OOA / D and Corba / IDL:
A Common Base

Desmond D'Souza and Alan Wills
(Questions to dsouzad@acm.org)

A response to the OMG's RFI on Object Analysis and Design

1 *Interoperability: the next step*

OMG goals imply precise behavioral specifications.

THE OMG's principal aims of widespread re-use and interoperability cannot be realized without precise definitions of interfaces and architectures[1]. Conformance to clients' requirements, and to standards, cannot be properly checked unless the behavior expected of a component can be specified unambiguously. These specifications must be machine processable: for re-use to work effectively, a developer's OOA / D tools must be able to search for suitable components and check their conformance to requirements. In fact, the most flexible open systems need to be able to search for, interrogate, select, and appropriately couple components at run time.

Specification of Behavior – the OMG Need

We believe the need for such behavioral specification is a driving force behind the OMG's RFI. It applies to almost all areas that OMG standards apply, from the level of basic object models, through interfaces and architecture for CORBA, to interfaces and architectures for various services such as transaction management, and up to standards for dealing with high-level aspects like compound documents, vertical application integration, and analysis and design artifacts.

"Conformance" is a key notion.

The most important concept in relation to specification is conformance: determining whether a server provides (at least) the behavioral guarantees that a client expects by virtue of their behavior specifications. Any worthwhile method of specification (in relation to component-built and open systems) must include a method of settling this question.Moreover, as there are alternate ways of defining conformance, the semantic basis should permit precise definition of more than one form.

1. Documentation of several current OMG standards suggests this need is quite urgent.

Towards IDL++

We see the next step as being the extension of IDL to cover behavioral specification (rather than just signatures); and the definition of a common linguistic basis between developer's support tools, so that they can exchange information about component interfaces and architectures. We will refer to this (future) language as IDL++.

The language of the specs. must be extensible.

IDL++ must be sufficiently flexible that tools based on it can still communicate effectively even if they do not entirely understand each others' terms. For example, suppose I place a component in a library or on the Web and specify its behavior using plain functional postconditions; a year later, someone searches my library with a requirement expressed in more sophisticated terms, that include formal stipulation of real-time constraints. The search tool should be able to cope with the lack of performance traits in my specification, pulling out my component as a candidate if it looks feasible in plain functional terms.

OOA/D Methods and Behavior Specification

IDL++ and OOA / D specifications must interoperate.

IDL++ specifications and their corresponding implementations will be generated and documented through different means, including a variety of OOA / D languages and CASE tools. If their workproducts can be made to map to a common basic language, then it will be possible not only to interchange information across methods and tools at design time (directly and through libraries), but also for components to be able to advertise themselves in a commonly understood language at run time.

OOA / D methods could help with specification.

Furthermore, if we look around for sources for suitable extensions to IDL, we find that popular OOA / D methods and the more formal specification languages provide specification facilities. The former (exemplified by the Unified Method, Coad / Yourdon, etc.) are more oriented towards being intuitively readable but have poorly defined meaning: that is, they are not amenable to mechanical processing at any significant behavioral level, and in particular, the all-important determination of conformance to a specification. The formal specification languages, (such as Larch, Z, VDM and their OO derivatives), are much better from that point of view, but less user-friendly. Importantly, however, some of the more advanced OOA / D methods such as Fusion, Syntropy, and Catalysis combine these virtues in different degrees, and provide the prospect of a convergence of the worlds of CORBA specification and OOA / D.

IDL++ and OOA / D can meet.

This paper illustrates how diagrammatic OOA / D methods and an IDL++ (this name is a placeholder for such a language — we make no complete proposal here) can express the same set of essential constructs in specification and high-level design. We also show how they can be given a common semantic basis in terms of a much more fundamental metamodel. This is important, because it provides the common terms in which tools functioning in any OOA / D method can read each others' component specifications.

We show how, based upon 'traits'.

We define the common semantic basis in terms of the very simple idea of 'traits' — individual named characteristics of a specification. Traits provide the required precision and extensibility for a metamodel across methods and notations.

Re-focus on the RFI

So, the RFI needs are…

We therefore contend that the OMG needs to hear a proposal which has the following characteristics ('traits'):

Extend IDL. Provides a **basis** for extending IDL to provide precise, mechanically processable, behaviors specifications.

Conformance. Defines behavioral conformance precisely enough to enable machine-processing, including search and selection of suitable components.

OOA/D notations. Defines a mapping from diagrammatic OOA / D notations to this formal basis, and possibly proposes an effect OOA / D methodology.

Refinement. Supports multiple levels of abstraction and refinement, from for example, from use-cases through high-level specs and IDL to code.

Practical. Offers a continuum from high-precision verifiable specification and design to informal fast-turnaround descriptions.

In addition, since both OOA / D and specification constructs will evolve as the technology evolves (e.g. towards mobile, intelligent agents), we have one more requirement:

Extensible. Facilitates extensions and interoperability across specification tools speaking different languages, in whatever terms they understand in common.

A short-term fix is not the answer.

Specifically, it is **not** sufficient to illustrate mappings of notations or terms to a common basis, without addressing the OMG's larger needs: namely, of extending IDL with machine-manipulable behavioral specifications and other OOA / D artifacts. We are setting standards now that will be hard to change in the future, and merely to cover the models we know at present would do a longer-term disservice to users of OOA / D methods as they evolve.

Outline of paper

We have adopted a layered approach to this paper, as outlined here by section:

1. Interoperability demands behavior specs in IDL++ and OOA / D methods.
2. Traits and objects: the semantic basis used for defining any specification construct
3. OOA / D and IDL++: key constructs defined in terms of traits
4. How we have addressed the RFI concerns
5. Concluding observations and summary

Note that the initial sections define fundamental constructs; they will necessarily contain formalisms. Most of these will be completely hidden from end users (e.g. designers, modelers), who will use the specification and modeling constructs provided by IDL++ or OOA / D diagrams in the higher levels instead.

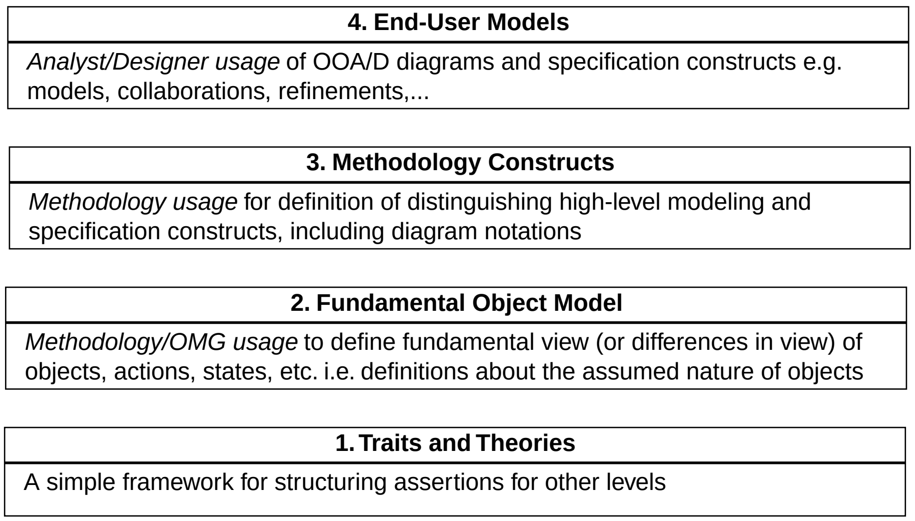
2 *The semantic basis: traits and objects*

THIS is the basis in which to define the mapping between IDL++ and OOA/D notations. It provides the extensibility and precision that we require in abstract descriptions of behavior. The main purpose in this paper is to expose the principle for discussion, rather than show exhaustive detail.

We use layered 'traits' to define a metamodel foundation.

The aim is to provide a number of layers, so that a wide range of powerful practical constructs can be translated to a small number of much simpler concepts. This is rather analogous to compiling programs into machine code: the simpler primitives are not practically usable; but the purposes here are (a) to ensure, by providing the translation, that we know what the powerful constructs mean; (b) to provide common terms into which different convenient notations can be translated; and (c) to provide a means whereby machines may process them automatically — again, rather like compilation.

This semantics are presented in approximately 4 layers as described here:



4. End-User Models

Analyst/Designer usage of OOA/D diagrams and specification constructs e.g. models, collaborations, refinements,...

3. Methodology Constructs

Methodology usage for definition of distinguishing high-level modeling and specification constructs, including diagram notations

2. Fundamental Object Model

Methodology/OMG usage to define fundamental view (or differences in view) of objects, actions, states, etc. i.e. definitions about the assumed nature of objects

1. Traits and Theories

A simple framework for structuring assertions for other levels

End-users will typically only utilize level 4(the top layer) to specify their systems. This section will cover levels 1 and 2, which are the foundation and are not dealt with by everyday designers. Section 3 covers the top two layers. As we will show, we can readily accommodate finer-grained layering because of the very simple fundamental model of traits.

2.1 Traits

A trait — the term is borrowed from Larch [Larch] — represents an individual specification fragment or requirement, usually with a name. It may have either an informal or a formal definition, or both. Typical traits include:

- Signatures: e.g.' trait 1= understands message square_root(x:Real)'

- Postconditions: e.g. 'trait 2= result*result==x'
- Invariants: e.g. 'trait 3= always maintains x>0'
- Temporal constraints: e.g. 'trait 4= does not allow access without login'

Traits can be exploited to effectively separate concerns. For example, 'trait 5=result>=0' may be another postcondition characteristic of square_root, separated from trait 2 as it is only of interest to some clients. Because each trait has a limited extent and scope, search tools can simply ignore categories of traits that they do not understand, and still make sense of the others. For example, if we had two candidate implementations, one of which guaranteed traits 1-5, while another guaranteed traits1-4, a search tool might offer one or both implementations to a client depending upon the traits which were significant for that client's requirements.

A trait has a unique name: it may be a URL — which makes it possible for a client and server on opposite sides of the planet to be sure that they are both referring to exactly the same requirement, which may be defined in a third place. As described below, this enables both the simple name-based matching of traits, as well as matching based on examining the definition of that trait.

Traits may be just simple names…

The simplest use of traits is based only on trait names. Every server advertises a list of provided traits, and every client lists it's required traits. A server conforms to any requirement that is a subset of the traits it provides. Searching for fully conformant servers in a directory is easy; if preferred, looser Bayesian 'percentage' schemes (as seen on Web searchers like Yahoo) can be used rather than strict matching. Simple trait-sets are useful in a limited domain where each trait is a name for a globally well-understood property. Modem interoperability is an example: the dialogue always begins with the devices passing the identifiers of their own traits, dealing with speeds and encodings.

…or formulae

In a more complex scheme, a trait can be a predicate formed from other traits. A server's provision (a trait) is met by a client's requirement (also a trait) iff:

provided_trait => required_trait (D 1)

Notice that there is no obligation that the provided trait must have been defined up front as a derivation from the required trait: all you have to do is establish that the predicate defining one implies the other. Of course, the verification is easier if one is derived from the other, as happens if the server was designed specifically for that client, and if provided_trait is simply defined as "required_trait & other_things_I_do". (See [Dhara] for a good discussion.)

Conformance must be checkable

Verification can be also done automatically if traits are limited to parameterized 1st-order propositions (using AND, OR, NOT or =>) with universal quantifiers, of the specific form:

for all x, trait(x)= for all x, y, z, P(x,y,z) => Q(x,y) & R(y,z) ...

It is much more powerful to be able to mix quantifiers ('for all', 'there is') arbitrarily. In general, this means that an automatic and complete verifier may not be feasible. So a library search tool may miss some suitable candidates, though it should never choose an unsuitable one. [Zaremski]

Traits are monotonic: no surprises down the line

An important characteristic of traits is that they are monotonic. If, as a client, you have found a server that provides the traits that you require, then you are guaranteed that, although it may also provide other traits in which you are not interested, none of those others will contradict the ones you know about. This arises naturally from the interpretation of conformance (D 1). Contradictory traits, like '(rick>20=>commit) & (rick>50=>abort) & (commit => not abort)', might not all be detected mechanically, but you will never find a server that conforms to such a trait.

Genericity in traits

Traits may be parameterized with values (and also, as we shall see, with types). This makes generic specifications possible. For example:

```
trait general_sq_root_accuracy_trait (range, accuracy, sqrt) {
for_all range, accuracy, sqrt;                    // introduce terms
range float & accuracy:float =>                   // assert type membership
general_sq_root_accuracy_trait (range, accuracy, sqrt) ==// define trait
    // range is arithmetic limit, accuracy is relative to parameter
    (     for_all x;      x:float &               // this trait says nothing about non-floats
          0<x &     x<range                       // allows local limit on arithmetic range
    =>    abs (sqrt(x)*sqrt(x) – x )< accuracy*x  // allows realistic result
    )
}
```

Now specify a function with several traits, utilizing this generic trait:

```
service my_sqrt_function(x:float)
    specification: general_sq_root_accuracy_trait (32000, 1e-4, my_sqrt_function)
                & for_all x, my_sqrt_function(x)>0
```

Genericity helps avoid non-monotonic schemes

(Restriction of properties like accuracy is sometimes used to justify non-monotonic specification schemes. According to these ideas, one spec states a capability without restrictions, but further down the line, another says 'but it only works if...'. This means that a client can never be sure that a server really provides the service as advertised. We much prefer the monotonic 'no surprises down the line' interpretation, as parameterized traits can always be used for such purposes. That way, a client is told explicitly where the restrictions may arise.)

Theories

Theories define contexts

A theory is a (universally) named body of knowledge about a particular topic, consisting of a collection of traits. A typical theory may describe the behavior of an object (its type); or the properties of primitive values, for example the well-known axioms of logic or numbers; or may be about an operation — for example that "<" is transitive and irreflexive; or about the commit / rollback behaviors of a transaction scheme; or about the real-time properties of a function; or may be about the semantics of programming or specification constructs. In our semantics for OOA / D, we interpret types, collaborations, frameworks, views, versions, all as theories.

Theories structure knowledge

Theories nearly always import others. In order to talk about Points, for example, you need to know about Real numbers, and so you import their theory. The resultant theory contains the traits of the importee, just as if they had been written in place. (Notice that importing is not subtyping: that comes later.) Theories thus form an acyclic graph. They can be thought of as traits, used for extra layering and structuring.

Theories introduce named variables, which may represent objects, functions, or types. A theory may define the behavior of a type by interrelating these things; or may extend a definition by providing further knowledge: for example, a theory of, say, Fibonacci numbers may be separate from (but import) the basics of integers.

```
trait Fibonacci NumberTheory ==
(    import IntegerBasics              // so that we know what an integer is
     introduce     fib(i)              // declare a new query taking 1 parameter
     axiom fibgendef== ( for_all i · i:Integer & i>2 => fib(i) == fib(i–1)+fib(i–2) )
     axiom fib1== ( fib(1) == 1 ) // '1' — a zero-parameter query introduced in IntegerBasics
     axiom fib0== ( fib(0) == 1 ) // same for '0'
     theorem  fib2 == ( fi b(2) == 2 )  // a theorem — follows from fib1, fib0 —
                                        // and definition of '2' imported from IntegerBasics!
)
```

Theories "encapsulate" all knowledge.

This rather esoteric example illustrates how the absolute fundamentals upon which a method is founded — what a number is, what an object is, what a state and transition are, and the consequences that follow from those definitions — can be dealt with by this simple structuring of logical assertions. In the sections below, we show how the more practical specifications of business objects can be rendered down to the same form. Although much too detailed to be dealt with directly by designers (just like compiled code), this form gives an assurance that those more powerful specifications have a well-defined meaning; and also provides the common means whereby tools can communicate them.

Inference

The axioms of a theory are the traits which define it. Other traits, which can be deduced from the axioms, are called its theorems.

A theory in itself can be thought of as a big trait. And a range of traits may be extracted from it, of the 'inference rule' form:

for all *introduced_variables* · *all_axioms* => *any_theorem* (D 2)

For example, in FibonacciNumberTheory we introduced the function fib. If you have any function corresponding to fib, that conforms to the axioms of FibonacciNumberTheory, then you may also believe the derived theorems about it. This is known as "theory export". It makes it possible to use a theory as a context within which to make inferences under a given set of assumptions; and then later to later apply the results wherever the assumptions are found to be true. In turn, such proof rules may be used to justify the theorems of other theories. (This scheme is derived from [Mural].)

Theories are used unconsciously.

Software developers make use of theories incorporating a standard body of knowledge, including sets, type membership, arithmetic, predicate logic, and other fundamentals. On top of this, they build (using higher-level tools and concepts, such as OOA/D constructs offered by methodologies or specification languages) specifications and designs, which each boil down to theories about object behavior.

What do traits do for us?

Traits and theories:

- Provide the basis of a precise language of requirements. This much is also provided by ordinary predicate logic. (We actually use three-valued logic, to cope easily with partial functions [Cheng].)

- Provide a notion of context. This is important, as we shall use it for any case in which what you are saying makes local assumptions: within the spec of a given system, within the meaning of a given language, within a collaboration of types.
- Provide a precise and relatively natural foundation for reasoning, using the extraction and application of proof rules. Even if reasoning is largely done informally — and automatically only within a restricted range — the foundation makes it possible always to go into such detail whenever doubt arises.

None of this shows directly at the user level: it is the engine under the hood.

2.2 Objects and Actions

What is our fundamental model of objects?

This is the next of our semantic layers. Here we define an underlying model of what an object is, and build up a practical means of specifying objects and describing their designs, ultimately defining a diagrammatic form of the familiar OOA / D variety.

Objects

We model an object as a map (symbol '→') from an object identity to a state; which in turn is a map from queries to other objects' identities. Note that the OMG's object model has never been formalized.

A theory of Objects

```
trait SimpleObjectTheory ==
    (   import     BasicMapsAndLists     // Defns of ×, → etc; see [Mural]
        introduce      OID, Query, ParameterName, QueryName, ObjectSpace
        axiom          ObjectSpace ==        OID → Query → OID
        axiom          Query ==              QueryName × OID-list
        axiom          Parameter-list ==     ParameterName → OID

        syntax         x.q(parameters) →     ObjectSpace(x)(<q, parameters>)
                       q(parameters) →       self.q(parameters)
    )
```

Our SimpleObjectTheory may be imported by every theory that defines object behavior; there is an example in "Interpretation in traits" on page 13, which shows how user-level type-definitions are interpreted. Other more sophisticated theories are possible — they may also be defined, and used where necessary.

The syntax lines define syntactic shortcuts, an interpretation of query invocations as more primitive non-object function calls over the object space.[1]

Queries abstract state

A query is an abstract attribute; it corresponds roughly to the implementation idea of an instance variable, or more accurately to the idea of a hypothesized read-only function. A query implies that there is certain information somehow associated with its object, though there is no specific suggestion of how. In particular, there is no obligation on the designer to actually implement a function or variable with that name: just

1. In a realistic system, the grammar for defining syntax would probably have to be a little more complex. However, this will do for now.

so long as the information is available somehow. Queries are used to describe a client's idea of the state of an object: and by the principle of encapsulation, this need not correspond directly to the real internal structure, provided the externally-visible behavior is nevertheless what the client is led to expect.

Queries can have arguments, which are themselves object identifiers:

query == name $\times$ OID-list

At this level, there is no strong difference between the receiver of the query and other arguments — so for example '5+3' can be thought of either as 5.plus(3) or plus(5,3).

By writing down traits, the queries can be interrelated:

for all m, n· m:Integer & n:Integer => m+(n.successor) == (m.successor)+n

for all h,r · h: Hotel & r: Reservation => (r.confirmed => h.hasRoom(r.dates))

Values

Primitive values, by the model we use here, are immutable objects[1] (this saves us having to invent a separate concept). '5' is a reference to a universally unique object. We may choose to think of the integers as having a successor query, with the property that each takes us to another integer we have not seen before. Other queries represent all the other relations between integers.

Actions

Actions define behavior

The externally visible behavior of an object means its response to actions. An action is an occurrence at the interface of an object: it is visible to objects outside. Messages are examples of actions; and any sequences of actions, for example making up a dialogue, may also be considered an action. [2]

The simplest action is one that just returns a result to whoever invokes it. Just like a query, except that an action must be implemented by the designer.

An action may cause a change of state — that is, a change in the result of applying some query to a participant. (Queries themselves never change objects, by definition.) An action has a number of participants — the objects whose states may affect and be affected by the action; two of them may be distinguished as 'receiver' and 'sender'.

Actions model all levels of object interactions

We use the idea of action to model any interaction between any set of objects — a system and its external users; or a collaborating group of objects in a design. At an early stage of design, a single action (like 'customer obtains cash from ATM') may be used to model an interaction that is later **refined** into a dialogue of smaller actions ('customer inserts card; ATM asks for PIN;...')[3].

1. We could also model "values" as a distinct category, but omit this option here.
2. The notion of an action could also be defined formally.
3. In the general spirit of "Use-cases"

Actions can be refined

There are two important kinds of action refinement. In one case, the more detailed actions involve all the original participants: it is the mutual dialogue that is being refined. In the other, only one of the participants executes a series of actions, privately from the others: in other words, a method servicing a message. This distinction makes it plain that the concern of a specifier is to abstract away, not just from internal structure, but also from the detail at the interface. When they first invented ATMs, they had the general idea, and the precise dialogue was evolved later. Conversely, it often makes sense to write down high-level internal structure even before sorting out the details of external dialogue.

Every action can be specified

In every case, the action may be characterized by an 'action spec': that is, a description of the effects of the action in terms of its effect on the states (queries) of the participants. This specification can be done independently from any implementation. The simplest action spec is one with a postcondition: a predicate relating the states immediately before and after the action. Although the action will take place over a period of time, a postcondition states only this relationship, and nothing of how it is achieved.

Specifying basic object behavior

Actions define object behaviors

An object's behavior can be described by a set of action specs and other traits, written in terms of the queries. So if an object representing a point on a plane is modeled with a pair of queries x and y, an action spec could be written:

```
{ true :– x–old(x) == dx & y–old(y) == dy} move(dx,dy)
```

Pre / Post pairs specify an action

In this notation, {pre :– post }action is a trait asserting that the action conforms to the postcondition under any circumstances in which the precondition is true in the prior state. (When the precondition is false, the postcondition just does not apply: this spec says nothing about the outcome. But there may be other specs of the same action which do apply, defining either normal or exception n behaviors.)

Action Theory

The underlying semantics of objects can be specified as well.

Action-specs must be interpreted in a context in which the idea of a previous state of every OID is understood. A proper object-history treatment of this can be quite complex, though again, this is below the user level; see [Utting] and others. For the purposes of illustration, a simplified version might be sketched as follows. (Skip at first reading!)

```
trait ObjectActionTheory = SimpleObjectTheory & (
    introduce Action, ObjectHistory, ActionInvocation
    axiom    ObjectHistory ==        ObjectSpace-list        // from SimpleObjectTheory
             // an object history is a series of object spaces;
             // an event history is a set of actions linking pairs of states in an ObjectHistory
    axiom    EventHistory ==         ActionInvocation × startStateInde × endStateIndex
    axiom    ActionInvocation ==     ActionName × arguments:OID-list
    axiom    ActionSpec ==           ActionName × precond[old: OID-list] ×
                                         postcond[old:OID-list, new:OID-list]
    axiom    conforms (ActionSpec(an, pre, post),
                     ActionInvocation(an, args), startState, endState) ==
             // conformance of an indivvidual action invocation to a spec means:
                 ( pre[startState] ⇒ post[startState, endState] )
    axiom    conforms (ActionSpec(an,pre,post), eventHistory, objectHostory) ==
             // conformance of an entire history of events and objects to a spec means...
```

```
                (<invocation, startStateIndex, endStateIndex> : eventHistory ) ⇔
                // ... that any invocation is only in the event history if it conforms individually:
                conforms(ActionSpec(an, pre, post), invocation,
                        ObjectHistory[startStateIndex], ObjectHistory[endStateIndex])
        )
```

It is this theory that type definitions implicitly import. Again, a different idea of what an action spec means can be written as a different theory. The power of traits is that very little is bolted into the language.

What do models based on actions and queries give us?

- Queries abstract away from the internal structure of a design. They give the specifier a way of describing the required effects of the actions, without commitment to any design. There may be many designs that meet one specification.
- Actions abstract away from the details of dialogue, so that behavior of individual objects and collaborations between them can be discussed at a high level.

3 *OOA/D and IDL++: a common base*

"User-friendly" notation also needs semantics

FINALLY we arrive at a level seen by users. Users draw diagrams to model the systems they build and the interfaces they require or conform to; and they generate IDL++ specifications of the components they design. Diagrams make the designs very readable, but we must ensure that they have a clear meaning in terms of more primitive constructs — partly so that they can be translated to mechanically processable form. This is what we illustrate here. Although these examples are cast in terms of Catalysis, the same principles can be applied to any well-designed OOA/D notation.

We provide examples for OOA/D and IDL++

We will concentrate on what we believe to be the most important aspects common to all good analysis and design notations. For each aspect, we will discuss: the need for it; a definition; its realization in IDL++ and interpretation in traits. The same for a diagram-based notation. Even visual aspects can very easily be modeled formally.

The aspects illustrated here are:

- Specifying behavior — of components and systems.
- Designing by collaborations — between components within a system, and between external users and systems or elements of a business
- Abstraction and refinement — conformance and subtyping
- Generics — generic types and framework collaborations

3.1 Specifying behavior

The need

Corba-IDL defines an interface as a set of typed signatures. The idea is that any client who wants to use the services defined by that interface can utilize any object which implements that interface. However, signatures themselves are woefully inadequate to describe the expected behavior of the server object. For example, if a client implementing a DrawingEditor wishes to edit Shapes from different sources, we might define:

```
interface Shape {
    void move (Vector to);
    void resize (int percent);
}
```

One implementation might interpret the “to” parameter as a “delta” from its current position, another might interpret it as an absolute position in the host windowing system’s coordinates, and a third might assume that it can only be moved when it is on the top layer of a multi-layer drawing.

The problem becomes greatly exacerbated as we move towards the OMG’s vision of plug-replaceable components which could be provided by any vendor, and, eventually, could even be provided dynamically by some brokerage agent from among all available objects on the Worldwide Web.

Therefore, we must support specification of the effect of an operation, the conditions under which it might be legally invoked, etc. In addition, we might need to specify outgoing messages (or, in some cases, changes to another’s model) required.

Definition

Type: a set of objects which conforms to a specification of behavior. Externally-visible behavior is the only criterion for type membership: not internal structure, nor inheritance from a given supertype, nor even having been declared to be a member of a particular type. Whereas an object is created as a member of a particular class, it is a member of any type to whose specification it conforms; whether or not that was intended when it was first designed, it can be used as a member of those types.

A type can be specified by a set of behavioral constraints, including ‘action specs’ (typically pre / postconditions) on individual actions, and overall temporal constraints on groups of actions. As noted in our basic definition of objects, action specs are normally written in terms of a model consisting of abstract query-functions.

Type Model: the set of queries used to characterize that type.

IDL++

A type can be used both to specify a complete system or an individual component interface. Here is an example in IDL++. We model the general idea of a Shape by saying that it either covers any given point in the plane, or it doesn’t. This allows us to abstract away shape variations with no loss of precision or generality. Note that although the query covers is a function, we don’t have to define it: it is the job of anyone who claims to have designed a Shape to define the meaning of cover, although even they are not required to make it available to clients or even to implement it.

IDL++

```
type Shape {        // The keyword type to add semantics to an IDL interface
model:
    boolean     covers   (Point p);      // General idea of a shape: covers some points
interface:
    void move (Vector to);
    void resize (int percent);
action specs:
    move(Vector to) { true :– for_all p · old(covers(p)) <=> covers(p+to))
    resize(int percent){
        percent>0 :–                        // precondition
        for_all p · there_is centre ·       // postcondition
            old(covers(p-centre)) <=> covers((p–centre)*percent/100)}
invariants: // any additional invariants, besides query types, would be here
}
```

Decisions made vs. deferred.

This description specifies the guarantees and assumptions of services, and the model which defines the vocabulary for describing that contract. This model might include both an abstract view of "exclusive" private state of the object, as well as an abstract view of its connections to other (possibly shared) objects (like the Editor itself, so it could call-back with the "damage rectangle" information for re-painting). It defers decisions about the implementation of methods and data representation.

Interpretation in traits

The interface can be translated to a Trait fairly directly, in which the type, the variable self, and the names of the queries and actions are declared. The other aspects — signatures, invariants, action specs — will become subtraits. The outer trait says that membership of the type is equivalent to conformance to all of the subtraits.

```
trait ShapeTrait = ActionObjectModel &
            PointTrait & VectorTrait & BooleanTrait & IntegerTrait & (
    (   introduce Shape, covers, move, resize ·
        Shape ⊆ OID & covers:QueryName & move:Action & resize:Action
         for_all self ·
            self:Shape <=>
                ( ...the subtraits ... )
    )
```

Let's take this apart:

```
ShapeTrait = ActionObjectModel
            & PointTrait & VectorTrait & BooleanTrait & IntegerTrait & (
```

None of what we want to say would make any sense unless we know what a Point is, what a Boolean is, and so on: so we import those traits into ShapeTrait: so we say that if you know ShapeTrait, then you know all there is to know from PointTrait etc. as well — and then we add another piece of our own in the lines that follow. (Some of these are probably redundant — for example BooleanTrait probably is already imported along with the others.)

These imports can be made explicit in the form of **capsules** [Fresco]. A capsule can contain any combination of code and specifications: a system is a composition of capsules. The structure of theories and proof system in which proof rules may be constructed from enclosed theories comes from [Mural], a tool for building and structuring proofs.

SimpleObjectTheory is imported so that we can refer to the underlying semantics. **Notice that if we wanted to use a different underlying model of what an object is, we would merely have to import it. There is no problem with different object models co-existing, and none built into the underlying framework.** (Lossy) translations from one to another could be defined so that a search tool (for example) could make some sense of specifications defined upon, say, a temporal model. And again, if trait names are quoted as URLs, then anyone can look up an unusual model as required.

```
( introduce Shape, covers, move, resize ·
```

These names are 'declared' here, and will be recognized in any trait that imports ShapeTrait; just as PointTrait introduces Point and the operations on it. If any of these names is already introduced (for example, PointTrait might introduce move), then the new introduction is redundant.

```
Shape ⊆ OID & covers:QueryName & move:Action & resize:Action
```

These define the broad categories to which the names belong.

```
for_all self · self:Shape <=> ... the subtraits ...
```

If ever you meet any object — call it 'self' for now — that claims to belong to shape, then it will conform to the subtraits (see below); and vice versa. (When you meet one, it will have its own name, but we are saying this applies to them all, and self is just our own name for any object you apply this to.)

So now let's look at the subtraits, that define what the characteristics of this type are.

Query signatures each produce a subtrait like:

```
for_all p · p:Point => (self.covers(p)):Boolean
```

Each action spec is a trait. We just have to slot it into place, taking care to put in 'self' wherever appropriate.

```
self.move(to) { to:Vector & self.inv
                :–self.inv & for_all p · old(self.covers(p)) <=> self.covers(p+to) & to:Vector)
```

The parameter type has moved to the precondition. "object:Type" is a predicate testing type membership. If there was a return type, we would type this in the postcondition.

Any invariant is added as a conjunct of every pre and postcondition: it is something that should be true both before and after every operation. This particular example has no explicit invariants besides the signature types of queries, so we've written inv where it would appear.

Notice that this is a statement about what 'self.move' means, *under the assumption that self is a member of Shape.* So if there happens to be a definition of x.move where x is, for example, an Applet, then that definition can also be true. We don't have to think about move as a separate action: just one that does different things when applied to different types of object. And in some cases, for example where there is a spec of the same operation in a subtype, both specifications will apply.

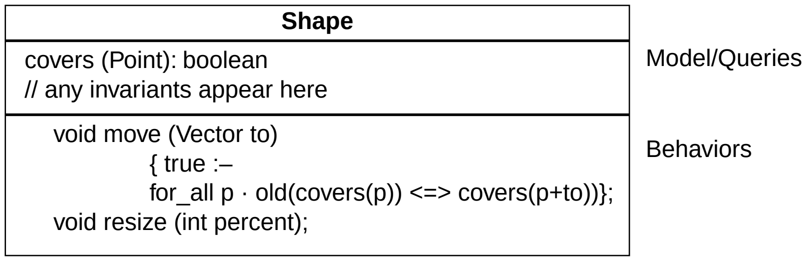
Discussion

By doing this, significant portions of CORBA interface descriptions could be made less ambiguous and verbose. By using abstract queries as illustrated here, services could be specified with no loss of either precision or generality. Specification of exceptions raised are also easily handled[1].

Diagram-based models

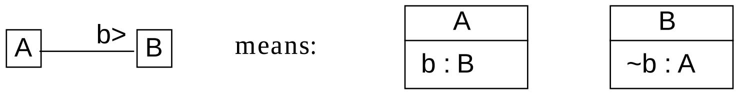
Diagram-based notations can be given a precise meaning by rendering them into traits. We will illustrate this with some of the basic object modeling notations from Catalysis; the same approach should be possible with other notations.

Firstly, presenting a type in a box is no great semantic feat, and has exactly the same interpretation as the IDL++ given earlier:



```
covers (Point): boolean
// any invariants appear here

    void move (Vector to)
            { true :–
            for_all p · old(covers(p)) <=> covers(p+to))};
    void resize (int percent);
```

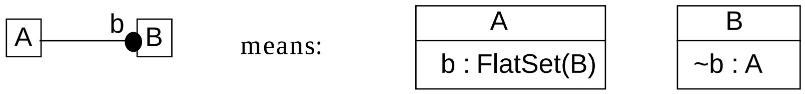
In Catalysis, associations are equivalent to queries: for us, the difference is only a matter of presentation. This may be different in other notations, and would have to be represented in their translation to traits.



There is an implicit reverse name ~b for the object from which b gets to this object:

∀ x:A, y:B; x.b.~b==x and y.~b.b==y

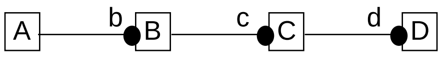
The meaning of these separated type-definitions can in turn be translated to traits as illustrated previously.



If x is a member of A, then x.b is a set of members of type B; and again, there is an implicit reverse link.

FlatSets are an interesting and very useful variant of sets. The key property is that a flatset never contains other flatsets, but just their members; and applying a query to a flatset yields a flat set of the results of querying the members. So if we have a chain of

1.The conventions used in ADL [ADL] could be conveniently adopted to this end.

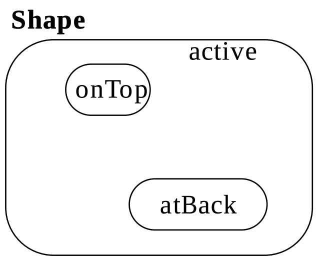
types  and if an member a of type A, then

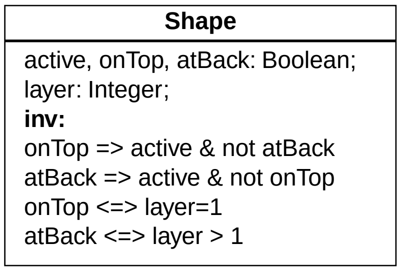
a.b.c.d is a (flat) set of D. This makes it very easy to navigate around models within postconditions etc., and supports more intuitive query expressions than ODMG-93.

Other variations on the association and the context in which it is used are possible. An optional link represents a query which may take the value NIL; relations of multiple arity may be represented, or where lifetimes or visibility are restricted. In all cases, it is essential to give a definite meaning in terms of the underlying simpler model. It is not sufficient to explain the usage and syntax. And though it can be helpful to give a 'metamodel' which models the notation in its own terms, this is not sufficient as a sound basis — particularly not as a medium of interoperability.

State Diagrams

In our model, states are simply boolean queries. A state diagram simply adds some invariants to those boolean queries, as outlined below. Using invariants, each state is further defined as a predicate on the type model based on the state structure (Harel state-charts). State transitions then simply become a visual presentation of actions, with very clear semantics even for complex objects.





Shape

active, onTop, atBack: Boolean;
layer: Integer;
inv:
onTop => active & not atBack
atBack => active & not onTop
onTop <=> layer=1
atBack <=> layer > 1

3.2 Collaborations

The need

Much of object-oriented design is about division of responsibilities and the specification of collaborations between objects. OOA / D methods pay careful attention to this, though many only produce informal or example-based models. IDL permits us to define the signature of an interface. Yet, most interesting IDL services (e.g. transaction service, relationship service) define sets of interfaces and interactions i.e. abstract architectures. The signature of each interface is defined formally, but the expected joint collaboration and sequences of interactions between the objects is defined in lengthy prose (e.g. see Section 10.2.1: "Typical Usage" of the Transaction Service).

Therefore, IDL++ must support defining "open" architectures [PSL]. It should capture the decisions being made in the OO-design (and not precipitate any decisions that are not implied by the OO-design). It must have rules for conformance i.e. the basis for determining if any claim to implement this architecture actually does so; equally, it could help define test suites for determining conformance. In order to specify collaborations, it must include specification of both incoming and outgoing invocations for each interface.

Definitions

Collaboration. A collaboration is a set of actions between objects playing certain roles, with a temporal constraint and a set of queries defining a joint type model. It does not necessarily specify senders and receivers for every message send, as long as the permitted sequences of interactions is adequately described.

Role. A role is a place in a collaboration. Any object that plays that role must support every interface required of that role within that collaboration.

IDL++

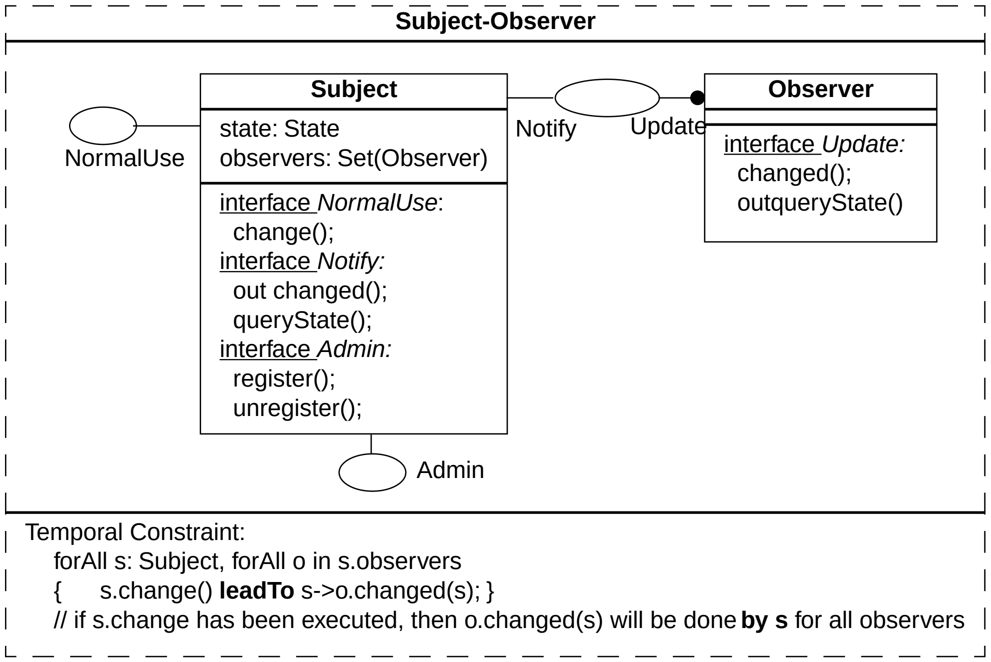
```
collaboration Subject_Observer {
// the roles in the collaboration
roles: Subject, Observer;
model:    // this is an abstract type model for the collaboration
          // no specific role in the collaboration need “know” any part of the model
    // there is some state query on Subject
    Subject.state: State;
    // there is some set of Observers for each subject
    Subject.observers: Set(Observer);
interfaces:    // these define the interfaces on each role
    // each interface is a list of incoming or outgoing messages
    Subject.NormalUse= {change()};
    Subject.Administer= {register (Observer), unregister (Observer)};
    Subject.Notify = {out changed (Subject), queryState()};
    Observer.Update= {changed(Subject), out queryState()};
protocol:

// synchronous version
    forAll s: Subject, forAll o in s.observers
    change() { true :-        o.changed(s) }
    // if s.change has been executed, then o.changed(s) has been done for all observers

// synchronous, with more specific senders of messages i.e. make more design decisions
    forAll s: Subject, forAll o in s.observers
    s.change() { true :-      s->o.changed(s) }
    // if s.change is done, then o.changed(s) has been done by s for all observers

// details elided here: when o.changed happens, o->s.queryState must follow

// asynchronous version: differentiates sent vs. reply, uses leadTo for temporal rules
    forAll s: Subject, forAll o in s.observers
    { sent (s->reply(s.change()(OK))
      // s has completed change request with OK
        leadTo // must lead to
        sent s->o.changed(s);      // s has sent (oneway) messages to all observers
    }
}
```

Decisions made vs. deferred

This model specifies roles which must exist; incoming and outgoing messages for the interfaces of those roles; queries needed to describe abstract joint state; permitted sequences of messages for a certain logical thread of interactions (noting that there might be otherwise unrelated logical threads which cause interleaved messages). It defers precisely which role is the sender and receiver of that message, and single vs. multi-threading servers playing each role.

OOA / D Diagram

This diagram represents the design of a collaboration between objects in different roles [Catalysis]. It corresponds quite closely to the notion of “Use-Case” [Jacobsen] but establishes semantics to allow refinements such as from joint actions to localized actions. Other diagrammatic variants exist in other methods, and looser versions appear in OMG documents (e.g. Fig. 10.1 in Transaction Services). The temporal constraint itself can also be depicted as a diagram[PSL].



Underlying Semantics

A collaboration can be interpreted as a theory containing traits about actions involving its participants.

In Catalysis, we can specify actions not yet localized to specific receivers, deferring decisions about responsibility while still defining the resultant effects. These joint actions may affect the states of any of their participants, represented by the ends of the action links (on the ellipses). Apart from this, the action specs are just understood as traits, as in the single-type case. The temporal constraint is also just interpreted as a trait.

This simplistic semantics — just take everything in the collaboration and wrap it up in a theory — highlights a property of the traits semantics. For it might be asked, what happens about the special operators like **leadsTo**? Do they not have to be translated in some way? How are they interpreted in terms of some deeper model?

The answer lies in the nature of our object model. As it stands, it is not rich enough to give meaning to temporal operators; to do so, we would have to augment it to include a proper notion of object histories. The beauty of the traits approach is that in order to interpret temporal logic, we need only import the appropriate theory about object histories. Any client unaware of this more sophisticated basis can ignore references to it.

This shows that because all the semantics are defined within the alterable traits framework, no future development in specification methods will demand a fundamental change in this scheme.

Discussion

In addition to illustrating how a powerful construct from OOA/D [Catalysis, Jacobsen] and specification [PSL] can be mapped to our underlying semantic basis, we also recommend this specific construct for IDL++. By using it, we believe that significant portions of Corba services could be made less ambiguous and verbose with no loss of either precision or generality.

3.3 Abstraction and Refinement

The need

During a development process, abstract models let us commit to certain decisions and defer others at various levels of refinement. IDL commits to interface signatures and defers implementation of data and methods[1]. OOA/D methods permit us to defer decisions such as distribution of responsibilities, finer-grained interaction dialogs, concrete messages and signatures, etc. A clear notion of exactly what decisions have been made helps us understand the process of refinement; we can thus define acceptable and unacceptable refinements i.e. **conformance.** It also helps define adaptors or wrappers required between existing components or models.

Having an underlying semantics in terms of traits gives us a very precise notion of refinement: no statements made at a more abstract level may be contradicted in a any implementable refinement.

Definitions

Refinement. A relation between models. If model A is refined by model R, then any component conforming to R will behave according to all valid expectations of a client who knows A.

Inheritance. A refinement in which the model R has been defined in terms of model A, e.g. an IDL derived interface, or a conjunction of named traits. Any correct implementation of R is also a correct implementation of A.

Conformance. A refinement in which the model R has been defined separately from model A, and conformance has to be established before a correct implementation of R can be considered a correct implementation of A. We need a mapping between the models, called a “retrieval”, to establish conformance of the form:

```
R & retrieval => A
```

IDL++

Inheritance: using IDL++ derived types:

```
type Circle: Shape { … } // all traits of Shape are conjoined with explicit Circle traits
```

Conformance: where Circle was defined independently of Shape.

```
type Circle {                // Shape traits are not relevant in this Circle definition
model:
    int radius;
    Point center;
```

1. IDL, with the CORBA infrastructure, also abstracts language, distribution, etc.

```
actions:
    move (Vector v) { :- center == old(center) + v }
    resize (int percent) { percent>0 :- radius == old(radius) * percent/100 }
}
```

Circle is still a valid member of Shape, once we establish the retrieval relation:

```
refinement Circle -> Shape {
retrieval:
for all c: Circle, Point p, c.covers(p) = |p-center| < radius;
}
```

Decisions made vs. deferred

In this example, defining Circle independently of Shape has the advantage that the most convenient model may be used for each of them. Thus, the relevant queries can be chosen separately. In general, it allows separately defined specs to be related.

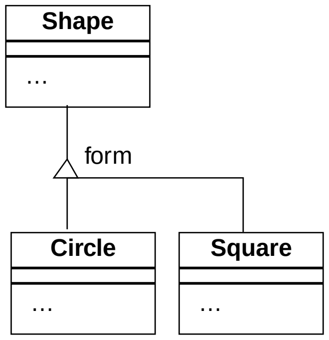
OOA/D Diagram

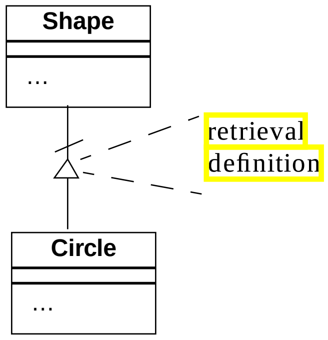
This diagram illustrates the two forms of refinement using Catalysis notation. The normal “subtype” version on the left defines a trait with exclusive subtypes:

```
trait shapeForm ==
(for all s · (s: Circle => s: Shape) & (s: Square => s: Shape) & not (s: Circle & s: Square) )
```

The conformant “**subtype**” version on the right uses the “/” to indicate a derived refinement relationship between the two types themselves. Its semantics are:

```
circleAsShape = (forall s · s: Circle & circleShapeRetrieval => s: Shape)
```





Discussion

By formalizing the semantics of a model, it becomes much easier to define refinement accurately. We can now guarantee that if any implementation correctly meets its specification; if that specification is a valid refinement of a more abstract model, then any assumptions made based upon the abstract model will not be violated. There are numerous other very practical and useful forms of refinement, some of which are used in the Catalysis method [Catalysis], including refinement of collaborations, of types, and of signatures.

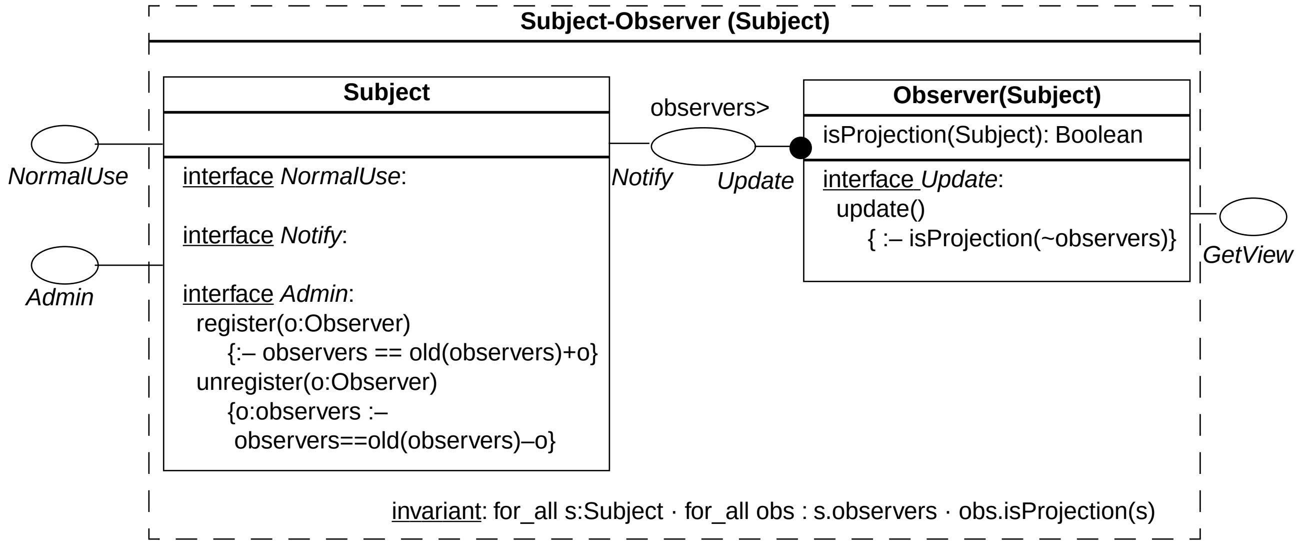
3.4 Generics

The need

A collaborations, like the Subject / Observer shown above, is a typical example of a "pattern". However, such collaborations can appear in a multitude of contexts with entirely unrelated sets of object types playing the roles. For example, Subject / Observer might be Worker / Manager, or Switch / SwitchDisplay. i.e. we can turn it into a framework — a generic collaboration that can be instantiated to a specification that can be applied to any example.

OOA / D diagram

First let's generalize the example slightly. Subject is now a parameter, and Observer is itself defined as a generic type parameterized by Subject. GetView is an interface by which the Observer might be queried by an outside client.


Subject-Observer (Subject)
Subject
interface NormalUse:
interface Notify:
interface Admin:
register(o:Observer)
{:– observers == old(observers)+o}
unregister(o:Observer)
{o:observers :–
observers==old(observers)–o}
NormalUse
Admin
observers>
Notify
Update
Observer(Subject)
isProjection(Subject): Boolean
interface Update:
update()
{ :– isProjection(~observers)}
GetView
invariant: for_all s:Subject · for_all obs : s.observers · obs.isProjection(s)

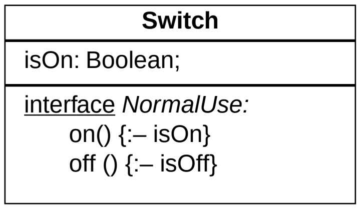
The goal of the collaboration is to maintain its invariant, which says that every subject will be correctly presented by all its observers. The query isProjection in Observer tells whether self is an accurate presentation of the Subject. The definition of this query will vary from one observer to another. But notice that calling update has the effect of making it true — how this happens would be up to the designer of the specific Observer implementation for the specific isProjection query variant.

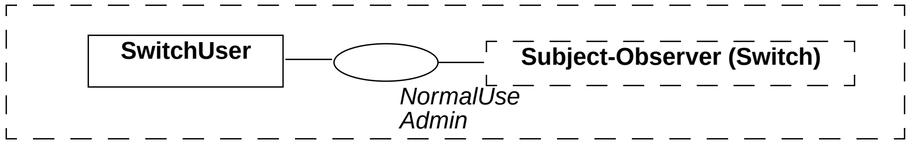
Both the collaboration and one of its types are parameterized — which means that the traits to which they translate are similarly parameterized. Filling in the arguments results in a specific set of traits.

Why would it not be sufficient just to subtype Subject and Observer? Because each specialization of Subject must be connected to Observers that will work with that type: it would be no good to refine Subject to BankAccount and have RailRoadSwitch-Display as its observer. They must conform as a collaboration, not individually[1].

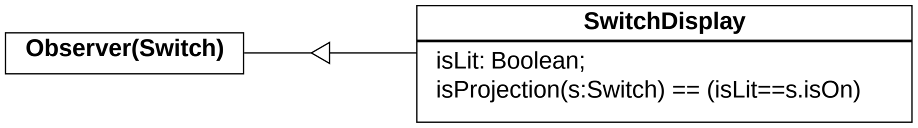
1. The fact that they might be implemented by subclasses is entirely orthogonal.

Here is one possible kind of Subject:





Now what is the full set of actions defined on type Switch? From the box called Switch, we have a set of traits telling the effect of sending the on and off messages to any member of type Switch; and also, from the specialization of Subject-Observer, we are told what happens if you send the register and unregister messages to a Switch. So a designer implementing a switch would have to satisfy them both — including the link to a set of Observers; and the collaboration invariant. Similarly, the designer of any Observer(Switch) must meet the specification of update, defined in terms of isProjection. There may be many variants, each a subtype of Observer(Switch), and each defining their own criterion for being a correct projection; for example:



How will the designer of Switch ensure that the collaboration invariant is met? If all he knows is the generic collaboration, then there is only one guaranteed way — to call the update function on all observers: it is guaranteed by its spec to have the desired effect, and we know that any Observer(Switch) – such as SwitchDisplay – will be obliged to implement it.

So the Subject-Observer framework defines the common attributes and behavior required to make a suitable pair of types work together as a collaboration.

Collaboration invariants

Notice that the invariant belongs to the collaboration, not either of the types. This means that as far as anyone outside the collaboration is concerned, it is always true; but from within, it may sometimes be false. In particular, it may be false after a change has been made to the Subject, but before all the Updates are complete. The invariant applies before and after all calls on the NormalUse and Admin interfaces, but not the internal Notify and Update interfaces.

This shows in our semantics for translation to traits. In the description of invariants above (page 13), we said that they were interpreted as conjuncts of the pre and post-conditions of every operation. That is true for the invariants of a self-contained type; but for a collaboration's invariant, this is done only for the operations visible outside the collaboration. If there is any operation used from both within and without the collaboration, then we can make two versions of its action spec: one with the invariants added pre and post, and one without; in other words, if the invariant is true beforehand, it will be preserved; if not, the operation will work anyway.

IDL++

(We omit a discussion of an IDL++ variant of this scheme — it is a question of choosing a suitable concrete syntax after the manner of previous sections.)

Discussion

Although we have not explicitly shown a translation to traits here — at this level it becomes long and tedious — we have obtained a clear understanding of what the generic collaborations and their specialization means by considering the translation.

Another essential aspect of generics is that constraints may be applied to the parameters — for example to constrain a type to have a "<" operation with suitable properties. In this system, it can be done by adding to the collaboration invariants to that effect, such as:

for_all a,b,c:ParameterType · a < b && b< c ⇒ a<c

This ensures that any supplied argument type must have this property: if it has a contradictory property, it will turn out to be impossible to implement the collaboration. (This is similar to the use of the separate "theory" construct in [FOOPS].)

4 *Summary to the OMG's RFI*

We end with a summary of how our approach addresses the RFI's primary points. The table below summarizes our reading of the main thrusts of the RFI, and relates each of them to the ideas and information presented in this paper. It is recommended to read this summary in conjunction with our discussion in Section 1 about the OMG needs underlying this RFI.

RFI Issue	How it is addressed by this paper
Re-use of OOA/D workproducts	We provide 4 key features for re-use and interoperability across levels of abstraction, methods and tools. • Semantics: the underlying trait semantics can capture different methods; any one of the suggested schemes for interoperability may be used, from name-based traits to sophisticated matching • Composability: workproducts from one method can be composed (using one method's 'composition' operator) with workproducts from other methods, with well-defined behaviors in terms of the traits • Conformance: workproducts are re-used with full confidence in conformance; decisions specified at one level of abstraction will not be contradicted later • Abstraction and refinement: with generics, collaborations, types, and 'capsules', we provide for re-use of and very incremental development, from the level of business models, through architectures and frameworks, to code, and, in fact, on to maintenance and enhancement
Development Lifecycle	This point was very unclear to us: is it requesting a description of the process model for a particular methodology, or a meta-description of lifecycle models in general? Assuming the former, we might describe the levels and workproducts as below. We imply no ordering at all between levels and activities, but we do require formal or informal refinement relations between descriptions. Our basic cycle is, recursively, specify an abstract component; design its internal collaborations (from the level of business down to code). • Domain or business models: collaboration model of interacting actors, utilizing abstract actions to concrete refinements; describes existing *business design* • System context: a collaboration model of the "to-be" domain, with the target (changes to) component(s) as a part of it; *design* the desired interactions • System specification: a type model and behavior specification for the target component(s). • Design: refine external actions to sequence of actions between roles and collaborations within the system.
Semantics and meta-model	Our foundation of traits and theories together provide a very strong basis for defining semantics. As outlined in the beginning of Section 2, everything from the most basic definition of what our model of objects is, through modeling constructs layered on it by a specific methodology, to end-user models built on some (combination of) methodologies, has a well-defined meaning. We outlined the mappings for some representative OOA/D constructs, and list some other comparisons below. The mappings clearly covered "static" semantics, dynamic and functional semantics, and some syntax in terms of IDL++ and OOA/D diagrams.

RFI Issue	How it is addressed by this paper		
Method projections	We list below the primary constructs we presented, together with the methods and specification languages that have a close formal relation, or a loosely defined relation to it. Note that several methods, either due to poorly defined semantics or inadequate understanding on our part, do not map easily.		
	Construct	**Formal Match**	**Loose Match**
	Contracts and “model” queries	Syntropy, Catalysis, BON, VDM, Larch, Object-Z	Fusion
	Collaborations	Design Patterns, Catalysis, OORAM, DisCo, PSL, CCS	Syntropy, ROOM, Shlaer&Mellor, Unified Method, Fusion
	Abstraction & Refinement	Catalysis, PSL, ROOM, VDM, Larch	Objectory, Unified Method
	Generics	Catalysis, Larch, Z, VDM	Unified Method
Tool Projections	Some of the concepts and notations are currently supported by the Protosoft tool. We do not have any details on concrete interchange facilities, but we have detailed the semantic scheme that should underlie such an interchange. In general, it would consist of: • Traits by name (perhaps using URLS, optionally reasoning about trait definitions) • Concrete syntax: e.g. based upon CDIF syntax • Visual information: this can easily be formalized in terms of traits, including “visual” semantics		

Summary

There have been two main thrusts to this paper:

- Traits
  - give a pluggable mechanism in which a wide variety of requirements can be described in common terms — even down to altering the fundamental object model
  - provide a structure in which individual requirements can be separated from others — for searching and conformance checks
- Requirements can be expressed equally abstractly and precisely
  - in an extended IDL, suitable for machine processing of interfaces to components in open and component-built systems
  - in (soundly-based versions of) diagrammatic OOA / D notations

We have illustrated these principles by giving translations from examples of these notations into traits; and by giving an account of how conformance can be checked between traits.

5 *References*

[ADL]: Sriram Sankar and Roger Hayes, “ADL – An Interface Definition Language for Specifying and Testing Software”, Sun Microsystems Laboratories, Mountain View, CA 1994.

[Booch]: Grady Booch, “Object Orientated Analysis and Design with Applications”, 2nd edition, Benjamin Cummings, 1994.

[Catalysis]: D D’Souza & A Wills “Extending Fusion — practical rigor and refinement” in *OO Development at work: Fusion in the real world* ed Malan, Letsinger and Coleman, Prentice Hall 1996. Also http://www.iconcomp.com; a complete book on Catalysis is currently being published.

[Cheng]: JH Cheng and CB Jones. On the usability of logics which handle partial functions. In C Morgan & J Woodcock, eds, *Proc. BCS-FACS 3rd Refinement Workshop* Springer 1991
Also http://www.cs.man.ac.uk/csonly/cstechrep/Abstracts/UMCS-90-3-1.html

[Larch-IDL]: GT Leavens and Y Cheon “Extending CORBA IDL to specify behavior with Larch.” OOPSLA ‘93 Workshop Proceedings: specificaion of behavioural semantics in OO Information Modeling, pp 77-80. Also ftp://ftp.cs.iastate.edu/pub/techreports/TR93-20/TR.ps

[FOOPS]: Paulo Borba and Joseph A Goguen. On Refinement and FOOPS. http://www.comlab.ox.ac.uk/oucl/publications/tr/TR-17-94.html

[Dhara]: KK Dhara & GT Leavens, Forcing behavioral subtyping through specification inheritance. In 18th Int COnf Software Eng, 1996. Also TR95-20a, Iowa State U. ftp://ftp.cs.iastate.edu/pub/techreports/TR95-20/TR.ps.Z

[Fresco]: Fresco. Application of formal methods to OOP, Alan Wills, PhD thesis Manchester University, UK. 1991. Also ftp://ftp.cs.man.ac.uk/alan

[Fusion]: “Object Oriented Development, The Fusion Method”, Derek Coleman, Patrick Arnold, Stephanie Bodof, Chris Dollin, Helena Gilchrist, Fiona Hayes, Paul Jeremes, Prentice Hall, 1994.

[Jacobson]: “Object Oriented Software Engineering: A Use Case Driven Approach”, Ivar Jacobson, Magnus Christerson, Patrik Jonsson, Gunnar Overgaard, Addison-Wesley, 1992.

[Larch]: Larch in Five Easy Pieces. Guttag et al 1986. DEC TR
Also http://larch-www.lcs.mit.edu:8001/larch/ — Larch home page.

[LiskovWing]: B Liskov and J Wing, A behavioural notion of subtyping. CACM 1994

[Mural]: Mural, CB Jones et al, Springer 1991.
Also http://www.cis.rl.ac.uk/proj/mural.html

[OMT] . “Object Oriented Modelling and Design”, James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen, Prentice Hall, 1991.

[PSL]: PSL, Doug Lea. http://g.oswego.suny.edu

[Rumbooch]. “A Unified Method for Object-Oriented Software Engineering”, Grady Booch, James Rumbaugh, Documentation Set Version 0.8, Rational Corporation, Santa Clara CA, USA, 1995.

[Syntropy]. “Designing Object Systems”, Steve Cook and John Daniels, Prentice Hall, 1994. http://www.objectdesigners.co.uk/syntropy

[Z]: “Software Development with Z” *J. B. Wordsworth* [Addison Wesley]

Also http://www.comlab.ox.ac.uk/archive/z.html, The Z Notation

[Zaremski]: A M Zaremski & JM Wing, Specification Matching of Software Components. Proceedings of 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering, October 1995. Also http://www.cs.cmu.edu/afs/cs.cmu.edu/project/venari/www/sigsoft95.html