Chapter 8 Composing Models and Specifications

When you're building models and specifications, it is important to be able to compose them with a clearly defined and intuitive meaning. This clarity makes reuse and learning easier, because you can understand the whole by understanding the parts and then recombine them in a predictable way. All descriptions in Catalysis can be composed—from attributes and actions on a type to entire packages.

A package is always built upon others that it imports. They provide the definitions of more basic concepts that it uses, all the way down to utter primitives such as numbers and Booleans.

A package augments and extends the material it imports. It can import several packages so that their different definitions are combined. In many cases, the different sources of material will deal with some of the same things. There must therefore be clear rules whereby the definitions that are found within a package are *joined* with others.

This chapter deals primarily with how to use packages in the building of other packages and how to interpret the resulting compositions. It also discusses some nuances of specifying and composing specifications of operations in the presence of exceptions.

8.1 *Sticking Pieces Together*

Every method of development must be good at building its artifacts—models, designs, plans, and so on—from smaller pieces. We can get our heads around only a small chunk at a time and can build big things only by sticking small ones together. Moreover, parts are more likely to be reusable if they can be put together in various ways with predictable results.

Note: Yes! Entire chapter is an advanced topic.

Much of this book is about building from parts. We'll discuss building models from frameworks and building software from components. The software has its own intricate plugging mechanisms; this chapter is about the much simpler matter of combining models, as used in type specifications and collaborations. Here, we are putting together specifications and high-level designs, so this is a design activity rather than one of integrating code.

There are three specific situations that call for composing models.

- The documentation chapter (Chapter 5) says that we can present a large model as a series of smaller diagrams. That's great for a guided tour through the model, but an implementor must see everything relevant to each action or type. So we need exact rules by which the diagrams recombine to make the big picture.
- The packages chapter (Chapter 7) talks about one package importing others. We need rules governing how to combine the definitions from the different sources. (The frameworks chapter, Chapter 9 takes this idea even further and combines generic models.)
- Our components (Chapter 10) can have multiple interfaces; that is, they must satisfy the expectations of several different clients, who might or might not know about one another. Each interface is defined by a type, so we need a way of working out what it means to satisfy two or more types at the same time.

This chapter deals with the basic combining mechanisms that are used in different ways in all these situations.

8.2 *Joining and Subtyping*

In Catalysis, we use two different ways of composing types: a type join and a type intersection.

© ***Type join*** A way of composing types that combines two views of the same type; each view can impose its own restrictions on what the designer of the type must achieve (conjoining postconditions) or what a client must ensure (conjoining preconditions). Joining happens when two views of the same type are presented in different places; they might be in the same package or imported from different ones. Joining is about building the text and drawings of a specification from various partial descriptions.

© ***Type intersection*** A way of composing types that combines two specifications, each of which must be fully observed (without restricting the other) by an object that belongs to the resulting type. The designer must guarantee each postcondition whenever its precondition is met regardless of the other's pre/post. Type intersection, or subtyping, happens when a component or object must satisfy different clients. Each specification must be satisfied independently of the other. A type specification defines a set of instances: the objects that satisfy that spec. Subtyping is about forming the intersection of two sets: those objects that happen to satisfy both specifications.

The behavior of anything from a simple object to a large, complex system can be specified with a type specification, which has actions specified in terms of a model. The rules for joining and subtyping can be described as operations that you perform on the specifi-

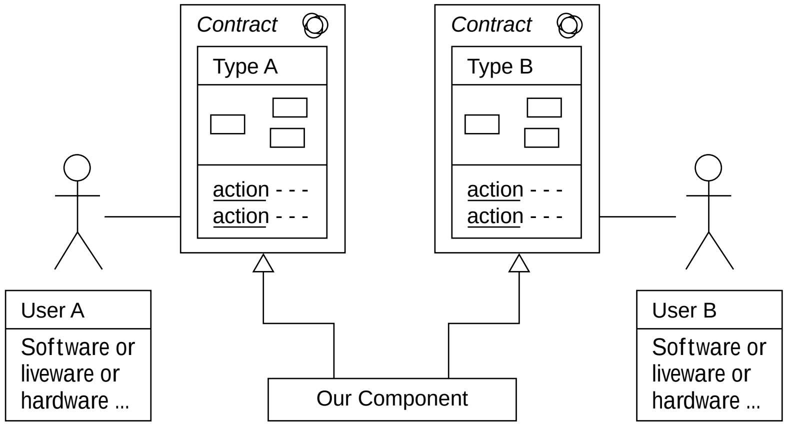


Figure 8.1 Component must satisfy multiple views.

cations. After a quick recap of type intersection, the rest of this chapter will focus on the joining operation.

8.2.1 Type Intersection: Combining Views

Type intersection means conforming to the expectations of more than one client, each of whom has a self-contained type specification you must conform to (see Figure 8.1). In any system that has more than one class of user, each of them has his or her (or its) own view. In software, the pluggable components that we want to build need the same capability. From user A's point of view, no matter what else Our Component does, it must always conform to the expectations set by Type specification A; the same thing is true of user B. In the most basic form of OO design, we use our type specifications as the basis for the design. But here there are two sets of models and actions. How do we go about designing it? Following are the basic steps for combining two types.

1. *Combine the two lists of attributes*. If any attributes from the two types have the same name, determine whether it's accidental or because the name is inherited from a common supertype. If it's the former, rename one of them; it won't make any difference to the meaning of the type. But if the duplicate names are inherited, they really do mean the same thing.
2. *Merge the two lists of operations*. If each action has a unique name, that's easy. For any operation that has specifications from both types, follow these steps.
   - and the precondition from each type with the invariant from that type.
   - and the postcondition from each type with the invariant from that type.
   - Write the new operation specification using the anded expressions as

```
pre     pre1 or pre2
post    (pre1==>post1) and (pre2==>post2)
```

Why must the invariants be absorbed into the action specs? They must be absorbed because it's the pre- and postconditions that are the real behavioral spec: An invariant is only a way of factoring out common assumptions made by all those within its own context. Outside that context, the same common assumptions might not apply (see Section 3.5.5), so we must make them specific in any pre- and postcondition we want to move out of the context. After all the actions have been combined, you can probably factor out a common invariant that occurs in all the combined operation specs.

If we are talking about two 300-page type specifications, this process may take a little longer. This task is covered properly in the components chapter (Chapter 10), but the essence is the same on a larger scale.

- Build a new model sufficient to include the state information from both types; verify this (and assist the testing team) by writing two sets of abstraction functions (see Section 6.4) that will retrieve any piece of information from your component back into the language of each type.
- Then build your component using façades, each of which is dedicated to dealing with only one of these clients (see Section 6.7) and supplies the actions it expects, translated from your component's internal model.

8.3 *Combining Packages and Their Definitions*

So far, we've rather glibly talked about the way definitions are augmented and combined when packages are imported. This section looks in more detail at how the materials from imported packages are combined with each other and with the additional material already in the importer.

8.3.1 Definitions and Joins

What exactly does a type box in a diagram mean? What does it mean if boxes claiming to represent the same type appear in different diagrams in a package? Or are imported from different packages into one? Within one package, boxes headed with the same type name may appear in different parts of the same diagram, in different diagrams in the package, and in textual statements in the dictionary or the documentary narrative. Some of the diagrams in the (unfolded) package may have been imported from other packages, but that makes no difference. Whatever the source of the multiple appearances, it is always possible to join all of them into a single type definition.

The rules are the same for all kinds of multiple appearances, whether they are multiple diagrams in one package or definitions from multiple packages. The operation of combining them is called *joining*. Frequently, the individual splinters, taken on their own, don't denote any type; it's only when you put them together that they define a set of objects characterized by a particular behavior.

Each kind of definition that you can find in a package has its own joining rules. This section describes the rules for each kind.

8.3.2 Joining Packages

You form the join of two packages by forming a bag of all the statements and definitions from both packages and then joining those definitions with the same names (after any renaming, as discussed in Section 7.7.1, Name Mapping on Import). This join is applied to any nested packages along with everything else.

You obtain a package's full unfolded meaning by joining its imports to its own contents. There are specific rules for joining different kinds of definitions.

Type specifications are joined by joining their static models and their action specs.

8.3.3 Joining Static Models

To join multiple appearances of a type into a single type definition, follow these rules.

- and all invariants. The type has an effective invariant that is the conjunction of all the separate ones.
- Put into a set all the attribute names from the appearances; the completed definition should have the same set of attribute names. Include association names and parameterized attributes in the same way.
- For each attribute (or association) name in an appearance, consider any type constraint to be an invariant. So count:Integer means, "For any object x of this type, x.count always belongs to the type Integer." The completed type should contain an invariant that ands these together. So

```
count : Integer   and   count : Number     ==>     count : Integer
                        -- all Integers are Numbers anyway
thing : Boolean   and   thing : Elephant
                        -- contradiction – can't join these definitions,
                           unsatisfiable spec
connection : Trasmitter   and   connection : Receiver
==> connection:Transceiver   -- which is a subtype of both of these types
```

- If the attribute has parameters (similarly, if the association has qualifiers), you can write the attributes as overloaded functions, provided that the parameter types don't overlap. So price(CandyBar):¢ and price(FreeFlight):FlyerMiles can remain as two attributes, because there is no CandyBar that is also a FreeFlight (yet). The general rule is that you first convert the attribute types into invariants of the form

```
(param1 : Type1 and param2 : Type2) ==> attribute : ResultType
```

  anding such expressions together gives a result that says, "If you start with parameters such as these, you get this kind of result; if you start with parameters such as those, you get that kind." If an argument ever belongs to both parameter types, then the result should belong to both result types.[1]

8.3.4 Joining Action Specifications

Action specifications are joined according to a covariant rule that permits any appearance of a type to reinforce preconditions and invariants. The rules apply to operations or actions localized to particular types and apply to joint actions.

Treat actions having different signatures (names and parameter lists) separately. For each action signature, take the individual pre- and postconditions and do the following:

- and the preconditions
- and the postconditions
- and the rely conditions
- and the guarantee conditions

anding preconditions means that in different packages or diagrams (or in different parts of your narrative) you can use preconditions to deal with different restrictions, confident that these restrictions will apply regardless of what is analyzed in other packages. Under Fault Management, we can say that a call can be made only if the Line is not under maintenance; under Billing, we can say that a call can be made only if the Account associated with the Line is not in default. Each package has no comprehension of the other's constraints; yet the net result is that a call cannot be made unless the line is free from maintenance and not in default.

anding postconditions means that, in different packages, you can use postconditions to deal with different consequences of an action: Under Billing, you can say that a charge is added to the associated Account; under Fault Management, we can say that the count of successful calls is incremented.

anding rely conditions means that in different packages, you can define different invariants the designer should be able to rely on while the action is in progress; anding guarantee conditions allows you do separate invariants your action will preserve.

Thus, the two separate specs could be

```
action Agent::sell_life_insurance (c: Customer)
    pre:     c.isAcceptableRisk      -- provided the risk factor is OK
    post:    c.isInsured             -- issue insurance when I "sell life insurance"

action Agent::sell_life_insurance (c: Customer)
    pre:  c.home : self.territory    -- if the customer is a part of my territory
    post: territory statistics updated   -- update statistics when I "sell life insurance"
```

The combined spec as the result of joining the two would be

1. All this conforms to the usual contravariant rules.

```
action Agent::sell_life_insurance (c: Customer)
pre:    c.isAcceptableRisk and c.home : self.territory -- combine restrictions
post:   c.isInsured and territory statistics updated -- combine outcomes
```

The reason for using these join rules for actions is that each action specification could have been written without knowledge of the other specification or of its attributes. When I write my preconditions I do not know what other preconditions you may want to impose on that action; we both should be confident that, when our separate specifications are joined together, each can rely on its restrictions still being in force. The same principle applied to postconditions.

Sometimes, however, you want to write a specification that makes guarantees for certain cases when these cases may overlap with others. In this case you can use an alternative style for writing the spec: do not use an explicit precondition but instead describe the case within the postcondition itself. Here we use the same composition rules:

```
action Stack::push (in x, out error: Boolean)
post: self.notFull@pre ==> self.top = x and error = false
        -- provided I was not full beforehand, x will now be my top element
```

This spec tells us what happens in the successful case and might be, on its own, all we need. But perhaps in another part of your spec you want to write down what would happen in the other case:

```
action Stack::push (in x, out error: Boolean)
post: self.full@pre ==> error = true
        -- provided I was full beforehand, you will get an error flag.
        -- I'm not telling you what might happen to my contents!
```

The joined spec would make one guarantee for one case and another guarantee for the second case (the two cases happen to be disjoint in this example).

```
action Stack::push (in x, out error: Boolean)
post:        (self.notFull@pre ==> self.top = x and error = false)
    and      (self.full@pre ==> error = true)
```

It is possible to compute a resultant precondition from this specification:

```
action Stack::push (in x, out error: Boolean)
pre:    self.notFull or self.full
post:        self.notFull@pre ==> self.top = x and error = false
    and      self.full@pre ==> error = true
```

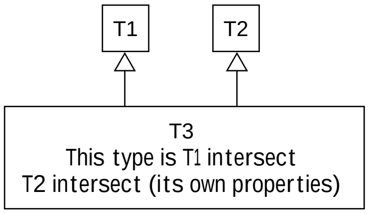
Hence, these two different styles of writing specs can be used to accomplish these two different goals of composing separate specifications. More details on dealing with exception conditions in specifications appear in Section 8.4, Action Exceptions and Composing Specs.

8.3.5 Joining Type Specifications Is Not Subtyping

A type specification denotes a set of objects: All objects that meet that specification are members of that type. Some ways of combining type specifications correspond directly to operating on the corresponding sets of objects; join does not.

When you join type specifications, you are combining the descriptions themselves and not directly the types (sets of objects) they specify. In the usual case, two type specifications are joined when a package, P1, imports two other packages—P2 and P3—each of which provides separate specifications (T_{s1} and T_{s2}) for the *same* type, T. Within package P1, the resulting specification of type T is the specification that results from a join:

T_{s1} join T_{s2}



In contrast, when you define a subtype, you are defining a subset of objects; when you combine multiple supertypes, you are intersecting the corresponding sets. The rule for intersecting action specs is quite different from the rules for join. You can write a different expression—T1 * T2—which represents the type of objects that conform both to T1 and also to T2—that is, the intersection of the two sets. Type intersection or subtyping is a "no surprises" combination. Anything you're guaranteed by one spec can't possibly be taken away by the other. It is what happens when you have multiple supertypes, each of which provides specs for the same action; or when you combine a supertype action spec with a corresponding spec in the subtype.

Suppose we had the following explicitly declared specs:

```
T1::m pre: A post: X
T2::m pre: B post: Y
T3::m pre: C post: Z
```

The resulting equivalent spec, after combining with the supertype specs, on the type T3 is obtained by anding all three pre/post *pairs*:[2]

```
T3::m ( pre: A post: X) and ( pre: B post: Y) and ( pre: C post: Z)
```

That is equivalent to

```
T3::m pre: (A or B or C) post: (A @pre ==> X and B @pre ==> Y and C @pre ==> Z)
```

An implementation of the resulting operation spec is guaranteed to meet the expectation of anyone who expected either T1 or T2. An invocation of m is valid whenever A is true (because that would make (A or B or C) true) and is guaranteed to result in X (and perhaps *also* Y, Z depending on whether B or C was also true).

When you join two type definitions, you are not usually intersecting the types, depending on how the action specs were written. For example, if I ask you for an object that conforms to Billing's idea of a Line, I would expect to be able to make calls whenever the Account is in order. If you give me something that conforms to Billing::Line join Fault_Management::Line, then I will find to my dismay that sometimes my Account is OK but I still can't make calls.

In fact, many partial specifications that you find in models don't constitute a complete type specification at all—they have attributes but no actions. Strictly speaking, any object

2. In a join, you separately and the precondition and then the postcondition.

would satisfy such a type spec, because it states no behavioral requirements. Types that are only attributes (and associations) are meaningful only as part of the models of larger types.

8.3.6 Joining Action Implementations

You can implement an action's specification by designing a refinement into a smaller set of actions—ultimately, in software, messages. Program code is the most detailed kind of action implementation. Implementations cannot be joined in the same way that specifications can be joined (see Figure 8.2). First, it isn't clear what anding two programs together would mean. The machine must follow one list of instructions or the other; which one should it execute? Both? In what order? Therefore, your support tool should complain if you try to provide code for two operations with the same name in the same class or for two refinements of the same action into different sets of smaller steps.

A second complication is that any implementation of the Call action provided by, say, the Billing package isn't likely to satisfy the requirements specified by Fault Management, because neither world understands the concepts of the other. So we cannot always accept an imported implementation even if there is no competing implementation from the other packages.

The only circumstance under which an implementation can be imported is when there is no difference between the pre/post specification in the imported package and the corresponding specification in the unfolded importer—that is, when there is no extra material about this action from the other imports, and no extra material is specified here. In that case, we know that the designer was working to the same spec.

Does this mean that we cannot bring together code written in different packages? Of course we can, but the code must be routines that can be referred to separately; and as designer of the importing package, we must design the implementation that invokes each of them in the right way. Some languages allow you to invoke super.method(); others permit Super1::method() and Super2::method().

Each of the packages for the telecoms network is a *view* of the whole system; it is constructed from the point of view of one department or business function. Knowing this, the package designers should not presume to provide their own implementations of overall actions. Instead, they should provide auxiliary routines that help implement their concerns. So Faults could provide successfulCallLog and Finance could provide callCharge. The designers of Telecoms Network Implementation can then choose to invoke these routines where appropriate in their own implementation of Call.

8.3.7 Joining Classes

A class defines an implementation or partial implementation of an object, with program code for localized actions and variables for storage of its state. A class can also be documented with invariants over its variables and pre/post specifications for each operation signature. Some programming languages support these features—notably Eiffel, which provides a testing facility that uses them.

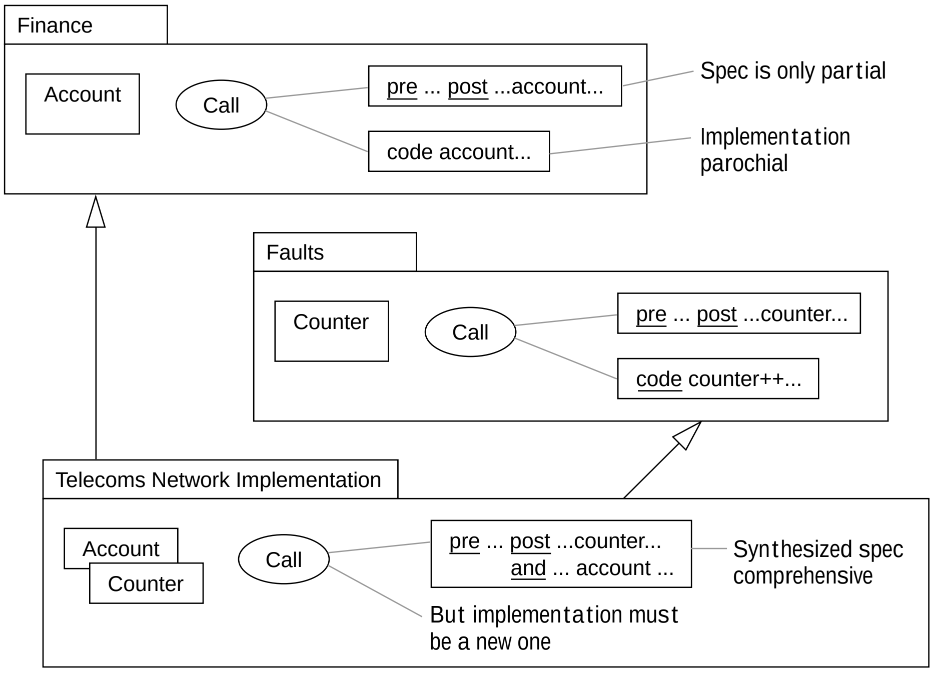

Finance
Account
Call
pre ... post ...account...
code account...
Spec is only partial
Implementation parochial
Faults
Counter
Call
pre ... post ...counter...
code counter++...
Telecoms Network Implementation
Account
Counter
Call
pre ... post ...counter... and ... account ...
Synthesized spec comprehensive
But implementation must be a new one

Figure 8.2 Importing and joining specs versus code.

The idea of joining class definitions isn't something you find in a programming language. By the time you get to compilation, it's assumed you've chosen your program code and there is no need to automatically compose it with any other code.

But some programming environments, such as Envy, allow a class to be synthesized from partial definitions imported from different packages ("applications" in Envy). There are restrictions that help prevent ad hoc modification of the behavior of the instances. Among other things, this arrangement permits a useful form of structuring development work. Because an object typically plays multiple roles and each role is meaningful in the context of interactions with objects playing other roles, the class is not the best unit of development work to assign to a person or team; instead, the collaboration between roles should be an implementation unit.

Other interesting work has been done in the area of *subject-oriented programming*, which strives to compose implementation classes that define different views, or roles, of some objects.

The rules in Catalysis are as follows.

- The joined class has all the variables in the partial definitions. Types must be the same. (If they were widened, the preconditions of some methods might fail, finding values in the variables they couldn't cope with; if they were narrowed, some methods might find they could not store the values they needed to.)

- The joined class has all the methods from the partial definitions. Methods are joined according to the rules for actions: Only one method for each signature is allowed (within this class) and not even that if there is a pre/post spec attached to that signature in one of the other joinees. Pre/post specs can be inherited from superclasses.
- Invariants over class variables, and pre- and postconditions attached to message signatures, are treated as for joining types: They are anded.

8.3.8 Joining Narrative

How shall we combine the narrative documentation of a package with those of the packages being imported? Perhaps the best that can be done automatically is to stick them end to end.

But decent support tools provide for a hypertext structure. Importing means that the points of reference in the original text can be referred to from the importer's text; and, in the context of renaming (see Section 7.7.1), the resultant imported text can actually be customized based on the importer's text and names.

8.4 *Action Exceptions and Composing Specs*

Exception handling adds complexity to any application. Even if the normal behavior of a component would be easily understood, the presence of exceptions often complicates things dramatically. We want to be able to separate exception specification to simplify the normal behavior specs, but we also want to address the specific characteristics of exceptions, and compose specs containing exceptions.

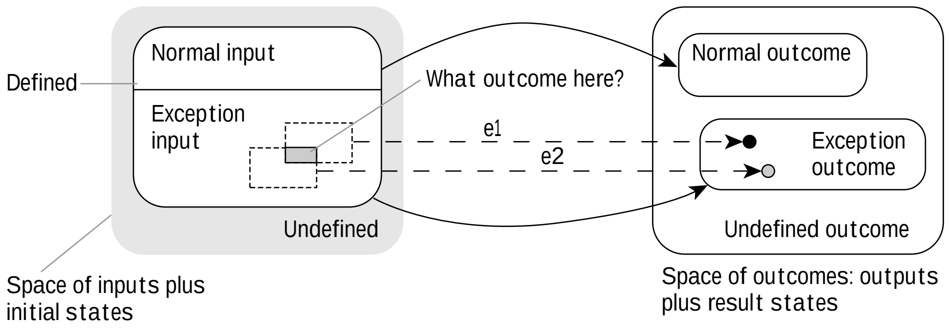


Figure 8.3 The spaces of normal and exception behavior.

8.4.1 Required Exceptions versus Undefined Behavior

The outcome of invoking an operation with some inputs and initial state will be either *defined*—the operation is required to complete and the outcome must satisfy some specification—or *undefined*—the specification does not constrain the outcome for those cases.

Any defined behavior could, in turn, be considered as (1) *normal*—the operation performed the required task—or (2) an *exception*—the operation did not perform the required task because of an anomaly and signaled the failure as required. Figure 8.3 illustrates this behavior.

It is important to distinguish the exception case, in which the operation has met its specification with an exceptional outcome, from the undefined case, in which the operation has no specified behavior. Figure 8.3 shows that normal and exception outcomes are disjoint, because there should be unambiguous checks to distinguish required success from required failures; the caller should not be guessing: *"Hmm... I wonder whether that last call succeeded."*

Figure 8.3 also shows that the same input can give rise to different exception outputs (`e1`, `e2`); some inputs may cause more than one exception condition to be true.

8.4.2 Design by Contract versus Defensive Programming

Over the years, there has evolved an approach to programming called *defensive* programming. In essence, when you implement any operation you do the following.

- Consider the normal invocation of your code; implement it.
- Consider the countless abnormal invocations of your code; implement checks for those conditions and take defensive actions such as returning a null or raising an exception.

This approach can be quite damaging. Each implementor provides identified defensive checks *in the code*, but none of the interfaces documents what is guaranteed to be checked and by whom or what outcome is guaranteed in the event of those errors. Responsibilities become blurred, and the code becomes littered with disorganized, redundant, and inadequate checking and handling of exception cases.

Instead, you should be clear about the separation of responsibilities in the design itself. The specification of each operation should clearly state what assumptions the implementor makes about the invocations—the caller must ensure that those are met—and what corresponding guarantees the implementor will provide. This includes a specification of which failure conditions or paths the implementor guarantees to check and the corresponding outcomes. Then implement to that contract; allow for the defensive programming mode when you're debugging the code and when running tests (see Section 6.1.3, Operation Abstraction).

By all means, employ "defensive specification" at appropriate interfaces in your system; but make sure that the checking and exception handling are specified and documented as part of the interfaces and not just in the code.

8.4.3 Specifying Exceptions

We want to separate normal and exception conditions, both within one spec and across multiple specs. However, we still want our descriptions to compose with predictable and intuitive results.

In Catalysis, the approach of specifying using pre- and postconditions simplifies matters, because you can have multiple specifications for an action that compose following clear rules. However, exceptions pose unique requirements.

We introduce two special names: normal and exception. These names can be used in two ways.

1. As Boolean variable names that can be bound before the pre/post specification section; define normal and exception in terms of the success and failure indicators that the operation will use. They are treated as special names, as opposed to names introduced locally within a let..., because their binding must be shared across all specifications of that action.

```
action Shop::order (c: Card, p: Product, a: Address, out success: Integer )
    normal = ( success=0 ) ... -- success indication to the caller
    exception = ( success < 0) ... -- failure indication to the caller
post: ...
```

2. As Boolean variables that can be used within a postcondition. For example:

```
action Shop::order (c: Card, p: Product, a: Address, out success: Integer )
post:     c.OK ==> normal -- success indication if card is OK
          if (....) then (exception and ....) -- failure indication must be raised if ....
          if ( exception ) then ( ..... ) -- any failure must guarantee ....
```

We can now require the operation never to have an undefined outcome:

```
post:     ( normal or exception ) = true
    -- must indicate success or failure; returning +2 would be an implementation bug
```

Or we require it never to raise any exception outside a particular set:

```
post: exception ==> ( success = -1 or success = -2 )
```

We can now write the successful outcome, assuming that the success indicator will be defined somewhere. Effects are guaranteed with the success indicator:

```
action Shop::order (c: Card, p: Product, a: Address, out success: Integer)
post:     normal ==> (        -- if success is returned, then caller is assured the following
              Order*new->notEmpty and -- new order
              c.charged (...) and      -- customer card charged
              ( p.noInventory ==> RestockOrder*new [...] notEmpty)
                                       -- restocking order if out of stock
```

Or we can write conditions under which success must be indicated:

```
action Shop::order (c: Card, p: Product, a: Address, out success: Integer)
post:     c.OK ==> normal -- if the card is OK, definite success indicator
```

We could write the exception outcomes in the same manner (but see the later discussion of why this would be inflexible with multiple possible exceptions).

```
action Shop::order (c: Card, p: Product, a: Address, out success: Integer)
post:     not c.OK ==> ( success = -1 ) -- specific indicator for bad card

action Shop::order (c: Card, p: Product, a: Address, out success: Integer)
```

```
post:    not a.OK ==> ( success = -2 ) -- specific indicator for bad address

action Shop::order (c: Card, p: Product, a: Address, out success: Integer)
post:    exception ==> ( Order*new–>isEmpty )
         -- if failure is signaled, guaranteed that no new order was created
```

Typically, however, you want to deal with multiple possible exception outcomes in a more flexible manner. If you place an on-line order, given the preceding spec, what should happen if the credit card number and address are both invalid? Which exception should be raised? It is best to leave to the implementor the choice of *which* exception to signal as long as failure indication is guaranteed. This technique helps with composition of specifications, each with its own exception conditions, as is the case of failures in distributed systems. Hence:

```
action Shop::order (c: Card, p: Product, a: Address, out success: Integer)
post:    -- bad card means some failure indication
         not c.OK ==> exception
         -- a failure indication, with code -1, will happen only if the card was bad
         (exception and success = -1) ==>  not c.OK
```

This is a common form of specification for exceptions, so we introduce a convenient query isException on the predefined type Boolean and rewrite it as

```
post:    (not c.OK) . isException (exception, success = -1)
```

This is exactly equivalent to the longer form. Our definition of isException is as follows:

```
-- a given trigger condition is Exception means...
Boolean::isException ( generalFailure: Boolean, specificIndication: Boolean ) =
         -- if the trigger condition was true, then some failure has been signaled, and
    (    (self = true) ==> generalFailure ) and
         -- the specific Indication will not be raised unless the trigger was true
         ( generalFailure & specificIndication ==> (self = true ) ) )
```

One final point: Suppose you write a specification that says simply

Success indicator = a; Failure indicator = b;

If a, then x is guaranteed to have happened;

If b, then y is guaranteed to have happened.

You would, strictly speaking, have to admit an implementation that simply failed every time, as long as it met the failure indication. Either you should be more strict about the kinds of exceptions that can be raised and when, or you should assume a reasonable convention in which the implementor is obliged to try to meet the success goals and should raise an exception only if that turns out to be impossible.

8.4.3.1 Other Exception Indication Mechanisms

Different languages have different mechanisms to indicate exceptions: return values, exception objects thrown, signals raised, and so on. For specification purposes, you can work with any of these, including some language-neutral mechanisms (such as return values; remember that the signature of an abstract action specification is itself always subject to refinements (see Chapter 6, Abstraction, Refinement, and Testing).

```
action T::m (....., out success: Boolean)
post:     not success ==>  (...guarantees about every indicated failure...)

action T::n (....) throws (Object) -- Java-like exception spec
     exception = thrown (Object) -- no using throw except to indicate failure

action T::n (.....) -- WrongState exception spec
post:     (self.wrongState) . isException ( thrown (Object), thrown (WrongState) )
```

Or to make a guarantee on any exception thrown

```
action T::n (....)
post:     thrown ( Object ) ==> (...e.g., all state cleaned up ...)
```

The meaning of the language-specific mechanisms, such as throw, is provided by working in a context where the appropriate packages have been imported (see Chapter 9, Model Frameworks and Template Packages).

8.4.3.2 Exceptions with General Actions

This approach extends to general actions. An exception in an abstract action can be traced across action refinements. You can specify traces or sequences of detailed actions as raising an exception on the abstract action (rather than having to invent a new abstract action

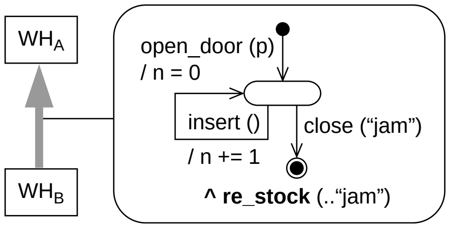


Figure 8.4 Mapping to an abstract exception action.

for it or having to ignore it at the abstract level). The exception can be traced through action refinements down to the level of exceptions in program code.

Of course, when specifying an abstract action you should describe only those exceptions that have meaning at that level of abstraction, not every disk or networking failure!

Let us revisit the example in Section 4.2.2, Preview: Documenting a Refinement, on restocking of a vending machine (see Figure 8.4). Suppose that the warehouse inlet door for a product can jam when closed. If this outcome is an interesting exception at the abstract level, it could have been specified as such on the joint action. In the refinement, the appropriate sequence can be mapped to this abstract exception action.

This provides a precise basis for exceptions in traditional use case approaches.

8.4.3.3 Exceptions and Use Case Templates

Just as we introduced a narrative-style template for defining use cases, it is also useful to incorporate exceptions into use case narratives.

use case	sale
participants	retailer, wholesaler
parameters	set of items
pre	the items must be in stock, retailer must be registered,
	retailer must have cash to pay
post (normal)	retailer has received items and paid cash
	wholesaler has received cash and given items
normal indicator	confirmation to retailer
exception indicator	no sale confirmation to retailer from wholesaler
on exception	neither cash nor items transferred

Similarly, as part of the use case documentation it is useful to document those sequences that might give rise to an exception. The mapping from the formal refinement description (such as a state chart) to this narrative is straightforward:

use case	telephone sale by distributor
refines	use case **sale**
refinement	1. retailer calls wholesaler and is connected to rep
	2. rep gets distributor membership information from retailer
	3. rep collects order information from retailer, totaling the cost
	4. rep confirms items, total, and shipping date with wholesaler
	5. both parties hang up
	6. shipment arrives at retailer
	7. wholesaler invoices retailer
	8. retailer pays invoice
abstract result	**sale** was effectively conducted
	with amount of the order total, and items as ordered
exception	retailer canceled order before it was shipped (step 6, use case **sale**)
exception outcome	confirmed cancellation
	-- implicit: non sales confirmation; no cash or items transfer

8.5 *Summary*

All modeling elements should be composable so that specifications and designs can be factored into smaller parts and recombined in predictable and intuitive ways.

Packages usually import other packages, those on which the importers' definitions are based. We have looked at the rules whereby imported definitions are combined with new material and with material from other imports.

Each package has a notional unfolded form, in which all the definitions from the imports and their imports are visible. New facts and definitions in a package can constrain its own declared names and those that are imported.

You need to take special care when you specify exceptions so that they can be composed and so that abstract actions with exceptions can still be refined. We outlined how to specify exceptions to meet both these needs, and we linked them to exception paths in use cases.