two parts:

- a generic model, which is made up of a collection of diagrams and textual annotations modelling the general behaviour of a system, usually a software system;
- one or more specific models, each a collection of diagrams illustrating specific examples of system behaviour.

In UML, the language of choice for this paper, the class diagram is central to any generic model. An example of this for a simple course scheduling system is given in Figure 1. This is a system for scheduling *presentations* of *courses* to a collection of *students*, with *instructors* who must be *qualified for* the course in question. As this is a specification model, the boxes are interpreted as types or interfaces, rather than classes, as they carry no implementation information. A full description of the notation can be found in [19] or [7]. The diagram is best explained by giving an example of (part of) a specific model. Figure 2 is an object diagram depicting an example state of the system at a particular point in time. It includes objects, depicted by boxes, and links. For this to be a specific model of the generic model described in part by Figure 1, the types of objects, indicated by :Type, must be of a type mentioned in the class diagram, and the links must correspond to (appropriately labelled) associations. Furthermore, the number of links between objects must obey the cardinality constraints imposed by the class diagram. For example, the latter states that a course may have zero, one or many qualified instructors (shown by a * appearing at the end labelled `qualified` of the association between `Course` and `Instructor`). This is clearly

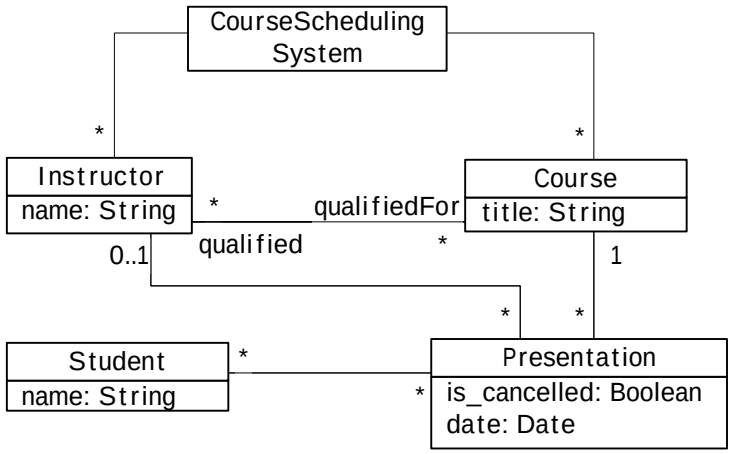

Figure 1: Class diagram for a course scheduling system

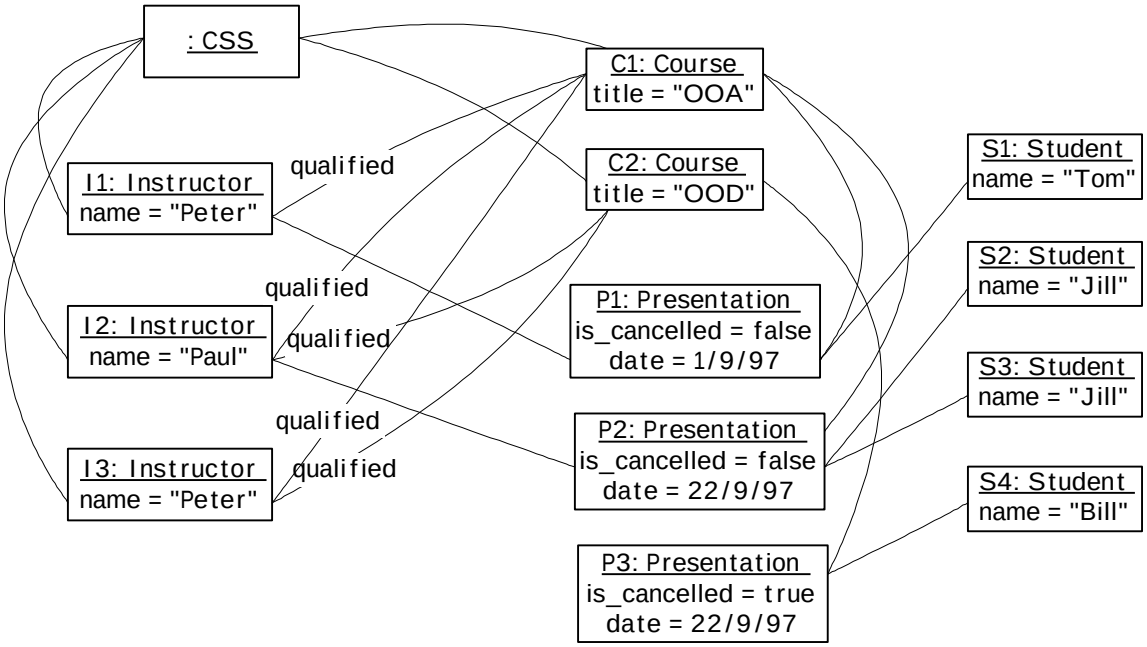

Figure 2: Object diagram

the case in the object diagram. It also states that a presentation may have at most one instructor: there are two presentations each linked to exactly one instructor and one with no link to an instructor. And so on. Objects on an object diagram may be given an explicit identity, to make it easy to refer to them in any explanation. Here, I1, I2, etc. are object identities. The only object without an identity is the one of type CSS (short for CourseSchedulingSystem). Values of attributes may also be shown on an object diagram and these should, of course, correspond to the attribute declarations in the class diagram. Thus the class diagram may be thought of as defining a set of object diagrams, namely the ones which are consistent in the way described. It identifies the types of object allowed in the system, and, via associations, the kind and number of links. It also identifies attributes that objects can have with their return types.

Navigation in OO modelling means following links from one object to locate another object. It is possible to navigate across many links, or navigate from a collection to a collection. *Navigation expressions* allow us to write in generic models constraints on the

behaviour of objects identified by navigating from the object or objects which are the focus of the constraint. At the specification level, the expressions appear in invariants, pre and post conditions. Examples of these will be given later; for this section we'll just consider basic navigation expressions in isolation.

In order to write a navigation expression we must start with an object of known type and we must have a way of referring to that object. Given the object type Course, a declaration as c:Course means that c is a variable that can refer to an object taken from the set of objects conforming to type Course. Here, the type name is used to represent the set of objects in the model that conforms to the type. A navigation expression is written using ".", an attribute or role name, and an optional parameter list. Given this declaration, the expression c.title represents the value of the attribute title for the object represented by c, namely, the title of course c. In this case the navigation expression yields a value of type String, which means that it is not possible to navigate any further. For example, if the variable c is assigned to the object C1 in the object diagram then the meaning of the expression c.title is the string "OOA".

Given the declaration i:Instructor, the expression i.qualifiedFor denotes the set of courses associated with instructor i. If the variable i is assigned to the object I2 then the meaning of the expression i.qualifiedFor is the set {C1, C2} consisting of C1 and C2. If the value of a navigation expression is another object or set of objects then we could navigate on to their attributes. In this case, the value of the expression i.qualifiedFor is a set of objects; any subsequent navigation must be applied to each member of the set and the result is a set constructed from the set of objects located. For example, the expression i.qualifiedFor.title yields a set of strings, namely {"OOA", "OOD"}. In this case the attribute title is applied to the set i.qualifiedFor (using the "." operator) and the result is a set whose members are the results of applying title to the members of i.qualifiedFor.

In Syntropy, Catalysis and now OCL is the idea of navigating across collections. For example, the expression i.qualifiedFor.presentations represents the set whose members are all the Presentation objects which can be got by traversing qualifiedFor and presentations. This is obtained by evaluating the expression i.qualifiedFor yielding a set of Course objects, then by navigating from each member of this set using presentations, to obtain a set of sets of Presentation objects. The resulting set is the union of these sets. Thus if i is I2 then i.qualifiedFor.presentations is the set {P1,P2,P3}, whereas if i is I1, the set is {P1,P2}, i.e. the presentations associated with course C1 only.

Another navigation expression which may occur is s[pred], where s is a set and pred is a boolean predicate. This expression denotes the set of members in the set s for which the predicate pred is true. That is, the set s is *filtered* using the predicate pred. In Figure 1, the expression c.presentations is the set of presentations for a course c. The expression c.presentations[is_cancelled] is the set of all presentations of course c, for which is_cancelled is true, i.e. the set of cancelled presentations. For example, if c is assigned to the object C2 then the meaning of the expression c.presentations[is_cancelled] is the singleton set {P3}.

The above describes navigation solely in terms of sets. More generally, navigation may occur over collections, including bags and sequences. This is discussed more fully in Section 4.4. Another extension of navigation expressions, as described, is to use them to constrain operation invocations, for example on state diagrams or sequence diagrams. This

is similar to the way in which navigation expressions are used in OO programming, and is not the focus of this paper.

3 Interpreting Class Diagrams

As explained in the introduction, we use the *Larch Shared Language* (LSL) [11] to illustrate how object types, associations, and navigation expressions are incorporated into structured specification. LSL uses specification modules, called *traits*, to describe abstract data types and theories. Traits are presented in the following form:

SpecName(*parameters*) : **trait**
includes
existing specification modules to be used
introduces
function signatures are listed here
asserts
axioms are listed here

SpecName is the name of the specification module or trait (not the name of a sort). Following the name, is the list of *sorts* and *operators* that form the parameters of the trait. The *includes* section lists other traits on which the specification is built. The *introduces* section lists a set of *operators* (function identifiers) together with their *signatures* (the sorts of their domain and ranges). Overloading of operators is allowed in LSL. All operators used in a trait must be declared so that *terms* can be sort-checked in the same way as function calls are type-checked in programming languages. The *asserts* section lists the axioms that constrain the operators expressed in first-order predicate logic with equality. An equation consists of two terms of the same sort, separated by ″==″. If one term of an equation is ″true″ then the equation can be abbreviated to just the other term. When using LSL, it is assumed that a basic axiomatisation of Boolean algebra is part of every trait. This axiomatisation includes the sort Bool, the truth values true and false, the logical connectives ″∧″, ″∨″, ″⇒″ and ″¬″.

3.1 Object Types

An object type is a description of a set of objects in terms of properties and behaviour they all share. In our formalisation, an object type is associated with an LSL basic sort consisting of elements that uniquely represent objects (instances) of the type, which can be thought of as object identifiers. The attributes of an object type are formalised as functions with the appropriate signatures.

The object type Course in Figure 1 is interpreted as a basic sort denoted by Course, namely a sort of object identifiers. The attribute title[1] is interpreted as a function title with signature title : Course → String, which is added to the specification for object type Course. The type String is interpreted as the sort of strings String. The definition of attribute functions varies with the system states, so no additional constraints are required.

According to the above description, the specification of the object type Course has only two sorts: Course and String. The only function is the attribute function title on Course. Hence, the simple specification given in Figure 3 is derived.

1. Alternatively, we can represent an attribute as a value of the sort of finite maps, e.g. title : Map[Course, String]

Object-Type-Course: **trait**
includes
Type-String
introduces
title: Course → String

Figure 3: specification of object type Course with attribute title

The trait Type-String specifies the sort of strings String, which is available in the Larch HandBook of specification modules [11]. Any constraints on the attribute title are expressed as axioms on the function title in the *asserts* section of the trait. In a very similar way we interpret the other object types for the course administration system.

3.2 Associations

We now extend the interpretation of object types and attributes given in the previous section to include binary associations. We interpret associations between object types as two related mappings that map an object of one type to the set of associated objects of another (or the same) type. These mappings are specified in a way that is independent of the structure of types they associate. Thus we have a generic Larch theory for associations that can be renamed to specify each particular association in the model.

The many-many association between Instructor and Course (Figure 1) has two role names qualified and qualifiedFor. Intuitively, the role name qualifiedFor of the association is a mapping that maps an object i of type Instructor to a set of objects of type Course that are associated with i. In some navigation expressions the role name qualifiedFor is also used to map a set of instructors to a set of courses. In our formalisation, this association would be represented as two mappings qualified and qualifiedFor with the following signatures:

$$\texttt{qualified} : \texttt{Set[Course]} \rightarrow \texttt{Set[Instructor]}$$

$$\texttt{qualifiedFor} : \texttt{Set[Instructor]} \rightarrow \texttt{Set[Course]}$$

where Set[Course] and Set[Instructor] are the power sorts of Course and Instructor respectively. By choosing power sorts for the domains and ranges of these mappings, we have a uniform treatment of associations which simplifies the formalisation and provides generic theory for associations. The case where navigation is from a single object is subsumed with the general case where the set is a singleton containing that object. In addition, the corresponding mappings that map single objects can be defined in terms of those that map sets of objects (see later). A similar approach for formalising associations as two related mappings is presented in [10].

The two mappings qualified and qualifiedFor satisfy the axioms:

$$\texttt{qualified}(\{\}) == \{\}$$

$$\texttt{qualifiedFor}(\{\}) == \{\}$$

$$\texttt{qualified}(s \cup s') == \texttt{qualified}(s) \cup \texttt{qualified}(s')$$

$$\texttt{qualifiedFor}(s \cup s') == \texttt{qualifiedFor}(s) \cup \texttt{qualifiedFor}(s')$$

The operation $\cup$ is the union operation on sets, and $\{\}$ is the empty set. These axioms imply that these functions are completely determined by their values at singleton sets.

In order to represent the association, these functions must also be related. This relationship is expressed by the following axiom:

$$c \in \text{qualifiedFor}(\{i\}) == i \in \text{qualified}(\{c\})$$

Intuitively, this axiom asserts that if instructor `i` is qualified to teach course `c`, then `i` must be included in the set of instructors qualified to teach `c`.

The corresponding functions that operate on single objects may be constructed from those whose domains are power sorts as follows:

$$\text{qualified}(c) == \text{qualified}(\{c\})$$
$$\text{qualifiedFor}(i) == \text{qualifiedFor}(\{i\})$$

Semantically, navigating from a single object is equivalent to navigating from a singleton set containing that object. Note that we are overloading the function names which is allowed by LSL.[1]

A trait for the association between `Instructor` and `Course` is presented in Figure 4.

Association-`qualified-qualifiedFor`: **trait**

includes

Set(`Instructor`), Set(`Course`)

introduces

$\text{qualified} : \text{Set[Course]} \rightarrow \text{Set[Instructor]}$
$\text{qualified} : \text{Course} \rightarrow \text{Set[Instructor]}$
$\text{qualifiedFor} : \text{Set[Instructor]} \rightarrow \text{Set[Course]}$
$\text{qualifiedFor} : \text{Instructor} \rightarrow \text{Set[Course]}$

asserts

$\forall i$:Instructor, c:Course, s,s':Set[Course], t, t' : Set[Instructor]
$\text{qualified}(\{\}) == \{\}$
$\text{qualifiedFor}(\{\}) == \{\}$
$c \in \text{qualifiedFor}(\{i\}) == i \in \text{qualified}(\{c\})$
$\text{qualified}(c) == \text{qualified}(\{c\})$
$\text{qualifiedFor}(i) == \text{qualifiedFor}(\{i\})$

Figure 4: Specification of the association between `Instructor` and `Course`

Multiplicity constraints on roles can be interpreted quite easily in terms of the mappings that represent them by imposing a limit on the cardinality of the obtained sets. For further details and for the generic traits of types and associations the reader is referred to [12] and [13].

4 Interpreting Navigation Expressions

In this section we extend the object type, attributes, and association semantics given in the previous section to include navigation expressions. Navigation expressions are actually interpreted as terms (or expressions) over the signatures of object types and associations specifications.

4.1 Invariants

We start with simple expressions. The expression `c.title` represents the title of the course denoted by `c`. In our language this expression is interpreted as `title(c)` that is, the opera-

1. An alternative way is to define the functions that operate on sets in terms of those that operate of single elements as:
$\text{qualified}(\{\}) == \{\}$
$\text{qualified}(\text{insert}(c, s)) == \text{qualified}(c) \cup \text{qualified}(s)$
where $\{\}$ denotes the empty set, `insert(c,s)` denotes the set obtained by adding `c` to the set `s`.

tor "." is interpreted as the application operator. Parameterised attributes can also be interpreted in a similar way.

An expression with a role name such as i.qualifiedFor is interpreted as `qualifiedFor(i)` or equivalently `qualifiedFor({i})`. When a navigation expression yields another object or a set of objects, then it is possible to navigate on to their attributes. In the case where the result is a set of objects, any subsequent navigation must be applied to each member of the set and the result is a set of values constructed from the attributes of each of the objects located. For example, the expression i.qualifiedFor.title yields a set of strings. In this case the attribute title is applied to the set i.qualifiedFor (using the "." operator) and the result is a set whose members are the results of applying title to the members of i.qualifiedFor. It is clear that we cannot interpret this expression as `title(qualifiedFor(i))` since the attribute title is interpreted as a function that operate on single objects. What we need is to define a function that takes a set as argument and returns a set obtained by applying `title` to each member of the argument set. For this, we introduce the function $\texttt{mapSet}_{\texttt{title}}$ with the signature: $\texttt{mapSet}_{\texttt{title}} : \texttt{Set[Course]} \rightarrow \texttt{Set[String]}$ satisfying the axioms:

$$\texttt{mapSet}_{\texttt{title}}(\{\}) == \{\}$$
$$\texttt{mapSet}_{\texttt{title}}(\texttt{insert}(c, s)) == \texttt{insert}(\texttt{title}(c), \texttt{mapSet}_{\texttt{title}}(s))$$

Function symbols can be overloaded, so we can use `title` instead of $\texttt{mapSet}_{\texttt{title}}$. Now the expression i.qualifiedFor.title is interpreted simply as `title(qualifiedFor(i))`.

The expression i.qualifiedFor.presentations represents the set whose members are all the objects of Presentation which can be obtained by evaluating the expression i.qualifiedFor yielding a set of Course's objects, and then by navigating from each member of this set using presentations to obtain a set of sets of Presentation's objects. Taking the union of these sets we obtain the resulting set. This expression is interpreted as `presentations(qualifiedFor(i))`.

4.2 Filters

Navigation expressions of the form s[pred], where s is a set and pred is a boolean predicate, denotes the set of members in the set s for which the predicate pred is true. That is, the set s is filtered using the predicate pred. In Figure 1, the expression c.presentations is the set of presentations for a course c. The expression c.presentations[is_cancelled] is the set of all presentations of course c, for which is_cancelled is true, i.e. the set of cancelled presentations. Formally, the attribute is_cancelled is represented by the function: `is_cancelled : Presentation → Bool`. Now, we define a function `filter` with the signature: `filter : Set[Presentation] → Set[Presentation]` satisfying the axioms:

```
filter({}) = {}
filter(insert(p, s))= if is_cancelled(p) then insert(p, filter(s))
                      else filter(s)
```

Intuitively, `filter(s)` returns the elements of `s` that satisfy the predicate `is_cancelled`. This can be easily generalised to predicates with several arguments.

The navigation expression c.presentations[is_cancelled] is interpreted as `filter(presentations(c))`. The expression c.presentations[is_cancelled].students

represents the set of students associated with the cancelled presentations for course c and is interpreted as students(filter(presentations(c))), where students is the interpretation of the default role name students.

4.3 Pre and Post Conditions

So far we have interpreted object types, associations and navigation expressions statically. However, when specifying actions or operations on an object type it is necessary to refer to the values of an attribute (say) before and after the action is executed. In some modelling notations $\overline{f}$, old(f) are used to refer to the value of attribute f before the action is executed and f is used to refer to the value after the execution. In our formalisation, we introduce a sort Σ to represent the system states. Attributes of a given object type are interpreted as functions with additional argument for the system states. For example, the attribute title is now interpreted as a function title with the signature: $\texttt{title} : \texttt{Course}, \Sigma \rightarrow \texttt{String}$ where the expression $\texttt{title}(c, \sigma)$ is the title of the course c in the state σ. If σ and σ' are the states before and after an action is executed respectively, then $\texttt{title}(c, \sigma)$ and $\texttt{title}(c, \sigma')$ is the title of course c before and after this action is executed respectively.

Associations are also interpreted as two related functions as in the previous section, with an additional argument for the system state. The trait that specifies the association between Instructor and Course is given in Figure 5.

Association-qualified-qualifiedFor: **trait**

includes

Set(Instructor), Set(Course)

introduces

$\texttt{qualified} : \texttt{Set[Course]}, \Sigma \rightarrow \texttt{Set[Instructor]}$
$\texttt{qualified} : \texttt{Course}, \Sigma \rightarrow \texttt{Set[Instructor]}$
$\texttt{qualifiedFor} : \texttt{Set[Instructor]}, \Sigma \rightarrow \texttt{Set[Course]}$
$\texttt{qualifiedFor} : \texttt{Instructor}, \Sigma \rightarrow \texttt{Set[Course]}$

asserts

$\forall \texttt{i}:\texttt{Instructor}, \texttt{c}:\texttt{Course}, \texttt{s},\texttt{s}':\texttt{Set[Course]}, \texttt{t}, \texttt{t}' : \texttt{Set[Instructor]}, \sigma:\Sigma$
$\texttt{qualified}(\{\}, \sigma) == \{\}$
$\texttt{qualifiedFor}(\{\}, \sigma) == \{\}$
$\texttt{qualified}(s \cup s', \sigma) == \texttt{qualified}(s, \sigma) \cup \texttt{qualified}(s', \sigma)$
$\texttt{qualifiedFor}(t \cup t', \sigma) == \texttt{qualifiedFor}(t, \sigma) \cup \texttt{qualifiedFor}(t', \sigma)$
$c \in \texttt{qualifiedFor}(\{i\}, \sigma) == i \in \texttt{qualified}(\{c\}, \sigma)$
$\texttt{qualified}(c, \sigma) == \texttt{qualified}(\{c\}, \sigma)$
$\texttt{qualifiedFor}(i, \sigma) == \texttt{qualifiedFor}(\{i\}, \sigma)$

Figure 5: Specification of the association between Instructor and Course

The interpretation of navigation expressions in this case is very similar to the interpretation given in the previous section where a new state parameter is added to the functions interpreting navigation expressions. For example, Figure 6 gives the specification of an action assignInstructor in terms of *pre*/*post* conditions. The navigation expressions in the *pre* and *post* conditions are interpreted as the following expressions respectively:

$$(\texttt{instructor}(p, \sigma) = \{\}) \wedge (\texttt{course}(p, \sigma) \subseteq \texttt{qualifiedFor}(i, \sigma))$$

$$\texttt{instructor}(p, \sigma') = \{i\}$$

where σ and σ' are the states before and after the action is executed respectively.

```
assignInstructor(p : Presentation, i : Instructor)-- assigns instructor 'i' to give presentation 'p'
    pre:    p.instructor = nil                    -- an instructor is not already assigned
          ∧ p.course ∈ i.qualifiedFor             -- the instructor is qualified to give the course
    post:   p.instructor = i                      -- 'i' is assigned to give presentation 'p'
```

Figure 6: A partial specification of action assignInstructor

4.4 Navigation Expressions Involving Bags and Sequences

In some cases s.f, where s is a set and f is an attribute, could be interpreted as a bag (rather than a set) of values (e.g., as in Syntropy). To use the same notation could lead to ambiguity. For this reason we propose to use the notation $s._{bag}f$ for expressions with bags as results. To interpret such expressions we define a mapping $mapSet_f$ that maps sets into bags using the function representing the attribute f. This mapping has the following signature $mapSet_f : Set[A] \rightarrow Bag[A]$, where Bag[A] is the sort of bags of A's elements, and satisfies the following axioms:

$$mapSet_f(\{\}) == nil$$
$$mapSet_f(insert(e, s)) == insert(f(e), mapSet_f(s))$$

where nil represents the empty bag and the function f represents the attribute f. The expression $s._{bag}f$ is interpreted as $mapSet_f(s)$ where s interprets the set s. In a similar way we interpret expressions $s._{seq}f$ yielding sequences as results.[1]

In most cases navigating an association yields a set. However, some associations have constraints that specify the order of objects obtained. For instance, we can use constraints to specify a sequence, a bag or a sort order.

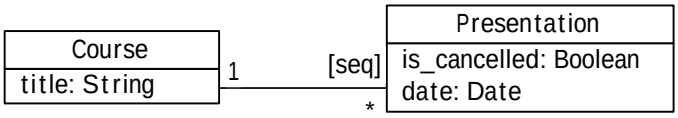

Figure 7: Association with sequence constraint

Figure 7 shows a modified association between the types Course and Presentation with a constraint that specifies a sequence indicated by the annotation [seq]. This means that navigating from an object of type Course via the association yields a sequence of objects of type Presentation. Formally, we interpret the role name presentations as a function with the signature[2]: $presentations : Course \rightarrow Seq[Presentation]$, where Seq[Presentation] is the sort of sequences with elements from the sort Presentation.

The relationship between the two functions course and presentations is expressed by the following axiom:

$$p \in presentations(c) == c \in course(p)$$

where the $\in$ used on the left of the equation is the membership function on sequences.

The expression c.presentations.is_cancelled is the sequence of results obtained by applying the attribute is_cancelled to every member of the sequence c.presentations. To

1. An alternative approach would be to interpret expressions of the form s.f as yielding sequences, and then using appropriate coercion functions to obtain sets or bags as results.
2. An alternative way is to represent sequences as totally ordered sets

interpret this expression, we define the function `mapSeq` with the signature: `mapSeq : Seq[Presentation] → Seq[Bool]` and satisfying the axioms:

```
mapSeq(nil) == nil
mapSeq(insert(p, s)) == insert(is_cancelled(p), mapSeq(s))
```

where `nil` denotes the empty sequence and `insert(p, s)` is the sequence obtained by adding the element `p` to the sequence `s`.

Associations with constraints that specify bags are interpreted in a similar way.

4.5 Subtyping and Navigation

Subtyping is a special relationship between two object types, known as the *is-a* relationship. An assertion that type B is a subtype of type A implies that objects that conform to type B inherit all the attributes and associations of the supertype A, and can be used in contexts where objects of type A are expected. Navigation expressions involving subtypes are interpreted in exactly the same way we interpret expressions involving supertypes. To do this we must provide an interpretation of the subtype, its attributes and associations. See [12] for details.

5 The Denotation Mappings

In interpreting the type diagrams of a model we associate with any object type name A a sort in Larch of all possible object identities that conform to the type, denoted by `A`. Let `Type` be the set of types given in a type diagram of a modelling notation. We assume that the set `Type` includes object types (*mutable*) and value types (*immutable*). Let `Sort` be the set of sorts in LSL. We define a mapping that associates with every type in the modelling notation a sort in Larch: `sort : Type → Sort` by `sort(A)` $=_{def}$ `A`. That is, `sort(A)` is the sort associated with the object type A. For example, we have `sort(Course)` $=_{def}$ `Course`, `sort(Instructor)` $=_{def}$ `Instructor`, `sort(Boolean)` $=_{def}$ `Bool`, etc..

For each attribute `f` of type `T` we associate with it a function symbol denoted by `f`. Let `Attributes` be the set of attributes of an object type A, and let `Functions` be the set of function symbols in LSL, then we define: `Fun : Attributes → Functions` where `Fun(f:T)` $=_{def}$ `f : A, Σ → T`. If the type A has a parameterised attribute `g(S):T` of type `T`, then we have: `Fun(g(S):T)` $=_{def}$ `g : A, S, Σ → T`.

Let `Associations` be the set of association symbols in the type diagram, then for association roles we define a similar mapping: `Fun : Associations → Functions` where `Fun(r:set[B])` $=_{def}$ `r : set[A], Σ → set[B]`.

In a similar way we deal with parameterised association roles. The table in Figure 8 summarizes the above mappings.

Modelling Notation	**Description**	**LSL**
A	object type (mutable)	A (sort of object identities)
T	value type (immutable)	T (sort of values)
f : T	attribute of type T, for object type A	f : A, Σ → T
g(S) : T	parameterised attribute of type T, for object type A	g : A, S, Σ → T

Figure 8: Mappings of types, attributes and associations

Modelling Notation	Description	LSL
r : set[B]	association role with set result for object type A	$r : Set[A], \Sigma \rightarrow Set[B]$
r (S) : set[B]	parameterised association role with set result	$r : Set[A], S, \Sigma \rightarrow Set[B]$
r : B	association role with single result for type A	$r : Set[A], \Sigma \rightarrow Set[B]$
r (S) : B	parameterised association role with single result	$r : Set[A], S, \Sigma \rightarrow Set[B]$

Figure 8: Mappings of types, attributes and associations

We now define a mapping that maps navigation expressions to LSL expressions based on the above mappings. Let NExpressions the set of navigation expressions and LExpressions be the set of LSL expressions. We define $L : NExpressions \rightarrow LExpressions$. The definition of L is given in Figure 9. The interpretation of a navigation expression as given by L is given at a moment in time corresponding to a system state σ. In this definition variables are mapped into variables in LSL, and expressions of the form a.f are mapped to $f(a, \sigma)$. Expressions of the form s.f, where s is a set and f is an attribute, are mapped to $mapSet_f(s, \sigma)$, where s is the interpretation of the set expression s, i.e. L(s) = s, which satisfies the following axioms:

$$mapSet_f(\{\}, \sigma) == \{\}$$
$$mapSet_f(insert(a, s), \sigma) == insert(f(a, \sigma), mapSet_f(s, \sigma))$$

Expressions of the form s.r where r (r : Set[B]) is an association role are mapped to $r(s, \sigma)$, where s is the interpretation of the set expression s, i.e. L(s) = s. Note that in this case we obtain a set rather than a set of sets because of the interpretation of association roles is different from the interpretation of attributes. If on the other hand we treat association roles as attributes, then the expression $r(s, \sigma)$ yields a set of sets obtained by applying the function r to each element of the set s. However, in order to obtain the desired result namely a set of objects of type B, the obtained set of sets need to be flattened by taking the union of all its elements. Therefore, it is always necessary to distinguish between association roles and attributes when interpreting navigation expressions. In Catalysis [5], [6] where an association is considered to be a pair of related attributes, a notion of *flat* sets is introduced when navigating via association roles, which does not seem to be strictly necessary in our formalisation.

Filter expressions of the form s[p] where s is a set and p is a boolean predicate are mapped to $filter_p(s, \sigma)$, where s is the interpretation of the set expression s, i.e. L(s) = s, which satisfies the following axioms:

$$filter_p(\{\}, \sigma) == \{\}$$
$$filter_p(insert(p, s), \sigma) == \text{if } p(a, \sigma) \text{ then } insert(a, filter_p(s, \sigma)) \text{ else } filter_p(s, \sigma)$$

Other value expressions such as sets, boolean values, sequences, bags can be mapped directly since value types can be specified algebraically.

NExpressions	Description	LExpressions
a	variable	a (variable)
a.f	a variable, f attribute of type T	f(a,σ)

Figure 9: Definition of the mapping L

NExpressions	Description	LExpressions
s.f	s set, f attribute of type T	$mapSet_f(s,\sigma)$,where(L(s)= s)
a.r	a variable, r association role	r(a,σ)
s.r	s set, r association role	r(s,σ),where (L(s)= s)
s[p]	s set, p boolean predicate	$filter_p(s,\sigma)$,where(L(s)= s)

Figure 9: Definition of the mapping L

6 Conclusions

A semantics has been provided for navigation expressions, which are essential to making object-oriented modelling notations precise. The semantics is given as theories in the Larch Shared Language, and a systematic mapping from class diagrams, with accompanying textual language, to Larch expressions has been defined. The semantics covers the use of navigation expressions in invariants, pre and post conditions, and handles not only navigation across single objects but also navigation across collections of objects, including sets, sequences and bags. Filters within navigation expressions have also been considered.

This work provides the basis for a semantics for UML incorporating OCL. Since this paper was originally submitted, this work has been further developed towards a semantics for OCL [14]. This will pave the way for giving a semantics to state diagrams, which state diagrammatically some of what otherwise can be said using class diagrams and pre/post conditions, and sequence diagrams. These will involve extending the semantics to deal with navigation expressions used to refer to operation invocations (This is the way navigation expressions are used in programming, and is similar to those expressions whose final segment is an attribute.). The semantics can also be used as a basis for the semantics of "Constraint Diagrams" in [15] which allow invariants and pre/post conditions to be visualised diagrammatically.

In the short term, the main purpose of the semantics work is to clarify concepts and refine notation. In the medium to long term we are interested in using it to develop CASE tools which are able, for example, to check the integrity of models and check conformance between models. This needs specific semantic support, so for example, we are currently working out the semantics of refinement in the OO/UML setting, that is checking the conformance of design and implementation models against specification models; and working on a compositional semantics [13], to support assembly of components through their specifications.

Acknowledgements

We gratefully acknowledge the support of the UK EPSRC under grant number GR/K67304. We thank Richard Mitchell and Franco Civello for their helpful comments.

References

1. Allen, P.: Components and Objects. Available at http://www.selectst.com/download (1997)
2. Bicarregui, J., Lano, K., Maibaum, T.S.E.: Objects, Associations and Subsystems: a hierarchical approach to encapsulation. Proceedings of ECOOP'97, LNCS Series, Springer-Verlag (1997)

3. Bordeau, H., Cheng, B.: A Formal Semantics for Object Model Diagrams. IEEE Transactions on Software Engineering, Vol. 21, No. 10 (1995)
4. Cook, S., Daniels, J.: Designing Object Systems: Object-Oriented Modelling with Syntropy. Prentice Hall (1994)
5. D'Souza, D., Wills, A.: Extending Fusion: practical rigor and refinement. R. Malan et al., OO Development at Work, Prentice Hall (1996)
6. D'Souza, D., Wills, A.: Objects, Components and Frameworks with UML: The Catalysis Approach. Addison-Wesley, to appear 1998. Draft and other related material available at http://www.trireme.com/catalysis
7. Fowler, M., Scott, K.: UML Distilled. Addison-Wesley (1997)
8. France, R., Bruel, J., Larrondo-Petrie, M., Shroff., M.: Exploring The Semantics of UML Type Structures with Z. Proceedings of International Workshop on Formal Methods for Object-based Distributed Systems (FMOODS), Chapman and Hall (1997)
9. Burstall R., Goguen J.: Putting theories together to make specifications. In Reddy R. (ed.) Proc. IJCAI 77, (1977) 1045-1058.
10. Graham, I., Bischof, J., Henderson-Sellers, B.: Associations considered a bad thing. Journal of Object-Oriented Programming, SIGS Publications, February (1997)
11. Guttag, J., Horning, J.: Larch: Languages and Tools for Formal Specifications. Springer-Verlag (1993)
12. Hamie, A., Howse, J.: Interpreting Syntropy in Larch. Technical Report ITCM97/C1, University of Brighton (1997)
13. Hamie, A., Howse, J., Kent, S.: Compositional Semantics of Object-Oriented Modelling Notations. Evans, A. and Lano, K., Making Object-Oriented Methods more Rigorous, LNCS Series, Springer Verlag, to appear (1998)
14. Hamie, A., Kent, S., Howse J.: A Semantics for OCL, submitted to ECOOP98 (1997)
15. Kent, S.: Constraint Diagrams: Visualising Invariants in Object-Oriented Models. Procs. of OOPSLA97, ACM Press, to appear (1997)
16. Kent, S., Lauder, A.: Rigorous Techniques in Component-Based Development. Evans, A. and Lano, K.: Making Object-Oriented Methods more Rigorous, LNCS Series, Springer Verlag, to appear (1998)
17. Rumbaugh, J., Blaha, M., Premerali, W., Eddy, F, Lorensen, W.: Object-Oriented Modelling and Design. Prentice Hall (1991)
18. Short, K.: Component Based Development and Object Modeling. Available from http://www.cool.sterling.com/cbd (1997)
19. UML Consortium: The Unified Modeling Language Version 1.1. Available from http://www.rational.com (1997)