Semantics through Pictures

Towards a diagrammatic semantics for OO modeling notations

Stuart Kent, Ali Hamie, John Howse, Franco Civello, Richard Mitchell[1]
Division of Computing,
University of Brighton, Lewes Rd., Brighton, UK.
http://www.biro.brighton.ac.uk/index.html, biro@brighton.ac.uk
fax: ++44 1273 642405, tel: ++44 1273 642494

Abstract. An object-oriented (OO) model has a static component, the set of allowable snapshots or system states, and a dynamic component, the set of filmstrips or sequences of snapshots. Diagrammatic notations, such as those in UML, each places constraints on the static and/or dynamic models. A formal semantics of OO modeling notations can be constructed by providing a formal description of (i) sets of snapshots and filmstrips, (ii) constraints on those sets, and (iii) the derivation of those constraints from diagrammatic notations. In addition, since constraints are contributed by many diagrams for the same model, a way of doing this compositionally is desirable. One approach to the semantics is to use first-order logic for (i) and (ii), and theory inclusion with renaming, as in Larch, to characterize composition. A common approach to (iii) is to bootstrap: provide a semantics for a kernel of the notation and then use the kernel to give a semantics to the other notations. This only works if a kernel which is sufficiently expressive can be identified, and this is not the case for UML. However, we have developed a diagrammatic notation, dubbed constraint diagrams, which seems capable of expressing most if not all static and dynamic constraints, and it is proposed that this be used to give a diagrammatic semantics to OO models.

1 Introduction

This paper outlines an approach to constructing a precise semantics for object-oriented modeling notations. There are at least four reasons why one might want to build a precise semantics:

1. To clarify meaning leading to refinements of the notation.
2. To clarify meaning for developers using the notation.
3. To clarify meaning for tool developers, thereby increasing the likelihood of interoperability between tools at a semantic level (e.g. code generated from different tools for the same model has the same behavior).
4. To support semantic checking of models, automated if possible. This includes checking that implementations meet their specifications, checking internal consistency of components, and checking for inconsistencies and conflicts between components.

(1) just requires the semantics to be written down in a precise form. (2) and (3) requires it to be written down in a form which developers and tool developers can easily understand. In addition, it would be desirable for (3) to provide a semantics in a form which directly assists the construction of tools, e.g. the automation of (4).

The approach to semantics advocated in this paper aims to meet all four objectives. Precision (1) is achieved by grounding the semantics in first-order predicate logic. The dialect used here is Larch (Guttag and Horning 1993): it has a precisely defined language; support for theory composition; and tool support in the form of a syntax/type checker and theorem prover.

One could argue that the use of predicate logic supports (2) and (3). However, we think we can do better. Specifically we are proposing to use a diagrammatic notation, dubbed constraint diagrams (Kent 1997): these allow sophisticated constraints on a model to be expressed diagrammatically; the meaning of diagrammatic notations for OO modeling can be characterized essentially in terms of the constraints they impose on the underlying model, and these can be expressed using constraint diagrams. At the very least, this provides an alternative approach to making mathematics more palatable to e.g. ADL (1997). (4) is achieved through at least two possible routes:

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

1. Larch comes with theorem proving tools, so, in theory at least, these could be used to perform semantic checks.
2. By treating constraint diagrams as graphs, they may be compared for matches or mismatches as a way of performing consistency checks between different perspectives and views of the model.

Extracts from the specification of a library system are used throughout the paper for illustration. This should be distinguished from the specification of the business domain with or without the system embedded in it. The same or similar notations and techniques can be used to model that as well; for example, it is claim of UML (UML 1997) that it is not just a language for modeling software. The specification is given in a subset of the notations proposed for UML, enhanced with a language for writing constraints precisely (e.g. invariants, pre & post conditions) based on Catalysis (d'Souza and Wills 1995, 1997).

2 Underlying model: snapshots and filmstrips

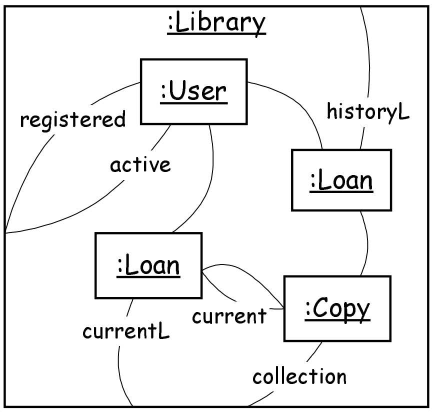

Figure 1: Admissible snapshot

An object oriented (OO) model may be characterized in terms of the possible states the system being modeled can enter and the order in which it can enter those states. We use the term ***snapshot*** to refer to a possible state of the system and ***filmstrip*** a possible ordering on those states, i.e. a sequence of snapshots. The set of snapshots form the ***static model***, and the set of filmstrips the ***dynamic model***.

Figure 1shows a snapshot that is admitted by the model of a library system. A copy is currently on loan to an active user (only active users may borrow things). The copy has previously been on loan to the same user. The notation used is that of instance diagrams in UML.

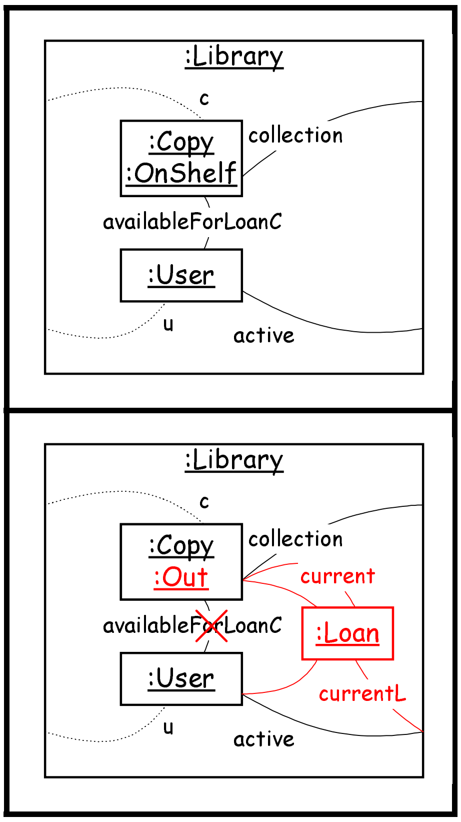

Figure 2: Filmstrip segment for borrow

Figure 2 shows a segment of one of the many possible filmstrips admitted by the model. This segment shows before (pre) and after (post) snapshots for an invocation of the action borrow(u:User, c:Copy) on the Library object depicted. The copy c is up for loan to u; u must be active, c in the collection and c available for loan to u. On completion the copy is marked as “loaned out”, and a new current loan object is created to record that c has been loaned out to u.

The set of filmstrips forming the dynamic model may be thought of as generated by stringing together pre/post segments for actions on the system being modeled.

3 Generic descriptors: perspectives on a model

Of course the sets of allowed snapshots and filmstrips characterizing a model are in general infinite (for specification models), and, at best very large (for implementation models). Therefore modelers need notations that are able to define very large sets in only a few diagrams. UML calls these notations ***generic descriptors***. Essentially generic descriptors provide ways of writing rules or constraints which determine whether any particular snapshot or filmstrip is allowed in a model or not. Here we consider type and state diagrams (from UML) combined with invariants and action specifications.

3.1 Type diagrams

Type diagrams define most of the language that can be used in snapshots and constrains multiplicities of links between objects in snapshots. The type model for the snapshots appearing in Figures 1 and 2, is given in Figure 3.

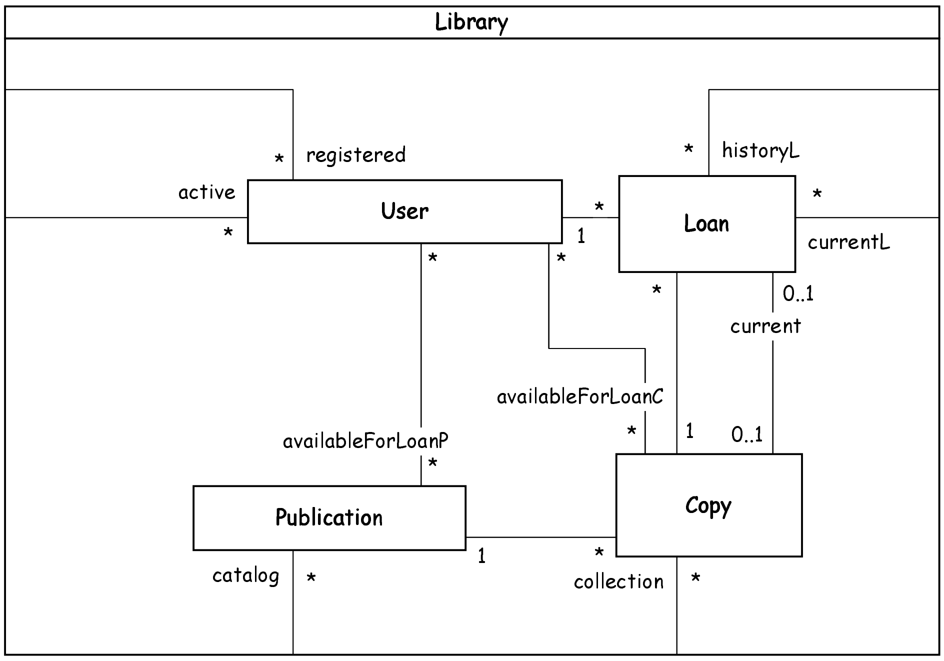

Figure 3: Type model for library

Only types and association rolenames appearing in the type model may appear in snapshots. Furthermore the number of links in a snapshot corresponding to a particular association may not exceed the multiplicity constraints declared on the type model, for any objects of the types associated. For example, focussing on the (unlabeled) associations between Loan and User and Loan and Copy, a loan object may be linked with only one user and one copy, though user and copy objects may be linked to many loan objects. This is the case in Figure 1 on page 2, where each loan object (there are two of them) are linked with only one user and one copy, but the user and the copy happen to be the same for both so are each linked to two loan objects.

The type diagram also uses the UML ***composite*** notation, by placing types and associations within the bounds of another type - Library in this case. There are a variety of possible meanings for this, reflecting the various possible interpretations of composite (see e.g. Civello 1993). All could be expressed as constraints on snapshots (e.g. no sharing) and sequences of snapshots (e.g. lifetime dependency). Certainly one constraint we would like is that whatever navigation route is taken, all paths lead back to the same library; this has a pre-requisite that the objects are only ever associated with one library object. For example, we would expect that self.catalog.copies.~collection = self or self.active.loans.~currentLoans = self, and so on.[1] In snapshots, this constraint is realized by always having a single bounding library object from whence all paths come and to which all paths go.

3.2 State diagrams

A state diagram places constraints on both the static and dynamic models. The state diagram for the type Copy, in the context of the Library, is given in Figure 4.

1. For more on the meaning of navigation expressions, see Section 3.3, “Invariants,” on page 5. Note that self on the LHS of these expressions is redundant.

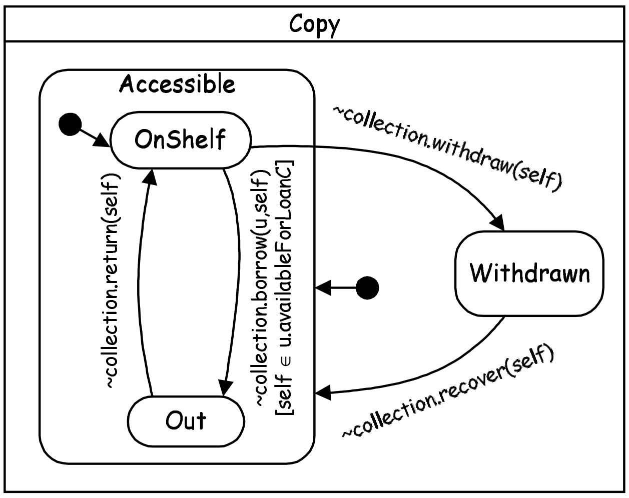
Copy
Accessible
OnShelf
~collection.withdraw(self)
~collection.return(self)
~collection.borrow(u,self)
[self ∈ u.availableForLoanC]
Withdrawn
Out
~collection.recover(self)

Figure 4: State Diagram for Copy

This is essentially UML notation, though we allow navigation expressions labelling the transitions. For example the diagram indicates that when the action borrow is performed on the object identified through ~collection with self as the copy argument, then, provided self is in the OnShelf state, the effect will be to move it into the Out state. This reflects a style of specification used in e.g. Catalysis (d'Souza and Wills 1995, 1997) and Syntropy (Cook and Daniels 1994) where actions on the "system" object only are specified. Note that at this level of modeling i.e. specification, navigation does not mean message passing. It is only in the design model that one begins to decide how actions are implemented by allocating responsibilities to objects and passing messages between them (viz. sequence diagrams).

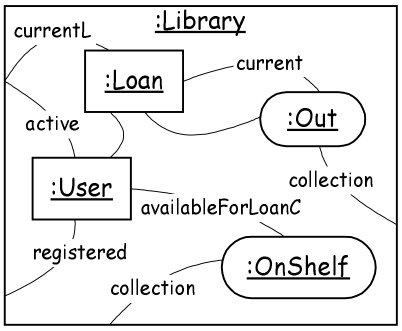

Figure 5: Snapshot admitted by state diagram

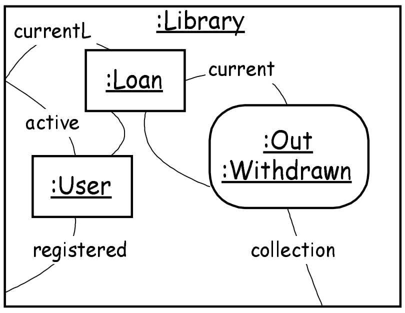

Figure 6: Snapshot disallowed by state diagram

The constraints on the static model imposed by a state diagram are the introduction of new states and the relationships between them. For example, Figure 4 declares that a Copy has states Accessible, OnShelf, Out and Withdrawn, and that Out and Withdrawn are substates of Accessible. States may be thought of either as Boolean attributes or as dynamic types, and indicated as such on snapshots. In figures 5 and 6 a state-like box[1] has been used to indicate an object of a dynamic type, in this case copies in a particular state. Figure 5 is admitted by Figure 4, but Figure 6 is not - the copy is Out and Withdrawn at the same time.

The constraints on the dynamic model may be expressed as formal specifications on the actions mentioned in the state diagram. For example, the specification for borrow read off from the state diagram is given opposite.

This uses a precise language to express pre and post conditions. If the pre condition is satisfied on invocation of the action then the post condition should be satisfied on completion. If the pre condition does not hold, then the effect of the action is undetermined. Only filmstrip segments with pre and post snapshots satisfying the pre and post conditions, respectively, will be admitted in the model.

Library::borrow(u:User,c:Copy)
<u>pre</u>
c ∈ collection[1]
c ∈ u.availableForLoanC[2]
c.Onshelf
<u>post</u>
c.Out

1. The state diagram refers to ~collection.borrow, so it only constrains Library::borrow for copies in collection.
2. c is available for lending to the user u. This comes from the guard (shown between [...]) on the state diagram.

1. As far as we are aware, this is not part of current UML. However, we note that rectangles are used to represent types on type diagrams and objects of those types on instance diagrams. Rounded rectangles are used to indicate states on state diagrams, and these directly correspond to dynamic types, so could be used to represent these on type diagrams. It is then a natural step to use them to indicate objects of dynamic type (i.e. in a particular state) on instance diagrams. This also explains why we have used a rectangle as the outermost "state" on the state diagram, which is really a static type.

3.3 Invariants

Diagrams, or at least those well known in OO modeling, can not express all required static constraints. For example, in the library system we would like to say that the current loans of copies in the collection is exactly the set currentL; and that only active users can have current loans. In other words the snapshot in Figure 7, is not allowed.

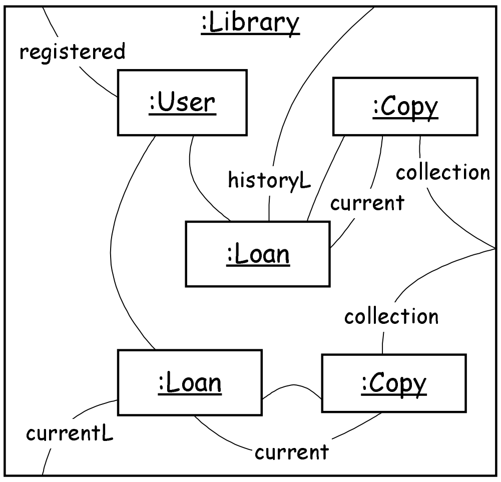

Figure 7: Snapshot disallowed by invariant

Formally these constraints can be written

(inv 1) collection.current = currentL

(inv 2) currentL.users $\subseteq$ active

respectively. The meaning of navigation expressions used here is taken from Catalysis (d'Souza and Wills 1995, 1997): collection.current returns the set of loans obtained by traversing the link current for every copy in collection; for the mathematically minded, this is the range of the composition of the collection and current relations, restricted to self in the domain.

3.4 Action specifications

A state diagram does not say everything about the effects of an action. For example, in the case of borrow, it does not say that a new, current loan object must be created recording the fact that the copy has been loaned out to the user. Using the same precise language as used in Section 3.2, "State diagrams", another fragment of specification for borrow is:

Library::borrow(u:User,c:Copy)
<u>post</u>
The loan of c to u is recorded in a new current loan object.
$\exists l$: loan, $l \in$ new $\wedge$ $l \in$ currentL $\wedge$ l.user = u $\wedge$
l.copy = c $\wedge$ c.current = l

This must be combined with the specification derived from the state diagram. Catalysis (d'Souza and Wills 1997) describes two of ways of combining action specifications, differing on whether and how pre and post conditions are conjoined/disjoined. View composition is appropriate here; details are given in the next section.

4 Semantics: compositions of Larch traits

A formal semantics for an object-oriented model makes precise the notion of snapshots, filmstrips and constraints on them. It also defines how static and dynamic constraints are derived from generic descriptors.

For the reasons given in the introduction, our approach is to use Larch (Guttag and Horning 1993) as the language of formalization, at least at the lowest level. Not only does this have an associated, freely available toolset, it also provides a relatively sophisticated form of theory composition which supports renaming. A theory or ***trait*** in Larch characterizes a (collection of) constraint(s) on a model, and the language required to describe it (them); theory composition is used to construct a theory characterizing all constraints on the model, hence the model itself. Theory composition supports what naturally takes place in modeling: the model is described using a series of diagrams and text fragments of various kinds, each contributing constraints on the model. Composition also plays an important role in providing sophisticated tool support: for partitioning semantic checks into manageable units; and for component-based development where components need to be composed, decomposed and compared.

By way of illustration, a sketch of the Larch semantics for the library system model is given in Figure 8.

`Type(T)` : **trait**
  introduces
  $\texttt{exists} : T, \Sigma \rightarrow \texttt{Bool}$

`DirectedAssoc(S, T, r)` : **trait**
  includes
  `Type(S)`
  `Type(T)`
  introduces
  $r : \Sigma \rightarrow \texttt{Relation}[S, T]$[1]
  asserts
  $\forall \sigma : \Sigma, s : S, t : T$
    $r(s, t, \sigma) \Rightarrow$
    $(\texttt{exists}(s) \wedge \texttt{exists}(t))$

$\texttt{Association}(S, T, r_{\texttt{StoT}}, r_{\texttt{TtoS}})$: **trait**
  includes
  $\texttt{DirectedAssoc}(S, T, r_{\texttt{StoT}})$
  $\texttt{DirectedAssoc}(T, S, r_{\texttt{TtoS}})$
  asserts
  $\forall \sigma : \Sigma, s : S, t : T$
  $\langle s, t \rangle \in r_{\texttt{StoT}}(\sigma) \Leftrightarrow \langle t, s \rangle \in r_{\texttt{TtoS}}(\sigma)$

`LibraryTA` : **trait**
  includes
  `Association(Library, User, active, ~active)`
  `Association(Copy, Loan, current, ~current)`
  …

`LibrarySC` : **trait**
  includes
  `Optional(Copy, Loan, current)`
  …
  `Library::Inv1`
  `Library::Inv2`
  …

`LibrarySM` : **trait**
  includes
  `LibraryTA`
  `LibrarySC`

`LibraryDM` : **trait**
  includes
  `Library::borrow`
  …

`Library` : **trait**
  includes
  `LibrarySM`
  `LibraryDM`

1. This is actually not standard Larch. For presentational reasons, we have used a similar trick for relations as has been used for sets in the new version of Larch (Horning 1995). It can be translated systematically into standard Larch.

Figure 8: Extracts of compositional semantics in Larch for Library system model

Each **trait** is a theory of FOPL. The **includes** section indicates which other traits are used in the construction of this one, the **introduces** section declares the new functions introduced in the theory and the **asserts** section lists axioms imposing constraints on all the functions (introduced or included). The traits in the left hand column illustrate how types and associations are modeled. The `Type` trait introduces the basic components for a (static) type: a sort `T` of identities for objects of that type, and a predicate `exists` indicating at what points in time (from Σ) an object exists; an object exists only after it has been created and before it is destroyed; only objects which exist have behavior.

The `DirectedAssoc` trait defines an association from type `S` to type `T` with rolename `r`, with the constraint that only existing objects may be related by the association. An association is then two directed associations, with a constraint that the inverse of one is equal to the other. This is similar to the semantics proposed for associations in Graham et al. (1997).

The right-hand column indicates how the semantics of the `Library` model, defined by the `Library` trait, is built up. `LibraryTA` defines the types and associations i.e. the language; `LibrarySC` the static constraints, including multiplicity constraints on associations; `LibrarySM` the static model, which is the composition of the language and the static constraints; and `LibraryDM` the dynamic model, which is just a list of action specifications.

A static constraint and an action specification are given in Figure 9 on page 7, and Figure 10 on page 7, respectively. The static constraint is the one derived from invariant 2 on page 5. Only the language (i.e. associations) required to write the invariant are included in the trait. The assertion illustrates how navigation expressions are given a semantics: the image (range) of the composition of the relations, corresponding to the associations involved, restricted to the set of objects being navigated from, in this case the library `l`.

`Library::Inv2` : **trait**

includes

`DirectedAssoc(Library, User, active)`

`DirectedAssoc(Library, Loan, currentL)`

`DirectedAssoc(Loan, User, user)`

asserts

$\forall \sigma : \Sigma, \mathtt{l} : \mathtt{Loan}$

$\mathtt{image}(\mathtt{currentL}(\sigma) \cdot \mathtt{user}(\sigma), \{\mathtt{l}\})$
$\subseteq \mathtt{image}(\mathtt{active}(\sigma), \{\mathtt{l}\})$

Figure 9: Semantics of a static constraint

The action specification is formed from the composition of the parts contributed by the state diagram in Figure 4 on page 4 and the separate fragment of specification on page 5, as defined by the `Library::borrow` trait in Figure 10.

`Library::borrow` : **trait**

includes

`Library::borrow-1`

`Library::borrow-2`

`Library::borrow-1` : **trait**

…

$\forall \sigma_{\mathtt{old}}, \sigma_{\mathtt{new}} : \Sigma \forall \mathtt{l} : \mathtt{Library}, \mathtt{u} : \mathtt{User}, \mathtt{c} : \mathtt{Copy}$

$\mathtt{borrow\text{-}pre}(\mathtt{l}, \mathtt{u}, \mathtt{c}, \sigma_{\mathtt{old}})$
$\Rightarrow (\mathtt{c} \in \mathtt{image}(\mathtt{collection}(\sigma_{\mathtt{new}}), \{\mathtt{l}\})$
$\wedge \mathtt{image}(\mathtt{OnShelf}(\sigma_{\mathtt{new}}), \{\mathtt{c}\}) = \{\mathtt{true}\})$
$\wedge \mathtt{c} \in \mathtt{image}(\mathtt{availableForLoanC}(\sigma_{\mathtt{new}}), \{\mathtt{u}\})$

…

$\mathtt{Action}(\Sigma)$: **trait**

includes

$\mathtt{Relation}(\Sigma, \Sigma)$

…

`Library::borrow-2` : **trait**

includes

`DirectedAssoc(Library, Loan, currentL)`

`DirectedAssoc(Loan, User, user)`

`DirectedAssoc(Loan, Copy, copy)`

`DirectedAssoc(Copy, Loan, current)`

introduces

$\mathtt{borrow} : \mathtt{Library}, \mathtt{User}, \mathtt{Copy} \to \mathtt{Action}[\Sigma]$

$\mathtt{borrow\text{-}pre} : \mathtt{Library}, \mathtt{User}, \mathtt{Copy}, \Sigma \to \mathtt{Bool}$

asserts

$\forall \sigma_{\mathtt{old}}, \sigma_{\mathtt{new}} : \Sigma, \mathtt{l} : \mathtt{Library}, \mathtt{u} : \mathtt{User}, \mathtt{c} : \mathtt{Copy}$

$(\langle \sigma_{\mathtt{old}}, \sigma_{\mathtt{new}} \rangle \in \mathtt{borrow}(\mathtt{l}, \mathtt{u}, \mathtt{c})$
$\wedge \mathtt{borrow\text{-}pre}(\mathtt{l}, \mathtt{u}, \mathtt{c}, \sigma_{\mathtt{old}})) \Rightarrow$

$\exists \mathtt{lo} : \mathtt{Loan}$

$\mathtt{exists}(\mathtt{lo}, \sigma_{\mathtt{new}}) \wedge \neg \mathtt{exists}(\mathtt{lo}, \sigma_{\mathtt{old}})$
$\wedge \mathtt{lo} \in \mathtt{image}(\mathtt{currentL}(\sigma_{\mathtt{new}}), \{\mathtt{l}\})$
$\wedge \mathtt{image}(\mathtt{user}(\sigma_{\mathtt{new}}), \{\mathtt{lo}\}) = \{\mathtt{u}\}$
$\wedge \mathtt{image}(\mathtt{copy}(\sigma_{\mathtt{new}}), \{\mathtt{lo}\}) = \{\mathtt{c}\}$
$\wedge \mathtt{image}(\mathtt{current}(\sigma_{\mathtt{new}}), \{\mathtt{c}\}) = \{\mathtt{lo}\}$

Figure 10: Semantics of an action specification

The semantics for the fragment of specification on page 5 is given in the `Library::borrow-2` trait. Again this includes only those associations required to write the specification. It introduces a `borrow` function which maps arguments (including one for the type `Library`) to actions, which, as indicated by the `Action` trait, are relations between points in time. The assertion in the `Library::borrow-2` trait characterizes the behavior of the action. Note the meaning assigned to **new**, which insists that there is now a loan object which did not exist at the pre or old state but which does exist in the post or new state, i.e. one that has been created by the action.

Finally note the way in which the pre conditioned is defined. A special function called `borrow-pre` is introduced, which is constrained so that at least the pre conditions supplied by all specification fragments hold when the action is invoked. That is when the theories characterized by this and the `Library::borrow-1` traits are composed to provide the full specification of **borrow**, as in the `Library::borrow` trait, the effect is to 'and' the pre conditions as required. This corresponds to view composition in Catalysis (d'Souza and Wills, 1997).

5 Constraint diagrams

Constraint diagrams (Kent 1997) are a diagrammatic notation for expressing constraints on models. They can be used in isolation to express static constraints, or in sequence to express dynamic constraints. They build upon the effectiveness of snapshots in illustrating the import of constraints on the model. They may be viewed as a generalization of snapshot notation (one diagram represents a set of snapshots), which is more expressive than type diagrams; or as the natural progression from UML object diagrams which have a notation for representing sets of objects. They make use of Venn diagram notation, with some extensions, to show relationships between navigation expressions, interpreted in the Catalysis fashion (see page 5).

Figure 11 on page 8 shows a constraint diagram for the library system model. As with Venn diagram notation

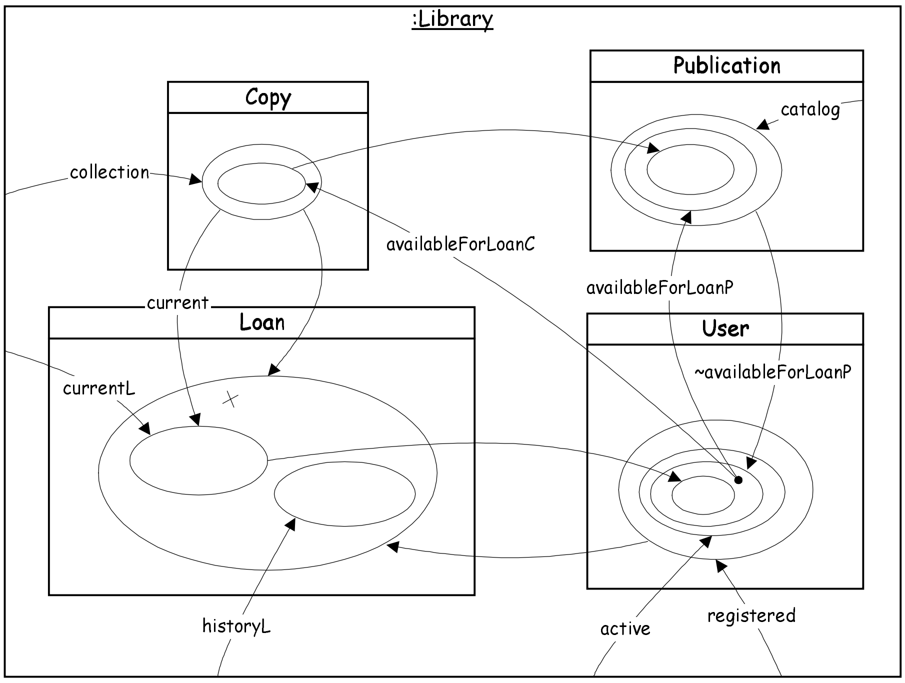

Figure 11: Constraint diagram showing some static constraints on the library system

an ellipse represents a set of objects; a type represents the universal set of objects of that type; a set at the target of an arrow is the set identified by the association at the end of the arrow, navigating from the *set* of objects at the source. Thus, focussing on the Copy and Loan types, a textual representation of one constraint imposed by this diagram is collection.current = currentL.

A set at the source of an arrow, which has no arrows targeted on it is assumed to be universally quantified; a small closed circle is a set containing only one element; a small open circle (not used here), is a set with zero or one elements. Written textually, a constraint illustrating this is

$\forall$u:User, u $\in$ catalog.~availableForLoanP $\Rightarrow$
u.availableForLoanP $\supseteq$ u.availableForLoanC.publications

A cross in any area means there are no elements in that area, as illustrated by the constraint

$$\mathsf{collection.loans} - (\mathsf{currentL} \cup \mathsf{historyL}) = \emptyset$$

Constraint diagrams can be used in sequence to express action specifications, this time generalizing the idea of filmstrips. Figure 12 shows the specification of `borrow` (the part not catered for by the state diagram). The diagram in the first frame shows any pre-condition information together with parts of the state that are subject to change by the action. The diagram in the second frame shows only the changes imposed by the action. The filmstrip is equivalent to the specification fragment for `borrow` given in Section 3.4 on page 5.

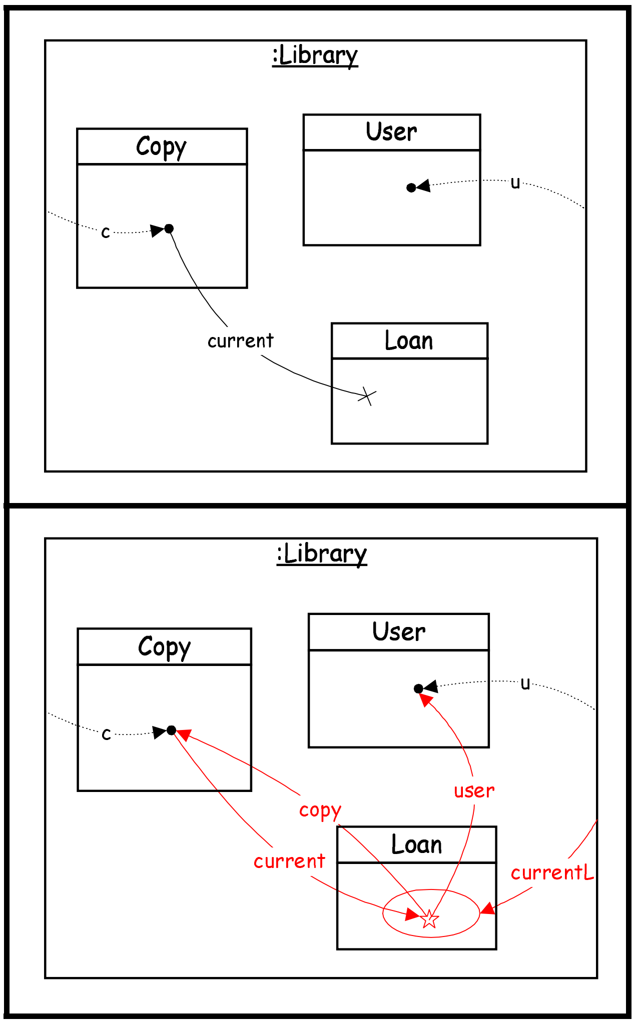

Figure 12: Constraint diagram filmstrip for `borrow`

6 Diagrammatic semantics

Constraint diagrams can be used to express most static and dynamic constraints that can be expressed with invariants, pre and post conditions. This includes constraints imposed by type diagrams - multiplicity constraints are just a particular form of invariant; and constraints imposed by state diagrams, which contribute to the type model (dynamic types) and action specifications.

This suggests that once a formal semantics has been given to constraint diagrams, they can then be used to give a semantics to other notations. This could help to make the semantics easier to understand, and could provide an alternative approach to providing tool support for semantic checking, through direct comparison of diagrams. As well as the diagrams mentioned it is expected that at least object composition (e.g. as described in UML) could be given a semantics using this approach. Indeed, this seems to be a prime candidate as the exact meaning of object composition in UML is far from clear. Civello (1993) and Kilov & Ross (1994) are likely to be helpful here. In particular, the latter makes considerable use of invariants to formalize various forms of composition.

In line with the compositional approach to semantics, it is desirable to construct a calculus for composing constraint diagrams and constraint diagram sequences. Then a model would be precisely characterized by the appropriate compositions of such diagrams. Theory composition as in Larch should provide sufficient mechanisms for formulating such a calculus.

One possible problem with constraint diagrams is the difficulty of showing constraints on attributes which hold values, e.g. of type `Integer`, rather than identify objects. It is expected that this could be cured by showing values like objects on snapshots, sets of values like sets of objects on constraint diagrams, and relationships between them as associations.

It is not possible to give sequence diagrams a semantics using constraint diagrams, as the former carry a different kind of information, namely the order in which messages are passed between objects. However, sequence diagrams combined with constraint diagram filmstrips can express at least the same information as UML collaboration diagrams, which could be given a semantics using their combination.

A sequence diagram with a corresponding constraint diagram filmstrip effectively characterizes the design of an action. A design model is the composition of the designs of all actions in the specification. Comparing the first and last diagrams in the filmstrip with the pre/post filmstrip fragment in the specification model, is sufficient to show that the design is a correct against or ***refines*** the action specification, subject to an appropriate

retrieval. ***Retrieval*** is the VDM (Jones 1990) term used to describe the mapping of the state characterization of one model (e.g. a design) into the state characterization of a higher level model (e.g. a specification), where, in the world of OO modeling, the state is characterized by the types, associations and attributes defined for the model.

7 Summary and conclusions

An object-oriented model has two parts: the static model, a set of allowed snapshots and the dynamic model, a set of allowed filmstrips. These are too many to explicitly enumerate, so generic descriptors are used to characterize general constraints on these sets. A significant subset of generic descriptors can be given a precise semantics by composing Larch traits (i.e. FOPL theories). These may be generated through an intermediate route: constraint diagrams, sequence diagrams and a calculus for composing them are given a semantics in these terms; these notations are then used to give a diagrammatic semantics to other notations. Semantic checks may be based on the Larch layer or the diagrammatic layer.

Many of the details of the approach described here still need to be worked out. A paper describing a semantics of type diagrams, invariants, state diagrams and action specifications in terms of compositions of Larch traits should soon be available (watch the BIRO project website http://www.biro.brighton.ac.uk/index.html). This is based on earlier work on the semantics of Syntropy (Hamie and Howse 1997). Work is just beginning on detailing the semantics of constraint diagrams and their composition. Areas coming into focus include refinement, object composition and component composition and interaction.

As the semantics is defined, it is expected that some refinement of the notation will be required, and this should be recognized by bodies such as the OMG, who are currently standardizing on object-oriented modeling notations.

A main omission from this paper has been any consideration of concurrency and distribution e.g. UML active objects/types/classes and deployment diagrams. One approach, to handling concurrency at least, would be to rework the semantics using a process calculus of some form. Our preference instead would be to use temporal logic, or an encoding in Larch, as this would be a natural extension of the approach advocated here.

References

ADL (1997) *Assertion Definition Language*, The Open Group (formerly X/Open), http://adl.xopen.org.

Civello F. (1993) “Roles of Composite Objects in Object-Oriented Analysis and Design”, in *OOPSLA’93*, pp.376-393, ACM Press.

Cook S. and Daniels J. (1994) *Designing Object Systems*, Prentice Hall Object-Oriented Series.

D’Souza D. and Wills A. (1995) *Catalysis: Practical Rigour and Refinement*, technical report available at http://www.iconcomp.com.

D’Souza D. and Wills A. (1997) *Component-Based Development Using Catalysis*, book submitted for publication, manuscript available at http://www.iconcomp.com.

Graham I., Bischof J. and Henderson-Sellers B. (1997) “Associations considered a bad thing”, in *JOOP*, February 1997, Sigs Publications.

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

Hamie A. and Howse J. (1997a) *Interpreting Syntropy in Larch*, Technical Report ITCM97/C1, University of Brighton, available at http://www.biro.brighton.ac.uk/index.html.

Horning J. (1995) *The Larch Shared Language: Some Open Problems*, presented at the workshop on Abstract Data Types, Oslo, 22 September 1995.

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

Kent S. (1997) *Constraint Diagrams: Visualising Assertions in Object-Oriented Models*, submitted to OOPSLA97.

Kilov H. and Ross J. (1994) *Information modeling: an object-oriented approach*, Prentice-Hall.

UML (1997) *Unified Modeling Language v1.0*, Rational Software Corporation, available at http://www.rational.com.