Appendix B UML Perspective

The Unified Modeling Language (UML) 1.0 was defined by a consortium of companies between 1996 and 1997 and was submitted for standardization by the OMG. The companies who cosubmitted the proposal were Rational, HP, TI, Microsoft, ICON Computing, Unisys, and MCI Systemhouse. UML 1.1 was standardized in September 1997.

Both Catalysis authors were involved in helping shape UML, which incorporated several Catalysis ideas. One author's company, ICON Computing, was a cosubmitter in defining and submitting the UML standard to the OMG.

This section briefly discusses UML and explains how we have used it in Catalysis, including some of the extensions or interpretation we have found useful or necessary, such as parameterized attributes, frameworks, and use cases. There are systematic approaches to using Catalysis with existing CASE tools (refer to www.catalysis.org).

Type versus Class

UML 0.8 did not distinguish type from class; 1.0 recognizes the distinction, thanks in part to Catalysis input. The standard UML box can have a <<class>> stereotype to indicate an implementation class, or can have a <<type>> for an implementation-free specification.

Combining Operation Specs

UML does not define multiple specifications for an operation. Catalysis permits two forms of combining multiple specifications: a conformant anding of the specs, for use with subtyping; and the join mechanism, in which one spec can impose additional preconditions to those defined in another view.

Frameworks

Catalysis depicts a framework application using the UML symbol for a pattern. The UML pattern notation has no defined semantics. Catalysis has a strong semantic foundation for frameworks; it is based on package imports and substitution. In addition to defining design patterns and domain-specific modeling patterns, frameworks can be used to define (and extend) the meaning of modeling constructs themselves. Type, associations, subtype—all these can be defined as frameworks, as can new constructs for workflow, event propagation, and property couplings.

Subjective Models of Containment

Catalysis gives a clear meaning to the containment of one type within another using state types. It is unclear in UML whether a given class diagram is defining intrinsic properties of the classes (what a theoretician might call axioms) or defining properties that happen to be true in the context of the containing type (theorems, based on constraints specific to the container).

const versus {frozen}

Catalysis uses an explicit const annotation to describe attributes that always refer to the same object. UML uses {frozen} and “composition” in sometimes unclear ways.

Stereotypes

UML provides the stereotype mechanism for extensibility. It has several weaknesses:

- There is no way to define the meaning of a stereotype and no mechanism for grouping stereotypes that can be meaningfully used together.
- A stereotype is attached to a single element; it cannot be used for a multi-way pattern.
- Only one stereotype can be attached to an element. UML permits a stereotype to be defined as a combination of several others. This arrangement causes problems with combinatorial explosion as well as lack of semantics.

Catalysis uses model frameworks for extensibility; stereotypes are only one special (and limited) syntax for applying a framework.

Attributes versus Operations

UML treats attributes on a <<type>> as abstract but does not permit attributes to take parameters; instead, you can use an operation that has been marked as read-only. Catalysis encourages a slightly different way of thinking about an object: Some things are abstractions of its state, and others are abstractions of its behaviors. Attributes, parameterized attributes, and invariants all help to define the model of state; operation specifications define the behaviors. There are no preconditions and postconditions on attributes, because they represent state; there can be conditional invariants on an attribute, and there can be syntactic shortcuts for expressing them, but they are not pre- and postconditions:

```
    inv Man::          married ==> wife <> null
```

In Catalysis an attribute does not imply an access method; you would have to explicitly add one (or use a framework as a shortcut).

Sets and Flat Sets

Catalysis makes a distinction between sets and flat sets. By default, traversing a "*" association yields a flat set; further traversals also yield flat sets rather than sets of sets. However, you can always explicitly annotate a different type if needed; so you can, when needed, deal with sets of sets; UML/OCL does not permit this.

Primitives and Values

UML provides a set of primitive types as "values." There are no primitives in Catalysis; every type can be defined in Catalysis itself and we do not force a distinction between object and value. Programming "call-by-value" is easily accommodated by using a "copy" framework: A call by value means the same as to make a copy and call by reference to the copy.

Attribute, Role, and State

UML has three distinct constructs that remain largely unrelated to one another: an attribute, a role in an association, and a state in a state chart. For example, a pre- and postcondition pair cannot refer to the name of a state. In Catalysis these three constructs are closely related. An association defines a pair of inverse attributes that are drawn differently; a state defines a Boolean attribute, and the structure of states in a state chart defines invariants on those attributes. Transitions on a state diagram are thus no more than a graphical depiction of action specifications. This approach provides a simpler core set of constructs with different presentations. Actions on UML state transitions are imperative and appear unrelated to pre- and postspecification of operations.

Packages

A UML package is used primarily as a namespace mechanism; elements in one package can refer to elements in another by using package-qualified names. There are no strong semantics associated with import; instead, a general "dependency" relationship can be defined between packages. In Catalysis, packages and imports are fundamental to the structuring of any model, separation of business, specification, and implementation. We do not support arbitrary references across packages.

Packages can be nested in UML and Catalysis. In UML, the containing package automatically imports (transitively) its constituent packages. In Catalysis we much prefer to use the proven scoping rules of programming languages (or predicate calculus or lambda calculus). For example, a Java inner class can access members of its container and "sees" the changes in their values; a Smalltalk block can access variables in its scope and its containing class. A Catalysis nested package automatically imports its container package. If you import a container package and rename one of its types, that renaming effectively applies to all its contained packages that became visible to you only as a result of the import. This provides the intuitive behavior you would expect with frameworks: A container package can have a type parameter, T, and all its nested packages can refer to that type T.

In addition, Catalysis packages let you “say more” about a type, attribute, operation, and so on imported from another package. You can separate viewpoints and then rejoin them in an importing package downstream using well-defined join rules.

Type System

The Catalysis type system treats types as sets and correspondingly provides type expressions such as HotelGuest * Passenger:, which is the intersection of the two sets with both sets of properties.

Parameterized Types

UML has a specific notation for parameterized types: a type box with a syntactical place for its parameters. The semantics of such types are not defined within UML. We support this notation in Catalysis; its semantics are entirely defined in terms of frameworks.

Activity Diagrams

UML has an activity diagram; it is given semantics roughly like that of a state diagram but is not integrated to the other models. This is unfortunate, because the activity diagram is offered as *the* UML mechanism for modeling business processes when enhanced with diagramming constructs such as *swimlanes*.

There is a way in Catalysis to integrate the traditional business-process notations, including organization charting and process and activity flow and decomposition; we have not published it as part of this book.

Use Cases

The use case has become widely used in object methods; unfortunately, it is at least as widely misunderstood and misused. Catalysis provides the framework of joint actions and refinement as a way to clearly separate the specification of any action from the lower-level protocol that might realize that action—whether or not it involves a user. Catalysis lets you factor out common parts of use cases as effects, define exception paths via refinement, and describe rules that affect multiple use cases as dynamic invariants.

Components

UML offers one notation to define the interface of a component (the “lollipop” notation). Catalysis recognizes that component technology is about the connecting together of encapsulated units. The types of these connections will vary; a given component architecture will define its own types. You can define a component architecture using frameworks and then use those frameworks to compose components as diverse as workflow, event-property-method connections, and traditional objects.

Components in Catalysis need not be built using object technology. The concept of a type model and operation spec can be applied equally to a Java, C++, COBOL, or assembler implementations—and therefore to legacy systems—thanks to refinement.

Collaboration

In Catalysis, a collaboration is defined as a set of related actions between objects. UML provides one definition for collaboration and then uses the term *collaboration diagram* to depict a sequence of messages that result from a particular triggering request to an object. We considered the term *use case diagram* but have not seen that used consistently for anything except the actor-level context diagram of a system.

Joint Actions

A Catalysis joint action corresponds to a use case; we prefer to separate the specification of the net effect of a joint action—its postcondition—from the interaction sequence (one of many possible) that realizes it. A joint action is an abstraction of that detailed interaction, just as an operation invocation is an abstraction of a language-specific call-return convention. We depict occurrences of joint actions on an interaction diagram; we have not found any way to show this in UML.

We are uncomfortable with the treatment of use cases as objects and have found much confusion among practitioners about what this means. Just because there are commonality and variation across use cases does not mean that they should be modeled as objects; to do so confuses the separation of actions (interactions that cause changes of state) and attributes (the states that affect, and are affected by, actions). In Catalysis, what is modeled as an action at one level of abstraction (for example, buy_product) can easily be reified into a model object in a refinement (for example, place_order and deliver and pay actions revolving around an order model object).

We have found many use case practitioners confused about the definition of use case: What is its level of granularity? Does a use case always correspond to a unit of meaningful business interaction for an actor? What if two use cases share common parts, but that part does not form a meaningful unit of business interaction: can it still be modeled as a use case and "used" by the other two? In Catalysis we recommend a distinction between the action, or use case, and the *effects* of that action: it is easy to factor out effects that are common across two actions.

Exceptions

Catalysis provides explicit support for specifying exceptions, for composing specifications of an action with such exceptions, and for refining abstract actions with exceptions down to the level of operation invocation sequences and language-specific exception mechanisms.

Capturing Rules

Catalysis static invariants (this constraint between these attributes is always true) and dynamic invariants (any action that causes such-and-such must also cause so-and-so) let us succinctly capture most business rules.

Events

The UML definition, even of an “event” is very limiting:

UML: “. . . constraint on state.”

Events are fundamentally about changes of state, not about constraints on state.

Enumeration Types

UML/OCL has an enumeration type and a separate concept of “class attributes.” In Catalysis we use shared attributes on a type, with an invariant, to provide enumeration types.

There are other important points to make regarding both UML and the Rational Unified Process. For details, see www.catalysis.org.