Modular Semantics for Object-Oriented Models

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

Abstract

This paper describes a formal, modular approach to the semantics of object-oriented models. These models are expressed in the UML notation augmented with the Object Constraint language (OCL), which provides a precise textual language for expressing assertions: invariant constraints and operation specifications. The approach to semantics we adopt could easily be applied to other, similar notation sets. An important aspect of our approach is to treat every component of a model, down to the level of individual diagram elements, as distinct semantic entities which, here, are theories in Larch. The semantics of a model or part of a model is then the composition of the semantic entities corresponding to the individual model elements. This leads to a highly modular approach, allowing reuse of generic semantic entities, resulting in an elegant, transparent semantics for individual models. The fine-grained modularity promises to support the extraction and manipulation of different "views" of a model. The composition of model elements supported by the semantics promises to underpin the development of systems from reusable components. Larch is used as it supports theory composition, ensures that the semantics is relatively accessible (essentially theories of FOPL), is well-defined, and is close to technologies most likely to provide automated tool support (including a toolset to support it directly).

1 Introduction

Modern object-oriented modelling notations, such as the Unified Modelling Language (UML) [28][14] are based on graphical notations for expressing a wide variety of concepts which are relevant to the presentation of software requirements. While these notations are intuitive and easy to understand by users, they are not generally given a precise semantics as part of their initial definition, although a number of researchers have taken up this challenge [3][4][6][12]. In addition, the popular graphical notations can not express all the constraints that it is desirable to express. To remedy this, a number of authors have proposed mathematically-based textual languages, as an adjunct to the diagrams. Syntropy [8] extends OMT [25] with a Z-like textual language for adding invariants to class diagrams and annotating transitions on state diagrams with pre/post conditions. Catalysis [10][11] does something very similar for UML. Recently, the Object Constraint Language (OCL) [27][29] was developed as a part of the UML standard, and is being used for precisely expressing constraints on a model. Kent [23][23] defines a diagrammatic notation, compatible with UML and these textual languages, which allows invariants and pre/post conditions to be visualised.

Some argue that precision, either in semantics or textual annotations, is unnecessary. For bespoke software development this argument may be sustainable. 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. So why spend a lot of time making these models precise if they are only going to be thrown away? However, the software industry is now moving towards so-called component-based development (CBD), and here the requirement for precision cannot be so lightly discarded. Object-oriented modelling notations are being proposed [1][26] 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 for CBD, both diagrams and text, is elaborated in Short [26]. 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 exam-

ple, 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.

Semantics work [6][12] for OO models in widespread use, such as OMT or UML, is generally restricted to capturing the meaning of the graphical notation used, so accompanying precise textual languages have yet to be considered, as they are only now becoming part of those notations in the form of the OCL. These approaches don't consider generic theories for the diagrammatic notations.

Exceptions to this are the work of Bicarregui et al. [3][4] and our own work [16][17][21][18]. The former uses the Object Calculus [13] to develop a semantics for Syntropy [8], where object types and associations are encoded as theories in first-order temporal logic. They use categorical constructions (morphisms and co-limit) to compose their theories to yield a formalisation of the entire system. Their semantics is quite coarse-grained (e.g. one theory per type, one theory per association). However, their formalisation of Syntropy has indicated some areas where the semantics of the notations are non-modular. This led to the introduction of the subsystem concept. No doubt their work could be adapted to UML and its extensions, but this remains to be done.

In this paper we have chosen The Larch Shared Language (LSL) [15] to express the semantics of OO models. This choice is motivated in part 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. The simplicity of the syntax and semantics of Larch and being based on first order predicate logic, makes it so accessible to a wider audience. It is also close to technologies most likely to leverage the sophisticated CASE tools that should result from increasing the precision and expressiveness of modelling notations. This is illustrated by the inclusion of an automated proof assistant in its accompanying toolset. We have also been using our Larch-based semantics as a starting and subsequent reference point for developing checking and animation tools in Prolog.

Essentially we propose that the semantics of a model be constructed by composing Larch theories, corresponding to individual elements of a model. The granularity of composition is quite fine-grained: it goes beyond the level of individual diagrams, to the level of individual diagram elements, such as associations and invariants. This highly modular, fine-grained approach in contrast to Bicarregui et al. [3][4], allows model elements to be extracted and recomposed, thereby supporting construction of alternative views of a model and "safe" transformation of models (e.g. by reorganising an inheritance hierarchy). It also allows the semantics of a model to be built from pre-defined generic Larch traits (theories). Not only does this lead to a high degree of elegance and transparency in the semantics, any results proved about a generic trait or combination of traits will carry forward to models whose semantics has been built using them.

There is a natural progression from using this approach to build semantics of individual models, to using it to compose models into larger models. This is what is required to support component-based development, where components are specialised and composed to build other components and, eventually, systems. It is our aim to apply our approach to defining the semantics of component composition, as proposed in Catalysis [10][11], where generic Larch traits will now correspond to reusable, specializable components.

The paper is organised as follows. Section 2 describes the static part of specification model (class diagrams, invariants, states) for a small case study (a library system). Section 3 builds the compositional semantics for this part. Sections 4 & 5 describes the dynamic part of model (pre/post conditions and state transitions) and its semantics. Section 6 provides a summary and outlines future work.

2 Library System Specification: Static Aspects

This section gives the specification of a small case study, part of a library system, using a kernel of the diagrammatic notation of UML (class diagrams and simplified state diagrams) supplemented with OCL expressions to express those constraints that can not be expressed using diagrams from UML (or, indeed, other OO modelling notations). UML has been chosen as it incorporates and unifies many of the notations of its predecessors, so may be viewed as representative of them, and is fast becoming the de facto standard in OO modelling.

2.1 Informal Requirements

The general requirements are to produce a computerized system to support the management of loans in a university library. A library maintains a catalog of publications which are available for lending to users. There may be many copies of the same publication. Publications and copies may be added to and removed from the library. Copies available for lending may be borrowed by active users registered with the library. When a publication (or more specifically a particular copy) has been borrowed it is on loan, and is not available for lending to other users. However, it still belongs to the library and so is still part of its collection. Users are able to reserve publications, when none of the copies are available for loan. A user may not place more than one reservation for the same publication. When a copy is returned after it has been out on loan, it may be put back on the shelf or, alternatively, held for a user who has reserved the publication of which it is a copy. This may be done immediately on return, or delayed, and done as part of a batch of returned copies.

2.2 Type (Class) Diagram

At the specification level, "classes" in a class diagram should really be thought of as object interfaces or object types [14]. For the remainder of this paper we will therefore refer to *object types* rather than classes, and to type diagram rather than class diagram.

The main type diagram for the library system is given in Figure 1. The UML notation uses boxes to depict the types of objects that can exist in the system. Each object type has a name, a set of attributes, and a list of operations (optional). Associations between object types are depicted as lines between the type boxes with various annotations. For the library system, there are six object types: Library, Publication, User, Copy, Reservation and Loan. A publication is a record of all the details of a book: title, authors, ISBN, etc. A publication may have many copies (or none), which correspond to the physical books. A copy only has one set of publication details. A copy may be availableTo loan to many users, and a user may have many copies which are available for loan to her. All the publications known to the library are in the catalog, all the copies form the library's collection, and all the users known to the library must be registered.

Loan and Reservation characterize objects used to record loans and reservations, respectively. A loan records which copy is on loan to which user, and a reservation records which publication has been reserved by which user. A copy may be put onHoldFor a reservation, which means that it is waiting to be collected by the user who made the reservation. After the copy has been collected the reservation will have been fulfilled.

Some associations in the diagram have been given rolenames at one or both ends. If the association is unlabeled the name of the object type at either end is used, with the first letter in lowercase: for example we can refer to the loan of a user or the user associated with a loan.

2.3 Invariants

The type diagram on its own can not express all the constraints that we would wish to express for the library system. In particular, cardinality restrictions do not allow relationships between associations to be expressed. For example, it should be a constraint that the publication of a copy onHoldFor a reservation is the one that has been reserved. In OCL this constraint can be specified as follows:

```
Copy.allInstances, Publication.allInstances, Reservation.allInstances->forAll(c : copy,
                    p : Publication, r : Reservation |
                    p.copy->includes(c) and c = r.heldCopy implies r.publication=p)
```

where allInstances is type feature returning the set of all existing objects of a type at any point in time. For example, Copy.allInstances is the set of all existing copies in the library. The expression p.copy is the set of all copies of publication p, r.heldCopy represents the copy held for reservation r, and r.publication is the publication object associated with reservation r. The logical operators in OCL are represented by and, or, not and implies. The operation forAll is used to assert that every element of a collection taken as a parameter (set, bag, sequence) satisfy a bool-

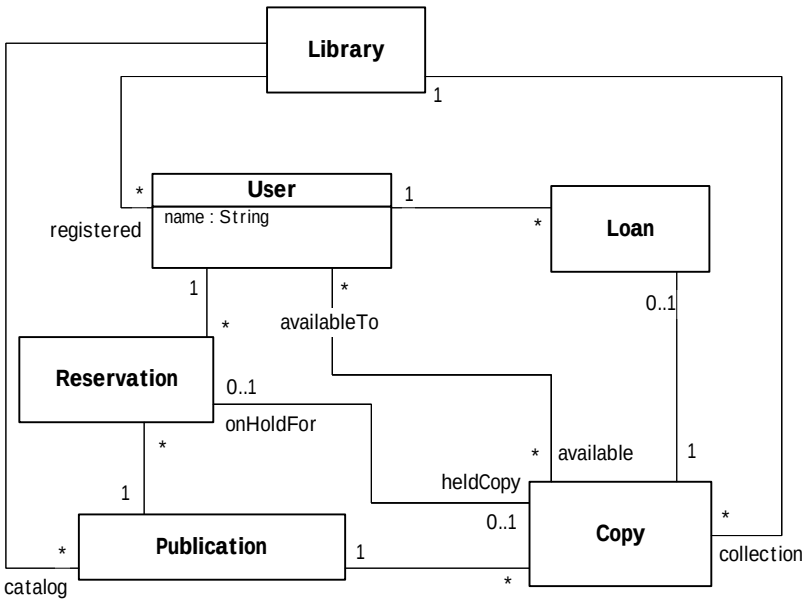

Figure 1: Type (class) diagram for library system

ean predicate. In here we used the expression (A.allInstances, B.allInstances)->forAll(a:A, b:B| pred(a,b)) as an abbreviation for A.allInstances->forAll(a:A | B.allInstances->forAll(b:B | pred(a,b)) where A and B are object types and pred is a boolean predicate involving a and b.

3 Semantics: Static Aspects

In this section we provide an LSL formalisation of type diagrams and invariants. Firstly, we provide the semantics of object types as theories of first order predicate logic expressed in Larch. Then we show how object type theories are combined to interpret associations between object types, and how theories of associations can be extended to model any cardinality and lifetime dependencies of the particular association. These theories are then combined in order to provide the formal semantics of the whole type diagram.

In our formalisation, a type diagram characterises a set of object diagrams which are consistent with the properties and constraints expressed by the graphical notation and textual invariants. An object diagram can be regarded as a snapshot of a system's structure at some point in time where the objects are those that have been created but not yet destroyed. We shall illustrate the rules for transforming type diagrams and invariants to LSL structured specifications that characterise object diagrams.

3.1 The Larch Shared Language

The *Larch Shared language* (LSL) [15] is a specification language based on many-sorted first order predicate logic with equality. Larch was chosen as the target language because it has a simple syntax and semantics and there are a number of tools that can be used to manipulate the specifications, including a syntax checker and the Larch (theorem) prover. 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. Following the name, is the list of *sorts* 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). In Larch overloading of function identifiers is allowed. 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 $''\wedge''$, $''\vee''$, $''\Rightarrow''$ and $''\neg''$.

3.2 Semantics of Object Types

In our formalisation, an object type is associated with an LSL basic sort consisting of elements that uniquely represent objects of the type, which can be thought of as object identifiers. That is, object types are characterised (interpreted) as sorts of *object identities*. At any point in time only a finite collection of a sort representing an object type will identify currently existing objects of this type. For example, the object type User of the library system is interpreted as a sort denoted by `User`. In order to record the set of existing objects of object type User, we introduce a function $\overline{\texttt{User}}$ with the signature $\overline{\texttt{User}}: \Sigma \rightarrow \texttt{Set[User]}$. The sort Σ is the sort of system states and the sort `Set[User]` is the sort of finite sets of elements of sort `User`. $\overline{\texttt{User}}(\sigma)$ is the finite set of existing users in the state σ. This expression actually interprets User.allInstances at a point in time.

The attributes of object types are interpreted as functions on the representing sort with the appropriate signatures. For example, the attribute name (user name) of (value) type String (strings) for the object type User is interpreted as the function `name` with the signature $\texttt{name} : \texttt{User}, \Sigma \rightarrow \texttt{String}$, which is added to the specification for object type User. The expression $\texttt{name}(u, \sigma)$ denotes the value of the attribute name for user u in the state σ.

According to the above description, the specification of the object type User has only two sorts: `User` and `String`. The only functions are $\overline{\texttt{User}}$ and the attribute function `name` on `User`. Hence, the simple specification given in Figure 2 is derived.

Object-Type-User: **trait**
includes
Type-String, Set(User)
introduces
$\overline{\texttt{User}} : \Sigma \rightarrow \texttt{Set[User]}$
$\texttt{name}: \texttt{User}, \Sigma \rightarrow \texttt{String}$

Figure 2: Specification of type User

The trait Set(User) specifies the power sort of User, and Type-String specifies the sort of strings String. Any constraints on the attribute name are translated to constraints on the function name and included in the *asserts* section of the trait Object-Type-User.

In a similar way we interpret other types in the library system. In general, given an object type A we construct a generic trait Basic-Object-Type(A, $\overline{A}$) that includes the function $\overline{A}$ which returns the set of existing object of type A in a given system state σ. This trait is shown in Figure 3 which can be used in specifying object types by renaming A and $\overline{A}$. The purpose of this trait is to manage the set of existing objects in a given state. In addition we construct a trait that specifies only the attributes of a given type Attributes-A. This trait is not generic and is specific for an object type A. The specification of object type A is a trait obtained by including the traits Basic-Object-Type(A, $\overline{A}$) and Attributes-A.

Basic-Object-Type (A, $\overline{A}$): **trait**
includes
Set(A)
introduces
$\overline{A} : \Sigma \rightarrow Set[A]$

Figure 3: Basic specification of type A

3.3 Semantics of Associations

We now extend the semantics of types and attributes given in the previous section to include binary associations. We interpret associations between 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 association between User and Loan (Figure 1) has two default role names user and loan. Intuitively, the role name user of the association is a mapping that maps an object l of type Loan to an object of type User that is associated with l. The role name loan is a mapping that maps a user object u to a set of loans associated with that user. In some situations the role name user is also used to map a set of loans to a set of associated users. This association would be represented as two mappings user and loan with the signatures:

$user : Set[Loans], \Sigma \rightarrow Set[User]$

$loan : Set[User], \Sigma \rightarrow Set[Loan]$

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).

The two mappings user and loan satisfy the axioms:

$user(\{\}, \sigma) == \{\}$

$loan(\{\}, \sigma) == \{\}$

$user(s \cup s', \sigma) == user(s, \sigma) \cup user(s', \sigma)$

$loan(s \cup s', \sigma) == loan(s, \sigma) \cup loan(s', \sigma)$

The operation $\cup$ is the union operation on sets. Note, that 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 axiom:

$$u \in user(\{l\}, \sigma) = l \in loan(\{u\}, \sigma)$$

Intuitively, this axiom asserts that if user u has a loan l, then l must be a member of the set of loans for user u.

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

$$user(l, \sigma) == user(\{l\}, \sigma)$$

$$loan(u, \sigma) == loan(\{u\}, \sigma)$$

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]

In addition, only existing objects can participate in an association. For the association between User and Loan we express this constraint by the following axioms:

$$l \in loan(u, \sigma) \Rightarrow l \in \overline{Loan}(\sigma) \wedge u \in \overline{User}(\sigma)$$

$$u \in user(l, \sigma) \Rightarrow l \in \overline{Loan}(\sigma) \wedge u \in \overline{User}(\sigma)$$

Temporarily omitting the specification for cardinality constraints on the association, Figure 4 gives a specification for the association between User and Loan. The traits Set(User) and Set(Loan) specify the power sorts of User and Loan respectively. A generic trait *BasicAssociation*(A, B, l, r) that specifies a generic association between object types A and B with role names l (at the left end) and r (at the right end) can be constructed easily. Using this trait we build another generic trait *Association*(A, B, $\overline{A}$, $\overline{B}$, l, r) by including the two functions $\overline{A}$ and $\overline{B}$ together with the axioms relating them to the role names. By renaming the parameters A, B, $\overline{A}$, $\overline{B}$, l, and r of the trait we can specify any association in the model. So the trait in Figure 4 can be concisely presented as *Association*(User, Loan, $\overline{User}$, $\overline{Loan}$, user, loan).

Association-user-loan: **trait**

includes

Set(User), Set(Loan)

introduces

$user : Set[Loan], \Sigma \rightarrow Set[User]$
$user : Loan, \Sigma \rightarrow Set[User]$
$loans : Set[User], \Sigma \rightarrow Set[Loan]$
$loans : User, \Sigma \rightarrow Set[Loan]$

asserts

$\forall u{:}User, l{:}Loan, s, s'{:}Set[Loan], t, t' : Set[User]$
$user(s \cup s', \sigma) == user(s, \sigma) \cup user(s', \sigma)$
$loan(t \cup t', \sigma) == loan(t, \sigma) \cup loan(t', \sigma)$
$user(l, \sigma) == user(\{l\}, \sigma)$
$loan(u, \sigma) == loan(\{u\}, \sigma)$
$l \in loan(u, \sigma) \Rightarrow l \in \overline{Loan}(\sigma) \wedge u \in \overline{User}(\sigma)$
$u \in user(l, \sigma) \Rightarrow l \in \overline{Loan}(\sigma) \wedge u \in \overline{User}(\sigma)$

Figure 4: Specification of the association between User and Loan, omitting multiplicities.

1. An alternative way is to define the functions that operate on sets in terms of those that operate of single elements as:

$$user(\{\}, \sigma) == \{\}$$
$$user(insert(u, s), \sigma) == user(u, \sigma) \cup user(s, \sigma)$$

where {} is the empty set, insert(u, s) is the set obtained by adding u to set s, $\cup$ denotes the union operation.

Having shown how the type and association constructs of the modelling notation are mapped to Larch, the diagram consisting of the object types User, Loan and the association between them is partially specified by a trait that includes the traits Object-Type-User and Object-Type-Loan that specify the types User and Loan respectively, and the association trait. This trait is given in Figure 5. This illustrates the compositional aspects of the semantics, where

Association-User-Loan-user-loan: **trait**
  includes
    Object-Type-User,
    Object-Type-Loan,
    Association(User, Loan, $\overline{User}$, $\overline{Loan}$, user, loan)

Figure 5:The Specification of the diagram User, Loan and the association between them

a diagram specification is obtained by composing the specifications of its parts.

3.4 Cardinality Constraints

Cardinality constraints of an association constrain the number of objects that can be associated with an object of a given type. A range cardinality has the form m..n , where m is the lower bound and n is the upper bound of the range. The range m..m can be written m. The cardinality symbol * indicates many, i.e. an unlimited number of objects. This symbol is equivalent to 0. .* i.e. zero ore more. These constraints can be expressed by further constraining the functions that represent the association.

One-many(A, B, $\overline{A}$, $\overline{B}$, l, r) : **trait**
  includes
    Association(A, B, $\overline{A}$, $\overline{B}$, l, r)
  asserts
    $\forall b : B, \sigma:\Sigma$
    $b \in \overline{B}(\sigma) \Rightarrow size(l(b, \sigma)) = 1$

Figure 6: One-many association specification

One-many Associations

The association line of the association between object types User and Loan is annotated with 1 on the left, that is each existing object of type Loan is associated with exactly one object of type User, in this case the map is total on the set $\overline{Loan}(\sigma)$. This constraint is expressed by the following axiom on the function user:

$$l \in \overline{Loan}(\sigma) \Rightarrow size(user(l, \sigma)) = 1$$

where size returns the cardinality of its set argument. A trait that specifies a one-many constraint can be constructed by extending the basic trait for the association by adding the above axiom. A generic trait for specifying a one-many association is shown in Figure 6. In this case the one-many constraint on the association between User and Loan is presented as One-many(User,Loan,$\overline{User}$,$\overline{Loan}$,user,loan)

In the many-one case a similar axiom is imposed on the function representing the role name at the right end of the association line. A similar generic trait can also be constructed for the many-one case.

One-one Associations

An association is one-one if its line is annotated by 1 at both ends. That is, each object of the type at the left end of the association line is associated with exactly one object of the type at the other end and vice versa. In this case we have both the set of axioms for many-one and one-many cases. A generic trait for this case is shown in.

One-one(A, B, $\overline{A}$, $\overline{B}$, l, r) : **trait**
includes
One-many(A, B, $\overline{A}$, $\overline{B}$, l, r),
Many-one(A, B, $\overline{A}$, $\overline{B}$, l, r)

Figure 7: One-one association specification

Optional Associations

The association between the object types Reservation and Copy is annotated by 0..1 at both ends. This means that the association is optional at both ends in the sense that an object of type Reservation can only be associated with at most one object of type Copy and vice versa. This is expressed by the following axioms:

$$(\forall c{:}Copy)(size(onHoldFor(c, \sigma)) \leq 1)$$

$$(\forall r{:}Reservation)(size(heldCopy(r, \sigma)) \leq 1)$$

where the functions onHoldFor and heldCopy represent the role names onHoldFor and heldCopy respectively.

A generic trait that specify the constraints of an optional association is shown in Figure 8. These can be (re)used to specify associations that are optional at both ends by suitable renaming.

Optional(A, B, $\overline{A}$, $\overline{B}$, l, r) : **trait**
includes
Association(A, B, $\overline{A}$, $\overline{B}$, l, r)
asserts
$\forall a : A, b : B, \sigma{:}\Sigma$
$(a \in \overline{A}(\sigma) \Rightarrow size(r(a, \sigma))) \leq 1$
$(b \in \overline{B}(\sigma) \Rightarrow size(l(b, \sigma))) \leq 1$

Figure 8: Optional association specification

3.5 Subtyping

Subtyping is a special relationship between two object types, known as the *is-a* relationship. An assertion that object type B is a subtype of object type A implies that objects that conform to object type B inherit all the attributes and associations of the supertype A, and can be used in contexts where objects of object type A are expected. The use of the word ‘‘ inherit’’ here is related to but should not be confused with class inheritance found in object-oriented programming. Subtyping is defined to imply type conformance in the sense that an object conforming to B always conforms to supertype A.

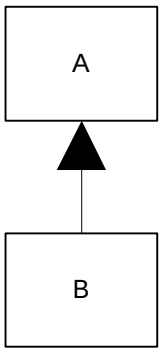

Figure 9: Subtyping

The notation used to assert that object type B is a subtype of A is shown in Figure 9. The notion of subtyping can be formalised by requiring that the subtype is represented by a sort which is a subsort of the sort representing the supertype. However, because Larch does not support a notion of subsorting, we use a function that maps between the corresponding sorts. That is, we introduce a function simulates with the following signature:

$$simulates : B \rightarrow A$$

Intuitively, the role of simulates is to map an object identifier of type B to the corresponding object identifier in A that behaves like it.

Subtyping can be divided into *static* and *dynamic* subtyping. Static subtypes are those that have objects that don't migrate into other subtypes during their lifetimes. On the other hand dynamic subtypes have objects that migrate between different subtypes of the same supertype. In fact, dynamic subtypes represent object states of their supertypes, where objects move from one state to another as a result of transitions. In Syntropy (Cook and Daniels, 1994), dynamic subtypes are referred to as *state* types. For example, Figure 10 shows the type User with two dynamic subtypes Active and Inactive. The filled in subtype arrow indicates partitioning: the object type User is partitioned between those objects in the state Active and those in the state Inactive. Subtyping is used to bring states into type diagrams, based on the semantic intuition that every Active object is a User object, but not vice-versa. Furthermore, in the true spirit of subtyping, states can be further constrained, e.g., they can have associations that objects not in that state do not have and may further constraint multiplicities on associations obtained from the supertype/superstate.

In an attempt to unify notation, we have chosen to use a box with rounded corners to represent a state, i.e. the same shape of box that is used state diagrams. The aforementioned methods generally use .

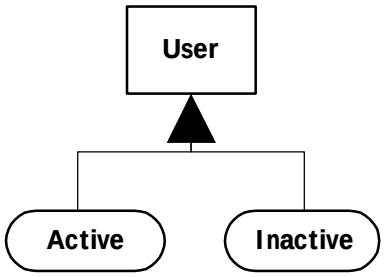

Figure 10: states of object type User

If B is a subtype of A then each existing object of B is associated with exactly one object of A through the function simulates, and each object of A with zero or one object of B. That is, simulates must be a total function. However, this property is ensured by the semantics of Larch which is based on total algebras, so all functions are assumed to be total.

For dynamic subtyping, the trait in Figure 11 captures this information. This trait introduces two functions simulates and memberB, where memberB returns true if an object of A is also an object of B in a given system state and false otherwise. The axiom for simulates asserts that the lifetimes of objects of B are subsumed within the lifetimes of their associated objects. While the other axiom defines the function memberB.

Basic-Subtyping-Dynamic(A, B, $\overline{A}$, $\overline{B}$, memberB) : **trait**

includes

Basic-Object-Type(A, $\overline{A}$), Basic-Object-Type(B, $\overline{B}$)

introduces

simulates : $B \rightarrow A$
memberB : A, $\Sigma \rightarrow$ Bool

asserts

$\forall a : A, b:B, \sigma: \Sigma$
$b \in \overline{B}(\sigma) \Rightarrow$ simulates(b) $\in \overline{A}(\sigma)$
$a \in \overline{A}(\sigma) \Rightarrow$ (memberB(a, σ) $\Leftrightarrow$ ($\exists b:B$)($b \in \overline{B}(\sigma) \wedge$ simulates(b)=a)

Figure 11: Specification of Basic Dynamic Subtyping

For static subtyping, we construct a trait Basic-Subtyping-Static(A, B, $\overline{A}$, $\overline{B}$, memberB) by including the dynamic subtyping trait and including the following axiom:

$$(\forall b:B)(b \in \overline{B}(\sigma) \Leftarrow \text{simulates}(b) \in \overline{A}(\sigma))$$

which asserts together with the other axiom that the lifetime of A is exactly the same as the lifetime of the corresponding object of B.

Subtypes may or may not partition a supertype, and may or may not be pairwise disjoint. One possible type of constraints on subtypes is given by Figure 10 which shows two object states Active and Inactive for the type User. The black arrow indicates that the two states partition the whole type and they are disjoint. That is all users are either active or inactive, and there is no user that is both active and inactive. This type of constraints on subtypes can be specified by a generic trait Partition-Disjoint-Subtyping(A, B1, B2, $\overline{A}$, $\overline{B}$1, $\overline{B}$2, memberB1, memberB2) where B1

and B2 are the subtypes of type A. This trait can be generalised to any number of subtypes. Other types of constraints on subtypes of a given type can be specified in a similar way. The diagram of Figure 10 can be specified by the trait in Figure 12. The traits Dynamic-Subtype-User(Active, $\overline{\texttt{Active}}$, memberA) and Dynamic-Subtype-User(Inactive, $\overline{\texttt{Inactive}}$, memberI) specify that Active and Inactive are dynamic subtypes of type User respectively.

Diagram-User-Active-Inactive : **trait**
includes
Object-Type-User, Basic-Object-Type(Active, $\overline{\texttt{Active}}$),
Basic-Object-Type(Inactive, $\overline{\texttt{Inactive}}$),
Dynamic-Subtype-User(Active, $\overline{\texttt{Active}}$, memberA),
Dynamic-Subtype-User(Inactive, $\overline{\texttt{Inactive}}$, memberI),
Partition-Disjoint-Subtyping(User, Active, Inactive, $\overline{\texttt{User}}$, $\overline{\texttt{Active}}$, $\overline{\texttt{Inactive}}$, memberA, memberI)

Figure 12: Specification of diagram User, Active **and** Inactive

Attributes

The *substitution property* is the most important aspect of subtyping. It asserts that for any operation defined on the supertype A, objects of type B can be substituted where objects of type A are expected with the behaviour of the operation preserved. The operation simulates provides the tool to ensure the consistency of the inherited operations and constraints of the subtype with those of the supertype.

Attributes of the supertype are inherited by any subtype. If object type A has an attribute f of type T, then every subtype B of A must also have the same attribute. So we add the signature to the specification of B: $\texttt{f} : \texttt{B}, \Sigma \rightarrow \texttt{T}$. The function f is constrained by the following axiom:

$$\texttt{f}(\texttt{b}, \sigma) == \texttt{f}(\texttt{simulates}(\texttt{b}), \sigma)$$

where f in the right hand side of the equation has the signature $\texttt{f} : \texttt{A}, \Sigma \rightarrow \texttt{T}$.

To specify the inherited attributes of a subtype B we construct a trait InheritedAttributes-A(B) which includes the trait Attributes-A twice with one renamed copy together with the axioms relating the inherited attributes. This trait is shown in Figure 13 assuming there is only one attribute f for type A.

InheritedAttributes-A(B) : **trait**
includes
Attributes-A, Attributes-A[B for A]
introduces
$\texttt{simulates} : \texttt{B} \rightarrow \texttt{A}$
asserts
$(\forall \texttt{b} : \texttt{B}, \sigma : \Sigma)$
$\texttt{f}(\texttt{b}, \sigma) == \texttt{f}(\texttt{simulates}(\texttt{b}), \sigma)$

Figure 13: Specification of inherited attributes by subtype B **of type** A**.**

We can now construct a generic Dynamic-Subtype-A(B, $\overline{\texttt{B}}$, memberB) trait that specifies that B is a subtype of A. This is obtained by including the traits Basic-Subtyping-Dynamic(A, B, $\overline{\texttt{A}}$, $\overline{\texttt{B}}$, memberB) and InheritedAttributes-A(B). This trait is given in Figure 14. In a similar way we construct a trait for static subtyping Static-Subtype-A(B, $\overline{\texttt{B}}$, memberB).

Dynamic-Subtype-A(B, $\overline{B}$, memberB) : **trait**
includes
Basic-Subtyping-Dynamic(A, B, $\overline{A}$, $\overline{B}$, memberB),
InheritedAttributes-A(B)

Figure 14: Specification of B as a dynamic subtype of type A.

Associations

Associations are also inherited by subtypes. That is, the associations in which an object type A is referenced must also be applicable when objects of A are substituted for objects of a subtype B of A. For each association role involving object type A, we induce the same association role on every subtype B of A, with B substituted for A. For instance, Figure 15 shows an association between objects of types A and C with role names l and r. The function that represents the role name r has the signature: $r : Set[A], \Sigma \rightarrow Set[C]$.

In order to allow objects of B to be used in place of objects of A in the association given above, a new function is introduced to represent the inherited association role between B and C. Formally, we introduce a new function r with the signature: $r : Set[B], \Sigma \rightarrow Set[C]$.

The function r given above is distinct from the function r that represents the association role involving the supertype, but because of the subtyping relationship, the two functions are closely related. The semantics of r above is constrained by the semantics of the function r for the supertype as follows:

$$r(\{b\}, \sigma) == r(\{simulates(b)\}, \sigma)$$

That is, an object b of type B, is related to those objects of C to which it relates when viewed as an object of type A; this ensures consistency. In addition, this constraint imposes all of the constraints regarding the association role of the supertype on the inherited association role.

The cardinality constraint on the association role l asserts that for every object of C, there is at most one object of A associated via l. This constraint has to hold also for objects of A including those belonging to a subtype of A. This can be formalised by constraining the simulates function to be injective for every subtype B of A:

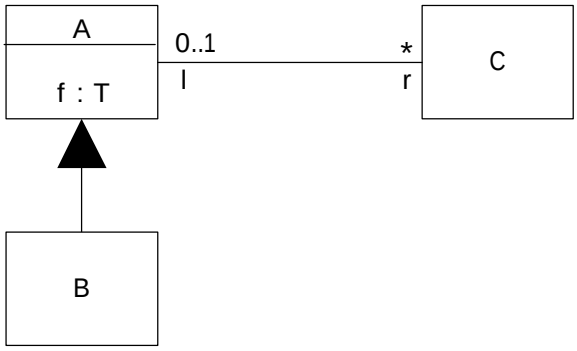

Figure 15: Association between object types A and C

$$simulates(b_1) = simulates(b_2) \Rightarrow b_1 = b_2$$

For associations we construct a trait InheritedRole(A, B, C, r) which specifies the inherited role r by subtype B of a type A, by introducing the two functions above with the axiom constraining them. Given that the specification of the diagram in Figure 15 is given by the trait shown in Figure 16. The trait Static-Subtype-A(B, $\overline{B}$, memberB) specifies that B is a subtype of A and it makes use of the trait InheritedAttributes-A(B).

Diagram-A-B-C : **trait**
includes
Object-Type-A,
Object-Type-B,
Object-Type-C,
Association(A,C, $\overline{A}$, $\overline{C}$, l, r), Static-Subtype-A(B, $\overline{B}$, memberB),
InheritedRole(A, B, C, r)

Figure 16: Specification of diagram in Figure 15

3.6 Semantics of Invariants

Invariants as written in OCL are basically boolean expressions expressed in terms of attributes and role names. Expressions within invariants may involve navigation expressions. In order to give a semantics of invariants it is necessary first to give a semantics of navigation expressions. In our formalisation, navigation expressions correspond to the language of terms built over the signatures of specification traits. However, in some cases navigation expressions may involve navigating through collections and filters which makes it necessary to define auxiliary functions to provide the semantics of these expressions. For details on the semantics of navigation expressions, the reader may consult [18] where we provided a semantics of navigation expressions covering the use of navigation expressions in invariants, filters, the use of navigation expressions in pre/post conditions, and navigating over collections such as sets, sequences, and bags. The notation we used is based on the textual language of Catalysis, which forms a subset of OCL with some syntactic variations. However, for some issues concerning the meaning of OCL expressions the reader is referred to [16].

In the library system, for example, the invariant in Section 2.3, *the publication associated with the held copy is the same as that reserved*, is translated to the following axiom:

$$\forall c:\mathtt{Copy},\ p:\mathtt{Publication},\ r:\mathtt{Reservation},\ \sigma:\Sigma$$
$$(c \in \mathtt{copies}(p,\sigma) \wedge \{c\} = \mathtt{heldCopy}(r,\sigma) \Rightarrow \mathtt{publication}(r,\sigma) = \{p\}))$$

where `publication` and `heldCopy` are the functions that represent the role names publication and heldCopy respectively.

A trait that specifies this invariant is built by including the specification of the involved association roles and the above axiom. This trait is presented in Figure 17. A generic trait can be easily constructed to specify this constraint pattern.

Invariant : **trait**

includes
BasicAssociation(`Publication`, `Copy`, `publication`, `copies`),
BasicAssociation(`Reservation`, `Copy`, `OnHoldFor`, `heldCopy`),
BasicAssociation(`Publication`, `Reservation`, `publication`, `reservations`)

asserts

$$\forall c:\mathtt{Copy},\ p:\mathtt{Publication},\ r:\mathtt{Reservation},\ \sigma:\Sigma$$
$$(c \in \mathtt{copies}(p,\sigma) \wedge \{c\} = \mathtt{heldCopy}(r,\sigma)) \Rightarrow$$
$$\mathtt{publication}(r,\sigma) = \{p\}$$

Figure 17: Invariant Specification

3.7 Putting it all together

We have shown so far that basic type diagram constructs can be given a precise formal semantics expressed in Larch, it remains only to describe the meaning of a static type model as a whole.

Applying the approach to formalisation described in this paper, the semantics of a type diagram is a formal specification expressed in LSL. The formal specification of a type diagram is a single trait, which is simply the inclusion of all object types, states, associations, and invariant traits derived from the type diagram. We illustrate this with the library system.

The library system consists of two views namely those shown in Figure 1 and Figure 10. The static model of library system is constructed by merging or composing these two views together. First, the view given in Figure 1 is specified by including the traits for the object types, associations, and their cardinality constraints in a single trait named Library-Types-Associations-Cardinality given in Figure 18. Then the view in Figure 10 is specified by including the traits for User, Active and Inactive in a single trait together and constraining the two states to be disjoint. This follows, by including the two traits specifying each view in a single trait to obtain the specification of the com-

Library-Types-Associations-Cardinality : **trait**
includes
Object-Type-Library, Object-Type-User, Object-Type-Publication,
Object-Type-Copy, Object-Type-Loan, Object-Type-Reservation,
Association(Library, User, Library, User, library, registered),
Association(Library, Publication, Library, Publication, library, catalog),
Association(Library, Copy, Library, Copy, library, collection),
Association(User, Copy, User, Copy, availableTo, available),
Association(User, Loan, User, Loan, user, loan),
Association(User, Reservation, User, Reservation, user, reservation),
Association(Publication, Copy, Publication, Copy, publication, copy),
Association(Publication, Reservation, Publication, Reservation, publication, reservation),
Association(Copy, Loan, Copy, Loan, HasLoan, copy, loan),
Association(Copy, Reservation, Copy, Reservation, copy, reservation),
One-many(User, Loan, Loan, user, loan),
One-many(User, Reservation, Reservation, user, reservation),
One-many(Publication, Copy, Copy, publication, copy),
One-many(Publication, Reservation, Reservation, publicatoion, reservation),
One-many(Copy, Loan, Loan, copy, loan),
One-many(Library, Copy, Copy, library, copy),
Optional(Copy, Reservation, copy, reservation)

Figure 18: Specification of types, associations and their cardinality constraints.

posite view. Note that in the obtained composite view, for each association involving the object type User we have to define the same (inherited) association for the states Active and Inactive. This is because Larch does not support subsorting. A trait that specifies the static model of library system is given in Figure 19.

Library-Static-Model : **trait**
includes
Library-Types-Associations-CardinalityC,
Invariant,
Diagram-User-Active-Inactive,
InheritedRole(User, Active, Copy, available),
InheritedRole(User, Active, Loan, loans),
InheritedRole(User, Active, Reservation, reservations),
InheritedRole(User, Active, Library, library),
InheritedRole(User, Inactive, Copy, available),
InheritedRole(User, Inactive, Loan, loans),
InheritedRole(User, Inactive, Reservation, reservations),
InheritedRole(User, Inactive, Library, library),
. . . .

Figure 19: Specification of static aspects of library system

4 Specification: Dynamic Aspects

The dynamic aspects of a system describes how the system moves from one valid state to another and indicates the conditions under which a state change may occur. These changes may occur as a result of invoking system actions or events. For the library system we shall only consider one action, namely, borrow. In OCL actions are specified precisely in terms of preconditions and postconditions. Preconditions specify the conditions under which the action can be invoked, while postconditions express the relationship between the states before and after the action is invoked. A specification for borrow may be written precisely as follows:

```
Library::borrow(u:User, c:Copy)
    pre
        u is registered and active, and c is available for lending to u
        registered->includes(u) and u.Active and
        c.availableTo->includes(u)
    post
        c is no longer available for lending,
        c.availableTo->empty
        and a new loan of c to u is recorded
        and Loan.allInstances->exists(l: Loan | not(Loan.allInstances@pre->includes(l) and
        l.user = u and l.copy = c)
```

Library:: indicates that this is a fragment of specification for the borrow action on Library objects. The operation includes is the collection membership predicate. exists is the operation for existential quantification on collections. allInstances@pre returns the set of existing objects of a type prior to the invocation of an operation.

Another way of defining dynamic behaviour in a model is to use state diagrams. We will not cover all aspects of state diagrams as described in UML. For example, we leave out activities on states and do not consider many of the different kinds of events allowed. Instead, following the lead of D'Souza and Wills [10][11] and Daugherty [9], we strive to give an interpretation to state diagrams which integrates well with the other object-oriented modelling notations at our disposal. In this context, a state diagram can be regarded as just a means of visualising some aspects of what can be written textually.

State diagrams in UML are based on Harel statecharts [19]. They are used to model (in part) dynamic behaviour, specifically how the (abstract) state of the system changes as it responds to events.

An example state diagram is given in Figure 20. It shows the states of an arbitrary Copy object, and how that object responds to various events. In this case the events are operations on the library system known to the copy.

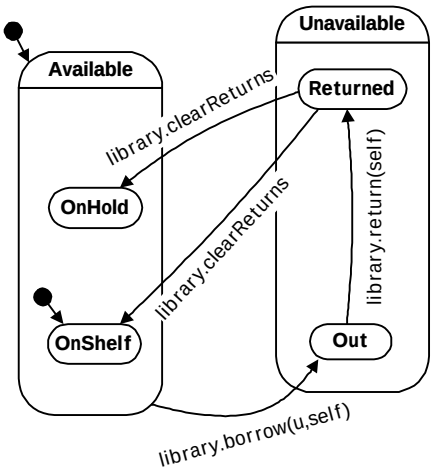

Figure 20: State diagram for Copy

The diagram indicates that the copy can be in one of two states, Available or Unavailable. Those states are further partitioned into substates. Thus when a copy is available it may either be on hold or on a shelf. Thus a state diagram provides an alternative notation for showing substates, dynamic subtypes.

Transitions run between states. They mean that when the labelled event happens, the object changes state from the source of the transition to the target. Thus if the library associated with the copy performs a return action and the argument to that action is this object, then the copy will move from the Out to the Returned state. When the library performs the clearReturns action it takes all the copies from the return bin and, for each one, either puts it back on the shelf or puts them on hold for a reservation. Actions with under-determined behaviour, such as for clearReturns, are tolerated in a specification model.

For the borrow transition in Figure 20, the textual equivalent may be derived as follows. First write the specification from the point of view of a Copy object:

```
Copy::library.borrow(u:User, self)
    pre
        Available
    post
        Out
```

Then convert this to a specification on a Library object, by replacing self with the formal argument of borrow, noting that self is generally omitted from the start of navigation expressions (i.e. available is equivalent to self.available):

Library::borrow(u:User, c:Copy)
pre
c.Available
post
c.Out

The full specification can then be obtained by composing this with the very first fragment: the pre- and post-conditions, respectively, are conjoined.

Library::borrow(u:User, c:Copy)
pre
u is registered and active, and c is available for lending, to u in particular
registered->includes(u) and u.Active and c.Available and
c.availableTo->includes(u)
post
c is marked as out and no longer available for lending to any users,
c.Out and (c.availableTo->isEmpty)
and a new loan of c to u is recorded
and Loan.allInstances->exists(l: Loan | not(Loan.allInstances@pre->includes(l) and
l.user = u and l.copy = c)

Another form of action composition is what has been dubbed angelic composition. Here pre-conditions are disjoined and the post-condition takes the form $(pre1 \Rightarrow post1) \wedge (pre2 \Rightarrow post2)$. Angelic composition is used e.g. when deriving the specification of an action by composing fragments derived from different transitions labelled by that action in the state diagram of any one particular object type.

In Figure 20 there is only one transition labelled borrow, so this approach is equivalent to adding the annotations direct to the state diagram aka Syntropy, where preconditions get re-termed *guards*. If there are many transitions with the same label, then some indication must be given as to which fragment of specification should be composed with the specification derived from which transition, where a fragment could be combined with many transitions. Syntropy allows a specification fragment to be combined with a single transition (the transition is annotated) or to all transitions (the specification is written separately). Catalysis proposes that compositions should be written explicitly.

To summarise, state diagrams can only be used to visualise part of an action specification, the rest must be written textually. It is largely immaterial whether these textual fragments appear as annotations on the state diagram or are written separately. It is also possible to write the part of the specification contributed by a state diagram textually, and this may be composed with any other fragments of specification to produce the complete contract for an action.

5 Semantics: Dynamic Aspects

In our formalisation actions are interpreted as relations between two system states, the state before and the state after the action is executed. We illustrate this by interpreting the specification of the action borrow.The action borrow takes a library, user and copy objects as parameters and its effect is to create a loan object associated with these objects. The action is interpreted by including the following signature:

$$borrow : Library, User, Copy \rightarrow Relation[\Sigma, \Sigma]$$

where the assertion $(\sigma,\sigma') \in borrow(l, u, c)$ asserts that the action borrow is executed on the library object l, user object u and copy object c in the transition from state σ to σ'. The sort $Relation[\Sigma, \Sigma]$ is the sort of relations between system states[1].

The transition axiom of an action act with precondition pre and postcondition post is in general expressed as:

$$pre \wedge act(p1, p2, \ldots) \Rightarrow post$$

that is, if the action act is executed on parameters p1, p2, . . . , when the precondition pre holds, then on termination the postcondition post holds. pre is the interpretation of the precondition pre and post is the interpretation of post.

borrow-Spec1 : **trait**

includes

BasicAssociation(Publication, Copy, publication, copies),
BasicAssociation(Reservation, Copy, OnHoldFor, heldCopy),
BasicAssociation(Publication, Reservation, publication),
Basic-Subtyping-Dynamic(User, Active, $\overline{User}$, $\overline{Active}$, memberA)

introduces

pre: Library, User, Copy, $\Sigma \rightarrow$ Bool
post: Library, User, Copy, Σ, $\Sigma \rightarrow$ Bool

asserts

pre(l, u, c, σ) == (c $\in$ collection(l, σ) $\wedge$ u $\in$ registered(l, σ) $\wedge$
memberA(u, σ) $\wedge$ c $\in$ available(u, σ))
post(l, u, c, σ, σ') == $\overline{availableTo(c, \sigma')}$ = { } $\wedge$
($\exists$lo:Loan(lo $\in$ $\overline{Loan}(\sigma')$ $\wedge$ lo $\notin$ $\overline{Loan}(\sigma)$ $\wedge$
user(lo, σ')={u} $\wedge$ copy(lo, σ')={c}))

Figure 21: Specification of a fragment for Borrow

The pre/post conditions of an operation are basically boolean expressions expressed in terms of navigation expressions built by applying attributes and role names to variables or other expressions. Therefore the semantics of pre/post conditions can be given by first providing the semantics of navigation expressions occurring in them. In our formalisation, navigation expressions correspond to the language of terms over the signatures of specification traits. For example, the feature allInstances can be interpreted in a straightforward manner, by interpreting Loan.allInstances as $\overline{Loan}(\sigma)$. The boolean operators and set operators are easily mapped to larch operators.

For each partial specification of the action borrow we construct a trait that interprets the pre/post conditions. For example, the trait borrow-Spec1 in Figure 21 interprets the pre/post conditions in the textual specification (not the one derived from the state diagram). While the trait borrow-Spec2 in Figure 22 interprets the pre/post conditions of the specification derived from the state diagram.

The full specification of the action borrow is given by the trait borrow-Spec (Figure 23) which includes the two traits borrow-Spec1 and borrow-Spec2 together with introducing two functions pre and post which interpret the full pre/post conditions. These two functions are defined in terms of pre1, post1, pre2 and post2 as given in the asserts section of borrow-Spec. In addition, this trait contains the transition axiom for borrow. Intuitively, this axiom asserts that if the action borrow is executed in the transition from state σ to σ' on an active user u and a copy c available to u then a loan is created and gets associated with the user u, copy c and library l. Angelic composition of action specifications as supported by Catalysis can be done in a similar way.

In the postcondition of the action we only stated what would change and not what does not change. However, to be complete, it is necessary to state exactly what remains unaffected by the action (referred to as *frame conditions).* For example, we must state that user objects other than u are unaffected by the action. The same applies to the copies

1. this is not standard Larch but it can easily be mapped to Larch sort.

borrow-Spec2 : **trait**

includes
Object-Type-Library, Object-Type-Copy, Object-Type-User,
Basic-Subtyping-Dynamic(Copy, Available, $\overline{\text{Copy}}$, $\overline{\text{Available}}$, memberAc),
Basic-Subtyping-Dynamic(Copy, Out, $\overline{\text{Copy}}$, $\overline{\text{Out}}$, memberOut)

introduces
pre: Library, User, Copy, $\Sigma \rightarrow$ Bool
post: Library, User, Copy, $\Sigma, \Sigma \rightarrow$ Bool

asserts
pre(l, u, c, σ) == memberAc(c, σ)
post(l, u, c, σ, σ') == memberOut(c, σ')

Figure 22: Specification of a fragment from state transition for Borrow

borrow-Spec : **trait**

includes
borrow-Spec1[pre1 for pre, post1 for post],
borrow-Spec2[pre2 for pre, post2 for post],
Relation(Σ, Σ)

introduces
pre: Library, User, Copy, $\Sigma \rightarrow$ Bool
post: Library, User, Copy, $\Sigma, \Sigma \rightarrow$ Bool
borrow : Library, User, Copy $\rightarrow$ Relation[Σ, Σ]

asserts
pre(l, u, c, σ) == (pre1(l, u, c, σ) $\wedge$ pre2(l, u, c, σ))
post(l, u, c, σ) == post1(l, u, c, σ) $\wedge$ post2(l, u, c, σ)
pre(l, u, c, σ) $\wedge$ (σ, σ') $\in$ borrow(l, u, c) $\Rightarrow$ post(l, u, c, σ)

Figure 23: Full specification of Borrow

other than c. However, loose specifications are necessary when actions can occur simultaneously. In this paper we shall not consider how to express frame conditions, however, a similar approach to Borgida et al. [5] can be adopted to express these conditions.

Each action in the specification model is specified by a separate trait by including only the relevant object types and associations traits. The specification of the whole dynamic model is obtained by including in a single trait all the traits that specify the actions of the system. The trait for the whole specification model includes the two traits that specify the static and dynamic aspects of the system respectively. This trait is given in Figure 24 where it is assumed that the specification of the state diagram for Copy is included in the trait that specifies the static aspects.

Library-System : **trait**
includes
Library-Static-Model,
Library-Dynamic-Model

Figure 24: Specification of library system

6 Summary and Further Work

A formal and compositional semantics has been provided for object types, states, attributes, associations, invariants, action specifications and state transitions, which are essential to making object-oriented modelling notations precise. The semantics is given in terms of theories in the Larch Shared Language, and a systematic mapping from type dia-

grams to Larch theories has been described. We have not considered all constructs in object-oriented modelling such as aggregation and qualified associations. However, these constructs can be given a precise semantics similar to the constructs we considered.

This work provides the semantics for a kernel of the UML extended with a precise language for expressing invariants, pre & post conditions, namely OCL. Future work includes:

- The application of this semantic approach to composition and specialisation of components, described as object-oriented models. We do not foresee any significant problems in this task.
- Formalisation of refinements, comprising an abstract model, a concrete model and a mapping between them. Here we expect to build on work in the formal methods community [20][30], and recent developments in applying this in the arena of object-oriented modelling. The novel part here will be working out the allowable mappings between models described *diagrammatically*.
- Using this approach to develop the semantics of "Constraint Diagrams'' in [22][23], including working out a calculus for diagram composition. This would allow us to provide an even more acceptable formal semantics for object modelling notations, as constraint diagrams could then be used to express the semantics of other diagrams [24].

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 assist with the development of CASE tools which are able, for example, to check the integrity of models, check conformance between models, to support view extraction from models, and to support assembly of components through their specifications. We have already begun to experiment with the Larch proof assistant to help with this and used the Larch semantics as a basis for mapping models into Prolog for automated checking and animation. We are also exploring the possibility of using more advanced techniques such as model checking [7], and, with constraint diagrams as the foundation, diagrammatic reasoning [2].

Acknowledgments

We are grateful to the BIRO research team at Brighton, in particular Richard Mitchell and Franco Civello for many useful comments and feedback. This research was partially funded by the UK EPSRC under grant GR/K67304.

References

1. Allen P. Components and Objects. Available from http://www.selectst.com/download, 1997.
2. Allwein G, Barwise J (eds). Logical Reasoning with Diagrams. Oxford University Press, 1996.
3. Bicarregui J, Lano K, Maibaum T. Towards a Compositional Interpretation of Object Diagrams. In: Bird and Meertens (ed) IFIP TC2 Working conference on Algorithmic Languages and Calculi. Chapman and Hall, 1997.
4. 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.
5. Borgida A, Mylopoulos J, Reiter E. On the Frame Problem in Procedure Specifications. IEEE Transactions in Software Engineering, 1995; Vol. 21, No. 10.
6. Bourdeau H, Cheng B. A Formal Semantics for Object Model Diagrams. IEEE Transactions on Software Engineering, 1995; 21:799-821.
7 Clarke E M, Grumberg O, Long D E. 'Model Checking and Abstraction', Proceedings of the ACM Symposium on Principles of Programming Languages, January, 1992.
8. Cook S, Daniels J. Designing Object Systems with Syntropy. Prentice Hall Object-Oriented Series, 1994.
9. Daugherty G. State Diagrams as 'Views' of Formal OO Models. OOPSLA97 Workshop on Object-Oriented Behavioural Semantics, Technical Report TUM-I9737, Technische Universitat Munchen, 1997.

10. D'Souza D, Wills A. *Extending Fusion: Practical Rigor and Refinement*. In: R. Malan et al. (ed) OO Development at Work, Prentice Hall, 1996.

11. D'Souza D, Wills A. Objects, Components and Frameworks with UML: The Catalysis Approach. Addison-Wesley, 1998, to appear. Draft and related material available at http://www.trireme.com/catalysis.

12. 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.

13. Fiadeiro J, Maibaum T. Temporal Theories and Modularisation Units for Concurrent System Specification. Formal aspects of Computing, 1992, Vol.4, No. 3, pp. 239-272, Springer Verlag.

14. Fowler M, Scott K. UML Distilled. Addison-Wesley, 1997.

15. Guttag J, Horning J. Larch: Languages and Tools for Formal Specifications. Springer-Verlag. 1993.

16. Hamie A, Civello F, Howse J, Kent S, Mitchell R. Reflections on The Object Constraint Language. In the International Proceedings of <<UML>>'98 workshop, Mulhouse, France, June, 1998.

17. Hamie A, Howse J. Interpreting Syntropy in Larch. Technical Report ITCM97/C1, University of Brighton. 1997.

18. Hamie A, Howse J, Kent S. Navigation Expressions in Object-Oriented Modelling Notations. In the proceedings of FASE in ETAPS98, Portugal, Lisbon, Springer Verlag 1998. Available from http://www.it.brighton.ac.uk/staff/Stuart.Kent, 1997.

19. Harel D. Statecharts: A Visual Formalism for Complex Systems. Sci. Comput. Prog. 8, pp. 231-274, 1987.

20. Jones C. Systematic Software Development using VDM (2nd edition). Prentice Hall, 1990.

21. Kent S, Lano K, Bicarregui J, Hamie A, Howse J. Component Composition in Business and System Modelling. Kilov H., Rumpe B. and Simmonds I. OOPSLA97 Workshop on Object-Oriented Behavioural Semantics, Technical Report TUM-I9737, Technische Universitat Munchen, 1997.

22. Kent S. Constraint Diagrams: Visualising Invariants in Object-Oriented Models. Procs. of OOPSLA97, ACM Press, 1997.

23. Kent S. Visualising Contracts in Object-Oriented Models. Presented at VISUAL98 in ETAPS98, available from http://www.it.brighton.ac.uk/staff/Stuart.Kent, 1997.

24. Kent S, Hamie A, Howse J, Civello F, Mitchell R. Semantics Through Pictures. ECOOP'97 workshop reader, LNCS series, Springer Verlag, 1998.

25. Rumbaugh J, Blaha M, Premerali W, Eddy F, Lorensen W. Object-Oriented Modelling and Design. Prentice Hall, 1991.

26. Short K. Component Based Development and Object Modeling. Available from http://www.cool.sterling.com/cbd, 1997.

27. UML Consortium. Object Constraint Language Specification Version 1.1. Available from http://www.rational.com, 1997.

28. UML Consortium. The Unified Modeling Language Version 1.1. Available from http://www.rational.com, 1997.

29. Kleppe A, Warmer J, Cook S. Informal Formality? The Object Constraint Language and its application in the metamodel. In the International Proceedings of <<UML>>'98 workshop, Mulhouse, France, June, 1998.

30. Woodcock JCP. The refinement calculus, Procs. VDM Symposium 91, Delft, The Netherlands, Springer-Verlag LNCS 552, 1991.