Scrall Specification

Author/Version: Leon Starr / Version 1.0.0

What is Scrall?

Scrall is an open source action language designed to leverage the distinct semantics of Executable UML. These semantics are informally described in the book Executable UML: A Foundation for Model Driven Architecture by Stephen J. Mellor and Marc Balcer. This document defines the rationale, structure and syntax of Scrall.

Resources and contact info

There is a bibliography section at the end of this document that lists a number of resources that you may find helpful when reading through this document.

The Elevator 3 Case Study will be published soon after this document and it includes many example uses of Scrall. In the meantime, the examples included in this spec will have to suffice.

You can also read our book Models to Code (modelstocode.com). Note that the Scrall used in that book was in its incipient phase and much of the syntax has evolved.

There are a few articles on xUML available at Model Integration's modeling learn page (www.modelint.-com/mbse):

You can also feel free to contact Leon Starr directly at: leon_starr@modelint.com

You may post public comments on twitter @Leon_Starr or the Executable UML linked in group or, well, anywhere you like!

Open source license

Introduction to Scrall and xUML

Scrall is an action language designed to fully leverage the powerful semantics of Executable UML (xUML). The semantics underlying most usages of UML have been loosely or tightly shaped by object oriented design and programming principles. xUML is distinct in that it is built primarily on mathematical foundations. These foundations are the relational model of data (sets, functions and predicate logic) for the class model, distributed networking theory for the state models and data/control flow semantics for activities.

Both requirements analysis and code generation goals benefit from a modeling language that does not seek to impose its own design assumptions. Mathematics provides a neutral ground upon which to specify such a language.

xUML was designed from the start to support real-time and embedded applications starved for computing resources where an object oriented implementation was not necessarily a preferred or workable solution. So great care was taken to remove any object oriented bias from the design and implementation while preserving those aspects of an object oriented perspective that were beneficial in analysis and system specification.

The xUML semantics do not presume an object oriented programming language on the target platform. Design concepts of namespaces, pointers, inheritance, constructors/destructors, etc. are absent from the xUML language. And yet there are no built-in impediments to using those same design concepts in any given implementation. Further, there is no bias in xUML toward any particular degree of implementation concurrency. Formulated properly, xUML models should specify the maximum possible concurrency (in other words they don't introduce any unnecessary sequencing) in the system specification. Any subsequent design can introduce as much sequencing as is desired (running everything in a single main loop, for example).

Similarly, Scrall rejects any object oriented assumptions since it deals directly with the xUML semantics. It doesn't support object pointers, for example, since there is no notion of "memory" in the relational world. Instead, functions are used for mapping elements. There is no support for null values since, in set theory, an element either exists or it doesn't.

There is of course no problem with a given implementation built on such concepts, but they have no place in a platform independent system specification!

Inspiration

The key inspirations for Scrall are the Small (Shlaer-Mellor action language) proposal, C.J. Date's writings on relational theory (see bibliography) and Andrew Mangogna's Rosea and Micca model compilers. (If you write your actions in Scrall, the transformation into Rosea or Micca statements which generate C code is straightforward).

Additionally, years of frustration with the inherent limitations of BridgePoint's (xtUML) Object Action Language (OAL) have been a driving force. That action language as well as its British cousin Action Specification Language (ASL) were a great first step, but neither evolved much beyond their original formulations. Each of these first generation action languages were a great step forward in their time. But they only scratched the surface and never unlocked the deeper semantics that Sally Shlaer and Stephen Mellor originally envisioned. Each of these languages was introduced with unnecessarily restrictive type systems and wriggled out of the restraints over time with hopelessly ad hoc type extensions.

Scrall takes C.J. Date's approach where the type system is considered to be orthogonal to the relational system. Thus Scrall washes its hands entirely of providing a type definition language facility. Scrall does, however, provide a syntax that accommodates arbitrarily complex types and the operations supported on those types.

The name

It stands for Starr's Concise Relational Action Language with a gratuitous 'l' to guide the pronunciation (sounds like "awl" not "Al"). Many years of scrawling out pseudo-code actions on the whiteboard are the inspiration for the pronunciation.

The logo

It's a simple language. If a chicken can scrawl, so can you!

Status of Scrall as a Language

Until now, the work on Scrall has been focused on developing a consistent and lean concept for expression, making the language practical for reading and writing and ensuring that the xUML semantics are fully supported.

The next step, in progress as of this 1.0 publication, is to specify a formal EBNF grammar. This effort will no doubt lead to a number of syntax adjustments as points of potential ambiguity are uncovered.

That said, we encourage you to begin using Scrall as a pseudocode for writing xUML activities. Feel free to report your progress or any issues you encounter. You can use Scrall in the comments of your action language to clarify what you are trying to accomplish. We took this approach in our book Models to Code. You should be able to accomplish in just a few lines what may require numerous awkward lines in first generation xUML action languages. Hopefully, thinking in the set, relational and concurrency terms of Scrall will make it easier solve problems and build more powerful models.

Terminology

Here's some terminology used throughout this document.

RHS and LHS

These refer to the right and left hand sides of an assignment expression.

MX domain

This is the model execution (MX) domain. The MX domain "understands" xUML and is able to run the models efficiently on some class of platform such as embedded, cloud, etc. The book Models to Code explains the concept in depth.

Signal / Event

The greater UML standard defines both signals and events. A signal is something you send and an event is something that happens. One kind of thing that can happen is the arrival of a signal delivered to some instance. A time event is another kind of event, for example, in the greater UML.

In xUML, the only kind of thing that can happen is an event. And all events are triggered by the delivery of a signal. Time isn't special, it is just the arrival of a signal at a designated time or some delay.

Consequently, in xUML, the terms signal and event can be used interchangeably.

Local class

When describing a method or state activity, this refers to the class upon which the method or state is defined.

Instance

An instance of a class. Also an object. The terms may be used interchangeably. We generally avoid use of the term "object" as it is so often confused with its implementation counterpart.

Local instance

An instance of the local class. It is typically used when describing the starting point of relationship navigation.

Remote instance

An instance that belongs to some class other than the local class.

Identifier

A set of one or more attributes of some class such that a set of values suppled for each of these attributes will match either zero or one object in the associated class population. For example, Waypoint attributes Latitude and Longitude, supplied with values should select either one or zero Waypoint instance assuming the two attributes, taken together, constitute an identifier.

An identifier is irreducible so the terms “identifier” and “irreducible identifier” are synonymous. This means that for the set of attributes that comprise an identifier, there is no proper subset that conforms to the identity constraint. It is taken from the relational theory definition of a key.

Reference

This is a set of referential attributes that refer to components of an identifier such that all components of the identifier are targets of the referential attributes. More formally, assume we have some identifier with attribute components (i1, i2 .. iN). A reference is a set of attributes (r1, r2 .. rN) such that r<a>, where a is some number of the set element, refers to i<a>. For example, a Mission Following Aircraft may have the attributes Next Waypoint Latitude and Next Waypoint Longitude which, taken together, form a reference to a Waypoint.

Instance reference

A class’s identifier, populated with attribute values serves as a reference to an instance of that class. A value pair corresponding to the Latitude, Longitude identifier of a Waypoint serves as a reference to some instance of Waypoint if there is, in fact, an instance of Waypoint at the specified location.

Referential attribute

An attribute that refers to some identifier attribute of a class on the opposite side of an associated relationship is referential. For any given instance on the referring side, the value of the referential attribute must match an existing value for the referenced identifier attribute for some instance on the other side of the relationship.

Thus Mission Following Aircraft.Next Waypoint Latitude is a referential attribute since for any instance of Mission Following Aircraft, the value of Next Waypoint Latitude must match the value Waypoint.Latitude for some instance of Waypoint.

Super identifier

A super identifier is a reducible identifier. So it is an identifier with one or more additional attributes taken from the same class. By contrast, the identifier we mostly use in xUML is an irreducible identifier in the sense that if any attribute is removed, it is no longer an identifier at all.

In rare circumstances you may want to build a reference to a super identifier instead of an identifier to include some referential attribute that helps us express a constraint. The term "super identifier" is analogous to the relational theory definition of a superkey.

Assume Tail Number is the identifier of the class Aircraft. Given some value for Tail Number, NX1334Q, let's say, exactly zero or once instance of Aircraft will be selected. The same could be said of the super identifier Tail Number + Airspeed. NX1334Q + 340 knts will still yield zero or one instance of Aircraft. But a super identifier is reducible in that we can remove Airspeed and still have a valid identifier.

Multiplicity

The number of instances that can participate in an association. 1:1..* = one to many, 1:1 = one to one

Conditional

This term indicates that one side of a multiplicity could involve zero instances. Thus 1:0..* means that there can be zero instances mapped on the many side. In other words if there is a 0 on a multiplicity side, it is conditional.

Guiding principles

Throughout the design of Scrall, when deciding on one syntax formulation vs. another, we return to the same fundamental modeling principles that have guided the design of the xUML class and state modeling languages.

1) There should be one ideal way to solve a problem, not fifteen.

2) Actions should be easy to write.

3) Actions should be even easier to read.

4) Keep the language lean, when you read an activity you want to be thinking about the logic, not the language.

5) Express the subject matter clearly and do not tell the programmer how to write the code.

6) Use consistent model level semantics.

One way, not fifteen

Many years ago one of us (Leon Starr) proposed the 'ham sandwich' test for a good requirements modeling language. The idea is that if you lock up two modelers in separate rooms, give each a set of requirements and slide ham sandwiches under each door, eventually you should end up with two nearly identical models. If not, what accounts for the difference? If both solutions express identical behavior, yet are formulated with different combinations of modeling elements, this tells us that the language is not lean enough. Naturally, the modelers will choose different names, but that is not a fault in the modeling language. a^2 + b^2 = c^2 expresses the same idea as x^2 +y^2 = z^2 does not suggest that math is too bloated.

This would be a different story if we were expressing a design. Diverse designs could yield the same behavior, but with different performance characteristics. Requirements analysis, on the other hand, should converge on a single ideal solution since design tradeoffs are not considered.

Easy to write

The original Shlaer-Mellor symbols for class models and domain were designed for ease of drawing on a whiteboard. It consisted of simple arrows, ovals and rectangles. UML came along and provided a load of symbology that is near impossible to draw comfortably, folders and ..*s for example. The idea was to encourage us all to rely on expensive tools for model expression.

In an effort to support standardization, Shlaer-Mellor adopted the UML notation, but the original lean notation still makes for fast, readable whiteboard scrawl.

Scrall adhere's to the original ease of writing principle for fast whiteboard/pen scribbling.

Readability

Historically, development languages have taken two approaches to readability. One is to introduce a lot of verbiage into the text. Relatively chatty languages like COBOL, AppleScript and the BridgePoint Object Action Language (OAL) introduce lengthy keyword phrases in an attempt to be readable. The opposite approach is to be ultra-concise employing cryptic keyboard symbols. Old-timers are quick to raise the example of IBM's APL language.

Scrall threads a fine line between these two approaches.

Chatty keywords that you must read and type repeatedly (such as OAL's `select many <> from instances of <> where…` are either kept short or represented by keyboard symbols. Compare:

```
// OAL
select many highAircraft from instances of Aircraft where selected.Altitude >
ceiling;
```

With OAL, keywords, model element names and the variable are mixed together.

```
// Scrall
high aircraft = Aircraft(Altitude > ceiling)
```

In the Scrall example there are no keywords at all. The text is devoted to user specified names.

When possible, the symbols are chosen based on visual mnemonics. The Scrall #= symbol, for example, assigns a table value.

Whereas verbosity imposed by the language itself is avoided, verbal expression by the developer is encouraged. Scrall permits space delimiters in model element and variable names. So a variable might be named **the max altitude**, for example. Class and attribute names can be referenced without having to insert underscores or using camelcase. Camelcase is discouraged since it is a compromise favoring writeability over readability. The use of an underscore trades off in the opposite direction. But spaces are the perfect read/write balance which is why we use them in everyday text.

The obvious problem with allowing spaces in names is that it leaves the language vulnerable to ambiguity with regard to keywords. For each word look ahead in a name we must classify it as either part of a name or a keyword. Thus, Scrall strives to minimize the number of keywords, favoring symbols instead. But we believe that the best readability doesn't come from keyword verbiage. It comes from expressive names.

For example, instead of saying in OAL:

```
select many aircraftToLand from instances of AIRCRAFT where selected.Altitude <
landingAltitude and selected.ReadyToLand;
```

(Seven keywords)

We could say in Scrall:

```
aircraft to land ..= Aircraft( Altitude < landing altitude and Ready to land )
```

(One keyword)

The OAL keywords `select` and `many` are replaced by the Scrall multiple instance assignment operator `..=`. The OAL keywords `from instances of` is implicit in the assignment and `where selected` is replaced by the `()` selection predicate.

In fact, the only Scrall keyword in this example is `and`. So, you know, don't use `and` or `or` or `not` in a name.

Scrall encourages natural reading model element names. Compare the code like names `Air_Traffic_Controller` or `AirTrafficController` to the more natural `Air Traffic Controller`.

The parsing issue can be simplified somewhat with a two-pass solution. A first pass through the action language text can shrink-wrap all of the names, by say inserting underscores or camel-casing prior to feeding the result into a more conventional parser.

It's a little more complicated than that since you still need to follow grammar rules to ensure that names are correctly isolated. But, if we can pull it off, the reward of a highly readable and writeable language should be worth the effort.

To speed development, we may find ourselves omitting this feature and living with underscores and camelcase for a while until we solve the parsing problem.

Express requirements

That said, the whole purpose of xUML is to express requirements, not prescribe an implementation. This purpose amplifies the need for clarity of expression. We model the required processing with an absolute minimum of language artifact. Like the other xUML model facets (domain, class, state) we want the statement of requirements to shine through for critical evaluation without noisy language artifact stealing the spotlight.

Use consistent model level semantics

Scrall operates on model level semantics. Period. There are no special blocks where you can insert C or Java code.

The primary reason for this is to ensure that a change in platform technology, such as choice of programming language, has no impact on the models, especially inside the activities.

Executable UML Semantics

To use Scrall effectively, it is critical to understand the underlying model semantics. This is because Scrall is designed to manipulate elements of the model defined by those semantics. There are already other references that define Executable UML semantics. Here we expand on those aspects key to understanding how the action language works.

Class Model Semantics

xUML class model semantics are an application of relational theory for the purposes of system and requirements analysis. And by "theory", we're referring to a branch of mathematics founded on set theory, functions and predicate logic. This is distinct from relational databases, which are an application of relational theory for quite different purposes. Whereas an RDBMS will be manipulated with some variant of SQL (Structured Query Language), our action language will instead be based on C.J. Date's Tutorial D language. The advantage of this approach is that it gives us a simpler, leaner, more mathematical set of operations and structures for accessing and manipulating data. For this reason, Scrall doesn't incorporate the awkward SQL'ishness of tool supported Shlaer-Mellor action languages. Instead we can take advantage of powerful nested expressions and other easily assembled building blocks.

Here is a brief summary of relational semantics relevant to action language.

A class is a named set definition. In the relational world this corresponds to a relvar (relational variable). A class defines a set of ordered pairs (a, t) where a is an attribute and t is a type (data type). We often visualize a class as a table header with a column for each (a,t) pair.

Every attribute has an associated type which establishes the finite set of possible values that may be assigned to that attribute. Rather than refer to (a,t) pairs, from now on we'll just refer to attributes with the understanding that each has some assigned type.

An instance supplies a value for each attribute of a class defined table header. We usually visualize an instance as a row in the table. Row order and column order have no significance since no ordering has been defined on the corresponding set elements.

An identifier is a set of one or more attributes belonging to the same class such that values supplied for those attributes result in the selection of zero or one instance of the class.

Every class has at least one identifier.

An identifier is understood to be irreducible such that if any attribute is omitted, it is no longer an identifier.

A super identifier is an identifier combined with one or more additional attributes from the same class. These are usually not marked unless you use them to enforce constraints in a relationship.

A table consists of some header with a set of attributes. This includes the possibility of an empty set! A zero-attribute table is not only possible, but useful as an intermediate result in an activity. For example, sometimes you restrict an instance set based on some criteria, but don't project on any of the attributes. The returned result is a zero-attribute table with either one or zero instances which corresponds to a true/false meaning. (You can't have more than one empty instance returned since duplicate rows are never allowed in a table).

Note that a table can be created from a class by filling it with all attributes of that class and all instance data. It is useful to construct temporary non-class tables during the processing within an activity. For example, you might want a table of all Aircraft with only tail number and altitude values as part of a computation.

A tuple is a row of values in a table. In other words, it is a set of values where each corresponds to a header attribute-type pair.

An instance reference set is a table whose header is defined by an identifier with any number, including zero, of associated tuples / rows.

An instance variable is a variable that holds as its value an instance reference set. So an instance variable represents a set of instances of some class including possibly the empty set.

A table variable holds a set of tuples / rows corresponding to a header which may or may not correspond to any particular class. Tables are often formed by taking subsets of a class's attributes and combining them across relationships with other class attribute subsets.

A scalar variable holds a single non-table value such as an integer, or arbitrarily complex structured value. A scalar is considered opaque in the sense that it cannot be accessed or parsed via table (relational) operations. Type specific operations, however, can be defined which can access designated components or apply unit conversions.

All relationships on the class model are formalized with referential attributes.

Actions are supported which make it possible to select and manipulate instance data and traverse relationships to find related instances.

State Model Semantics

With regard to state models, action language provides the ability to send a signal, possibly delayed, to a state machine instance and to access any parameter values received with a signal.

A signal must be directed to a single state machine instance.

A state machine instance is either a class instance or an association. In the case of a class instance, the state machine instance represents the lifecycle of that instance. For example, an aircraft instance takes off, flies around and then lands. An association state machine represents an assigner which manages competition on the association. For example, association R7 between classes Missile and Target executes an algorithm to bind two together since it wouldn't be possible to let a Missile instance choose a Target or vice versa without resource locking or race conditions occurring. The assigner provides a necessary single point of control. So, yes, you can send a signal to an association (assuming it has an assigner state machine) in xUML.

An assigner is either single (identified by the association only) or multiple (identified by both the association and an instance of the assigner's partitioning class). In the multiple assigner case, imagine that a Missile can track only those Targets within some Tracking Zone. In that case we need a separate assigner state machine instance for each Tracking Zone (the partitioning class). Each of these state machine instances is on the same association.

In the case of a lifecycle, an event is directed at a single class instance.

In the case of a single assigner, an event is directed at the association (R number).

In the case of a multiple assigner, an event is directed at the association with the target state machine identified by an instance of the partitioning class. In the Missile/Target example, we could send a signal to the R7 assigner state machine instance identified by Tracking Zone 22, let's say.

Action Semantics

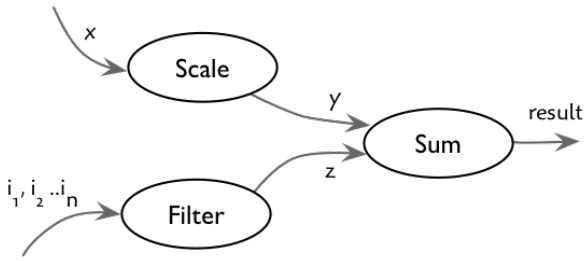
The following text is excerpted (and modified) from the book Models to Code:

Platform independent sequencing

To be truly platform independent, a model should not specify a particular sequence of computation unless that sequence must be enforced on every potential target platform. Here is an example of an arbitrary computation sequence:

```
y = scale(x)
z = filter(i1, i2, ... in)
result = y + z
```

Depending on the implementation, actions 1 and 2 could be reversed or executed concurrently. Action 3, on the other hand, must wait for both actions 1 and 2 to be complete. If the action sequence as written is intended to indicate a required sequence of computation, the model is unnecessarily limiting implementation choices. This breaks the principle that the model must specify only what is required on all potential platforms. By breaking that principle, the model loses a bit of credibility. "What else in the model might I ignore?" the implementor now begins to think!



Each action is represented as a circle, and we interpret each action to be runnable when all of its inputs are available. Action 3 must therefore wait until both actions 1 and 2 have produced output. This data-flow view of computation eliminates the statement of arbitrary sequencing. This is why the data flow is our fundamental view of algorithmic processing in xUML.

Unfortunately, text representations are faster and easier to edit than graphical representations of data flows. In Chapter 2 of Models to Code, we showed a data-flow diagram of the Logging In activity from the Air Traffic Control model. The difficulties of dealing with data-flow graphics and specification of the data-flow processes have meant that all translation schemes of which we are aware use a text-based language to specify activities.

The use of a text-based action language does not preclude having data flow semantics in the language.

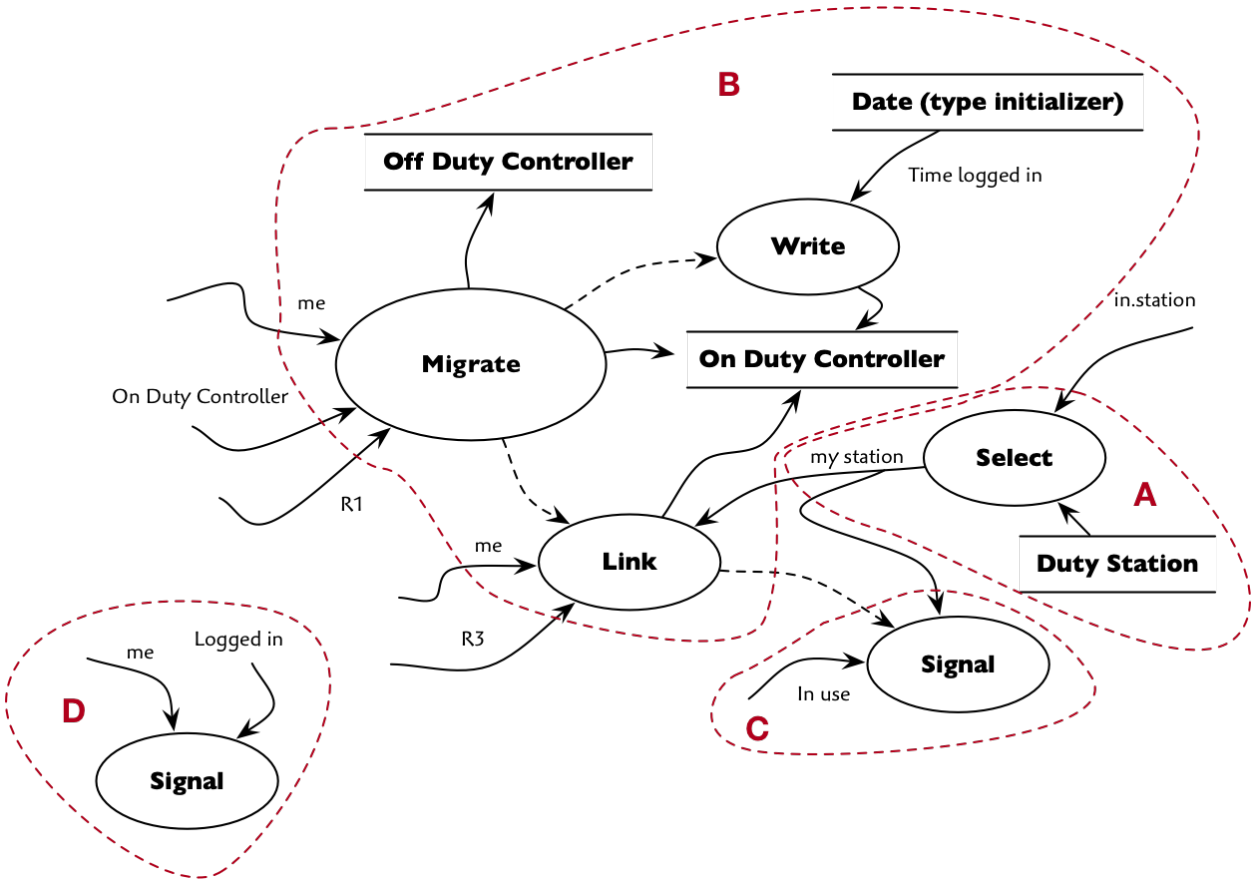
The same data flow diagram in Scrall might like this:

```
y = Scale(in.x)
z = Filter(in.ivalues)
result = y + z
```

Here we assume that Scale and Filter are methods of the local class. Both x and ivalues are passed into the activity as parameters. The ivalues could be either a scalar value typed to hold a set of values or it could be a set of instance references if the activity not a state activity. Since y and z are on the LHS of assignment expressions in this activity, we know that the third line cannot execute until the first two are complete. Since lines 1 and 2 have no data dependencies with one another, we know that they can be executed concurrently. In sort, we have enough information in the text to regenerate the essential data flow graph.

Expressing data and control flow dependencies in Scrall

Now let's take a look at one of the activities defined in the Air Traffic Control example from the Models to Code book.



The activity must migrate an instance of Off Duty Controller to an instance of On Duty Controller.

From the DFD we can see that there are three things that can be accomplished concurrently within the activity. (Because they have no data or control dependencies)

A) Get the instance reference

B) Migrate the instance, linking it to the Duty Station

C) Signal the Duty Station

D) Signal the local instance (me)

(A & D) can both go first since each has all of its data available at the start

(B) can activate after (A) now that it has an instance reference to an On Duty Station (my station). The migrate >>, link & and write actions must be combined so that we don't end up with any null attribute values.

(C) now the Duty Station instance can be signaled since it is now related to the local instance (dashed arrow from Link to Signal action)

We can express all this in Scrall as shown:

```
my station .= Duty Station( Number: in.station )

>>On Duty Controller(Time logged in: Date.HMS) &R3 my station <1>

<1> In use -> my station

Logged in -> me
```

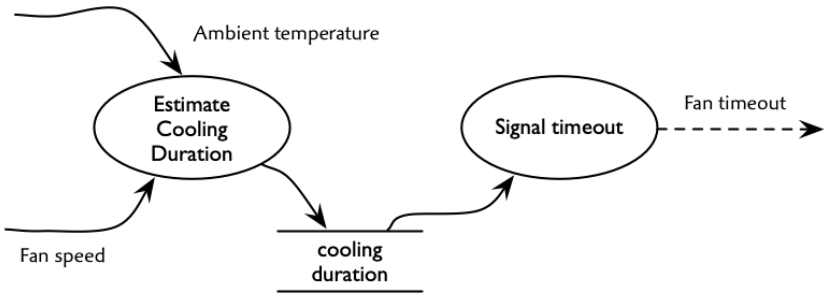
The first and last lines above can execute immediately since each has all of its data available at the start. The second line migrates the instance, sets the login time and links it to the Duty Station all in one expression. It can execute as soon as the `my station` variable is set. Once this expression completes it sets the named control token <1> fulfills the purpose of a dashed control flow in the diagram. With <1> set, the third line can execute and signal the **In use** event. The control token is named just like a variable, so it could have been something like <done> or <migrated> instead.

Regarding the expression in line 2, note that we have two ways of expressing a data flow. One way is through an assignment, as we do in line 1. The other is through nested expressions and compound actions. All attributes, including referential attributes, must be given legal values upon instance creation. Nulls are forbidden in xUML! Since the migrate action deletes an instance in one subclass (Off Duty Station) and creates one in another (**On Duty Station**), it must ensure that all attributes are set correctly. The action, therefore, includes both the attribute write and the **On Duty Station** link on **R3**. To prevent the newly linked station instance from peeking at incomplete data, we use a control token to sequence the **In use** event.

There's no harm, by the way, in sending the **Logged in** event right away since it is directed to the local instance (me) where the dispatched event will be held until the entire activity is complete.

Variables

When we draw an activity's data flow diagram, an action may output one or more intermediate results which are later referenced as inputs to one or more other actions in the same activity.



Named flows or data stores may be represented by temporary variables in action language. Input to a store may be represented by an assignment statement with the variable on the LHS. Output from a store may appear in text as a variable reference within one or more nested actions.

Variables are not necessary when the output of one expression is nested inside of another.

Categories of variables

There are three general categories of variables:

- instance set (ex: set of Aircraft instance references)
- table (ex: altitudes and headings)
- scalar (ex: position, distance, pressure)

An instance set is zero or more instance references on a single class. A table is a relation as defined in relational theory, a header row of zero or more attribute:data type pairs and a body of zero or more tuple rows where each row supplies a value for each header column. A scalar is a value with an opaque (not-necessarily-relational) internal structure such as an integer, coordinate pair or any arbitrarily complex data type.

Types

A "type" is a definition of a set of values and the operations supported on those values. The type of an instance set is determined by its class name. The type of a table is determined by the table header. The type

of a scalar is defined using a type definition system (not Scrall) to describe a set of values and supported operations on those values.

For both the instance set and table categories a standard set of operations given by theory is already defined on each. For scalars, a base set of operations is provided, but these can be extended and redefined and customized as the modeler sees fit.

Some general principles apply to all variables regardless of category and type.

Scope

There is no such thing as a global variable in xUML. Every variable is local to the activity in which it is defined. To persist a value outside the scope of an activity, write the value to an attribute of some instance or send the value as an output parameter in a signal or domain interaction.

Definition

The category of a value is defined implicitly. It can be inferred by the choice of assignment operator and/or the category of the source on the RHS of the assignment.

The type can usually be inferred unambiguously, though in some cases it may be necessary to explicitly declare the type on the LHS of an assignment. Once a variable has a value assigned it can take on new values, but its category or type may not be changed.

Variable names

Names are case insensitive in Scrall. This is to discourage the use of confusing naming patterns such as `current Altitude` and `Current altitude` meaning different things.

There is no need for a `self` keyword in Scrall to avoid name collisions. Instead, model element names in the local context take precedence. Consequently, a variable name may not conflict with the local class name or any of the local class's attribute names.

So if you are defining an activity on the **Aircraft** class, you cannot define an `aircraft` variable since it would collide with the local class name and be interpreted as such. But you might use the name `tracked aircraft`, `my aircraft`, or even `the aircraft`; "my" and "the" are not keywords. Similarly, if there is an **Aircraft.Altitude** attribute, you cannot define an `altitude` variable, but you might try `current altitude`. Be careful, however since a `Current altitude` attribute later added to the class will cause a naming collision in your action language.

A variable holds a value at all times

There is no concept, in Scrall, of an uninitialized, undeclared or empty variable. These are implementation concepts tainted with the mechanics of memory management. In the world of sets, a thing either exists or it doesn't!

There are no "null values" in Scrall. By definition, null is not a value anyway. Empty sets, however, are values and may be used with table and instance set variables.

Instance set variables

An *instance reference* is defined as a set of one or more values corresponding to each attribute of a class identifier. There is no concept of an internally managed handle or object id as these would leak implementation artifacts into the language. Set theory does not define or specify handles.

So the only way to refer to an instance of a class is by supplying a complete set of values for one of that class's identifiers. Later, when we talk informally about assigning an instance to a variable we really mean assigning an instance reference.

An *instance set variable* holds a set of zero or more instance references on a single class.

An instance set variable is implicitly declared to refer to instances of a particular class by virtue of its initial assignment.

Instance reference assignment

The `.=`, `..=` operators assign a value to (flow a value into) an instance set variable.

The intended cardinality (number of instance references) is signified by the choice of assignment operator. We say "intended" because the choice is generally determined by the operation on the RHS which produces the value. For readability though, it is helpful to indicate the desired cardinality so that, at a glance, the reader grasps the intended content of the variable. It is also helpful to force the modeler to be as explicit as possible about their intentions to reduce the likelihood of error.

We should be clear that regardless of what cardinality is initially assigned to an instance set variable, that variable is always capable of holding zero, one or many instance references.

In all cases, the LHS is an instance set variable or an input flow expecting the intended cardinality.

Let's examine each of these operators.

Assign zero or one instance

The `.=` assignment (assign-zero-one) operator requires an expression on the RHS which produces one or zero instance references. An error will result if the expression can (or does) yield more than one instance.

You would use this operator when you traverse a relationship path along a 1:1 association. Or, you would use this operator if you select an instance of a class with an identifier. You should not use the assign-zero-one operator in any case where you could legally select more than one instance.

Example: single instance assignment

Selection using values supplied for an identifier is a common idiom in Scrall. Consequently, the syntax has been streamlined for this action.

Here, a local **Pilot** instance locates its related **Aircraft** instance using an `aircraft id` value passed in as an input parameter.

```
my aircraft .= Aircraft( Tail number: in.aircraft id )
```

Assuming that an identifier of the **Aircraft** class is **Tail number**, it is known that the selection must yield zero or one instance. Use of the **.=** assignment operator implicitly defines `my aircraft` as an instance set variable. The type of the variable is established by the class name used in the selection expression on the RHS.

If the aircraft is not found, zero instances are assigned and the `my aircraft` variable holds the empty set *of Aircraft instances*. You cannot, in some later assignment, assign instances of a different class to this variable. The empty set evaluates to false in a predicate:

```
my aircraft .= Aircraft( Tail number: in.aircraft id )
my aircraft? {
   // at least one aircraft
} : {
    // no aircraft
}
```

Or you can explicitly compare against the empty set of Aircraft instances:

```
( my aircraft != Aircraft(0) )? { // if not empty
}
```

In this selection example the class has a multiple attribute identifier:

```
assigned runway .= Runway( Number: directed runway, Airport: local airport code )
```

Once again, one or zero instance will be assigned.

Assign zero, one or many instances

The **..=** assignment (assign-many) operator assigns any number of instance references, including zero, from the same class.

You might use this operator to select instances based on criteria that could apply to multiple instances. You would also use this operator if you traverse associations that lead to multiple instances.

You can use **..=** in any situation where a **.=** would work without any error occurring. But it is preferable to be as precise as possible about your intentions.

Example: Multiple instance assignment

Multiple instances may be selected and assigned with the **..=** operator.

```
low flying aircraft ..= /R3/Control Zone.Aircraft( Altitude < low alt )
```

All instances of **Aircraft** whose **Altitude** attribute value is below the specified altitude are assigned to the `low flying aircraft` variable. This could be zero, one or many instances.

The **Aircraft** selection below may return zero, one or many instances:

```
aircraft to land ..= Aircraft( Zone: Code, Altitude < landing altitude and Ready
to land )
```

In the statement above, the set of all **Aircraft** instances such that each instance has a value for its **Zone** attribute matching the value of the local instance's **Code** attribute (think of it as self.Code, if that helps) and an **Altitude** attribute value less than the value of the `landing altitude` scalar variable and a true value for the **Ready to land** attribute is assigned to the LHS.

Assign at most one

To assign at most one, just use the `.=` assignment operator in conjunction with a selection quantity of at most one.

Example: Arbitrary instance assignment

To select any instance of the Aircraft class or zero if none exists, you can specify this assignment:

```
some aircraft .= Aircraft(1) // select at most one
```

Here are some examples of sorting based on some criteria and then choosing one of the highest or lowest ranking instances:

```
some high flying aircraft .= Aircraft(1, Altitude: > high alt )
lowest fastest aircraft .= Aircraft(1, ^-Altitude, ^+Airspeed )
fastest lowest aircraft .= Aircraft(1, ^+Airspeed, ^-Alititude )
```

In all of these examples the RHS may yield zero or multiple instances.

Assigning the empty set

Using any instance set assignment operator, you can initialize an instance set variable to an empty set of some class as shown:

```
pilots aircraft .= Aircraft(0)  // Empty set is assigned
```

In this case the single instance assignment operator was used since it reflects the usage of the variable on the LHS. Remember though, that the LHS variable can only be assigned instance references for Aircraft in the future since the variable is declared, implicitly, as holding Aircraft instance references.

Table variables

A *table variable* holds a value referred to in relational theory as a *relation*. A relation comprises a *heading* consisting of zero or more attribute:data type pairs and a *body* consisting of zero or more tuples, where each tuple supplies a value for each element in the heading.

Here is an example relation drawn as a table:

Pressure::PSI	Temperature::DegC
2500.00 PSI	87.3 C
1009.98 PSI	32.3 C

Table variables are useful for selecting, updating, computing and otherwise manipulating class model data. Standard relational operators such as restrict, project, join, union, intersect, extend and subtract are supported in Scrall. These operations are all closed under relations so that the result of any relational operation is another relation. Nested relational expressions are then possible.

For example, you can use tables to grab a subset of instances of some class based on some criteria across a subset of the attributes of those instances, then join those across a relationship path to some subset of attributes of related instances in a destination class and then further subset your columns and rows until you arrive at a single piece of data as an answer to whatever query you originally intended. In many cases you can do the same by merely navigating associations using instance references only. But there are times when you want to perform a powerful and precise operation that can only be neatly handled with tables.

First generation xUML action languages did their best in the late 90's-00's to camouflage the details of relational mechanics to make it easier to appeal to the sensibilities of object oriented developers. It didn't work.

But it is precisely those relational mechanics that give xUML its platform independent computational power. Scrall provides powerful instance selection and navigation for the everyday computational tasks. The casual modeler can do a lot without having to learn relational algebra. But Scrall also supports the power modeler by unleashing the full relational beast when you need it to tackle complex computations.

More about relations

While a relation can be viewed as a table, it is more accurately defined in terms of sets and ordered pairs. We start with sets H and B representing the heading and body components. Set H is a set of zero or more ordered pairs such that the first element of an ordered pair is an attribute name and the second element is a scalar data type.

In our example it is: `H \{ Pressure ; PSI, Temperature ; Degrees celsius }`. As this is a set, no ordering is implied and there can be no duplicates. In other words, the same attribute name cannot appear twice in a header. A body is a set of zero or more tuples where each tuple holds a value corresponding to each attribute in the heading. Our example relation has this body: `B { T { ( Pressure ; 2500.0 ), ( Temperature ; 87.3 ) }, T { ( Temperature ; 32.9 ), ( Pressure ; 1002.98 ) } }`.

Note that the above ordering doesn't appear to match the table layout. That's because sets are not ordered, thus the pairing of attribute and value names. The table representation is only a helpful visualization of the sets in a relation. The implication that there is a row and column order is a misleading artifact of the table view. Always remember that the uglier set definition is the true representation and that a table is just one way of viewing a relation.

That said, we will use the term "table value" to be synonymous with relation for ease of discussion. But remember that row and column order has no modeling significance.

Assignment

Table values are assigned using the #= table assignment operator. The hashtag symbol was chosen because it resembles a table.

Here is an example assignment:

```
tanks #= Tank( Temperature > in.Max temp ).( ID, Pressure,  Temperature )
```

In this example all **ID**, **Pressure** and **Temperature** values for Tank objects with a temperature above the supplied `in.Max temp` are assigned as a table value in the `tanks` variable. As a result, the value of tanks could be something like:

```
H { ID ; Nominal, Pressure ; PSI, Temperature ; Degrees celsius }, B { T { ( ID ;
12 ), ( Pressure ; 2500.0 ), ( Temperature ; 87.3 ) }, T { ( Temperature ; 32.9 ),
( ID ; 22 ), ( Pressure ; 1002.98 ) } }
```

This is, of course, a lot easier to visualize as a table:

ID::Nominal	Pressure::PSI	Temperature::DegC
12	2500.00 PSI	87.3 C
22	1009.98 PSI	32.3 C

Here, a single attribute is extracted from a related class:

```
dogs owner #= /R7/Owner.Name  // across to-one relationship
```

If **R7** in the example above navigates to a one, unconditional side, we know that the resulting table has one row. If, it were conditional, then a table with zero rows could result.

Below, all the attributes of the **Dog** class are selected using the ∗ operator for the **Dog Owner** instance.

```
my dogs data #= /R7/Dog.(∗) // all data about all related dogs
```

Since traversal is to a one-many unconditional side, the resulting table will have at least one row.

The `(*)` select-all case is implicit, so you can shorten the previous expression with the same result:

```
my dogs data #= /R7/Dog // all data about all related dogs
```

Table construction

A table value with one row can also be constructed from scalar variables using the `#[]` table constructor expression.

```
aicraft speed #= #[ ID: my ID, Airspeed: my airspeed ]
```

True and false table values

You can assign an empty table with no columns and no rows to a table variable like this:

```
tabledum #= false  // Now the value is:  H{}, B{}
```

This kind of table is a relational false value.

To do the same, but with one row, do this:

```
tabledee #= true // Now the value is H{}, B{ T {} }
```

In this example a table with zero columns and one row is assigned.

This table is the relational true value.

You cannot assign more than one row since they would then be duplicates and duplicate rows are not allowed. So, with zero columns there is either zero or one empty row.

How can all this be useful?

An empty heading and an empty body is known as *table dum* and can be interpreted as false:

```
H {}, B {}
```

Let's say we have an **Aircraft** class and we want to know if any instance of **Aircraft** is flying below some floor altitude, say 300 m.

```
low aircraft #= Aircraft(Altitude < floor altitude).()
```

The expression above selects all aircraft below the floor, and specifies no attributes to be returned in the second set of parenthesis. If there are no aircraft found, a table with no columns and no rows will be returned. In other words, false.

Now let's say that several low flying aircraft are found. But, again, no attributes have been selected. So for each low flying aircraft, the tuple `T {}` is returned. Since these are all duplicates and duplicates are not allowed in a set, the duplicates are discarded to return the following relation:

```
H {}, B { T {} }
```

Here we have zero columns and a single empty tuple. This is called *table dee* which can be interpreted as true.

So you can do this:

```
low aircraft #= Aircraft(Altitude < floor altitude).()
low aircraft? { ...
}
```

You can also convert a class selection into a table value by putting a # operator in front of the class name:

```
#Aircraft(Altitude < floor altitude)? { ...
}
```

But you don't need to do the conversion since an empty instance set is also interpreted as false. So this also works:

```
Aircraft(Altitude < floor altitude)? { ...
}
```

Single attribute scalar conversion

You may want to convert the result of a table operation into a single scalar. You have to be careful to ensure that the cardinality of your table is one. A zero tuple or many tuple table will cause a scalar assignment to fail.

```
highest alt speed = #Aircraft(1, +^Altitude).Speed // ERROR
```

In the above example, the table on the RHS will be empty if there are no instances of Aircraft and the assignment will fail.

Instead, do this:

```
highest aircraft #= #Aircraft(1, +^Altitude)
highest aircraft ? highest alt speed = highest aircraft.Speed : no aircraft -> me
```

Here we first construct a table with one or zero rows to represent the highest aircraft. Then, if one exists, we extract the speed value. Otherwise, a signal is sent to the local instance.

This is just one way to handle the situation, but you must ensure that any scalar extraction is taken from a single row table projected on a single attribute.

If modeled constraints are respected, you don't necessarily need any special notation to guarantee the selection of a single tuple.

```
assigned runway = queued aircraft( Tailnumber: ID ).Taking off from
```

In the above table to scalar conversion example, we use our local class's ID attribute value to locate a runway ID. If the query on the right returns zero or many tuples, the action will fail. It is the modeler's re-

sponsibility to ensure that the query will always return a single tuple. You can't tell from the above action language, but, if the **queued aircraft** table includes the local class instance (Aircraft) and hence the local Aircraft.ID and there is an unconditional to-one association with Runway, it may be the case that a single tuple will always be returned. If so, the reasoning should be clarified in the comments.

Multi attribute scalar conversion

If a scalar data type specifies multiple components, you can convert from a table with multiple attributes. You just need to rename the table attributes so that they match the corresponding scalar type components using the >> rename operator.

```
location::Point = some waypoint[lower x>>x, lower y>>y].(x, y)
```

Here we have a scalar variable of type Point which has an **x** and **y** component in the type definition. We also presume that those components are defined to be settable (= assignment operator supported). Note that we must explicitly specify the data type of the scalar variable in this assignment.

Scalar variables

A *scalar variable* is similar to the familiar typed programming language variable. The term *scalar*, in this context, indicates that a corresponding value appears to be atomic from a relational point of view. The only way to get at any internal structure is through the use of type specific (non-relational) operations.

The type definition system for scalar values is entirely orthogonal to the xUML modeling language. Once defined, a type may be associated with one or more attributes in the class model.

The familiar unconstrained mathematical types such as Boolean, Integer, Real, Rational and String are available as system types. These are used as a basis for specifying more constrained types. Whereas the set of all possible String values of a given maximum length is quite large, the set of all legal ICAO Airport Codes is much more constrained. So a type like **ICAO Airport Code** could be defined as a constraint on the base type **String**.

There is no limit to the complexity or structure of an unconstrained type. You could, for example, define a type called **Video Image** if you like.

Note that a scalar type is defined not just by defining a set of values and a structure. Operations supported by the type must also be defined.

If you have a sufficiently complex type that supports an extensive set of operations on values of that type you might consider modeling it out in some domain.

Scalar value assignment

Use the = assignment operator to assign and implicitly declare a scalar variable.

```
cabin location = /R1/Cabin.Floor
```

Here, the type defined for the `Cabin.Floor` attribute establishes the type of the `cabin location` scalar variable. We know it is a scalar variable because the = assignment operator was used.

If the RHS specifies an attribute of a class, you must ensure that a single instance (not zero or many) is selected, otherwise a fatal error occurs. If, for example, the relationship to **Cabin** across **R1** is 0..1, an error will occur.

It is the modeler's responsibility to take the class model into account when writing action language. In the case of a 0..1 access, it is important to consider both cardinalities. Otherwise, the modeler can refactor the class model to eliminate the 0..1 access via an association class or a generalization. But there are certainly cases where the modeler may wish to retain the 0..1 access.

A smart action language editor can catch this error prior to runtime by examining the class model.

There are two ways to handle this conditionality in Scrall.

```
/R1/Cabin ? {
   x = /R1/Cabin.Floor
   // other actions
} : {
```

```
    // no cabin case actions
}
```

Or:

```
my cabin .= /R1/Cabin
my cabin? {
   // just use my cabin.Floor instead of x
} : {
   // no cabin case
}
```

Another common case of implicit assignment is via input parameters. Here we assume that there is one integer input i, one real input r, and one PSI pressure value.

```
x = in.r  // x is inferred as real since in.r is a real number
i = in.i // i is integer since in.i holds an integer
c = in.pressure // c is PSI to match the in.pressure data type
```

Scalar element access

Now consider a **Point** and a **Rectangle** data type. The**Point** data type has two components **X** and **Y**, each of type **Position**. The **Rectangle** data type has components **Origin** of type **Point** and **Length** and **Width** each of type **Distance**.

A component, computed value or operation made publicly available on this type may be accessed using dot notation.

```
rects origin = rect.Origin  // Assigns Origin component
```

In the example above the **Origin** component is extracted and assigned to the variable **rects origin**. Since the component is typed as **Point**, the assignment sets the variable to the same type.

If the LHS variable was previously initialized or implicitly defined as some other type, there will be an error on this assignment.

```
x, y = rect.Origin(X, Y)
```

Note that there are two levels of extraction. This is necessary so that the X and Y elements are correctly ordered for assignment to the x, y variables. So, this is also legal:

```
y, x = rect.Origin(Y, X)
```

In either of the previous two examples, the LHS variables will each assume the **Position** type.

Explicit scalar declaration

Any variable not currently in use may be introduced with an explicit type.

There are a few cases where explicit definition of a type is required. Use the `::` symbol to associate a type with a scalar variable. It is required, for example, when you want to assign multiple attribute values. Consider a **Point** type that has components **X** and **Y**, each of type **Real**.

```
pt::Point(X: in.x, Y: in.y)
```

In the example above, two input parameters are assigned using the **Point** type's init operation.

When a scalar variable is explicitly declared it must be assigned a complete value. Uninitialized or partially initialized variables will trigger a fatal error.

```
pt::Point(x: in.x, y: in.y)  // Correctly initialized
pt2::Point // Also ok, gets default initial value
```

Also you can define multiple variables in one line if they are of the same type:

```
pt1, pt2::Point(x: in.x, y: in.y) // both initialized to the same values
```

In the above example `pt1` and `pt2` are both initialized to the same value.

Constants and literals

Most programming languages permit the expression of literal values, `3.14` or `"Untitled"`, for example. Named values such as `PI` or `MAXINT` are also typically allowed.

It is bad form to mix literal or even named values in action language. In practice, model developers do it anyway, but with a few notable exceptions, there are safer and better alternatives.

You often end up with values that are special in the context of your application like, let's say, `7`, `270 degrees`, `2500 psi`, etc. Specifying these as literals or named values within the action language is bad form for many of the same reasons as in program language code. Here's a quick review of these oft documented issues:

Problem 1: Hard to find values

A critical value like `350 degrees` may need to be changed to `425 degrees` someday. If it is in an attribute of some specification class it will be easy to find. Furthermore, the purpose of that limitation could be documented in its attribute description. But if instead the value just pops up in the action language somewhere, how will we find it? And why was that particular value selected?

Problem 2: Hard to ensure uniformity

You might change the literal `2500 psi` in one place, but is it used elsewhere? What if it is used in three places, but you only change it in two? It's only defined in one place? Great, how do you *know*? By instead putting the value in the class model, you can ensure that you have one fact in one place.

Problem 3: Unnecessary re-compilation

As with the class and state models, any edit to the action language necessitates re-compilation of the models. When you change the literal `350 degrees` to `425 degrees` in an activity, you must recompile, retest and reintegrate the portion of the model that you modified. But you shouldn't have to. Instead, edit a value in the instance population data. Let's say that you locate the **Heating Specification** initialization table, scroll down to some instance and then change the value in the **Standard bake temperature** column. The model is unaffected by this change, but you can reinitialize it with the new value. In fact, in a running system, you may be able to effect this change by issuing a runtime edit command without any need to re-compile.

So what do we do in Scrall?

There are, of course, times when you want to perform a computation with a mathematical constant. A value like pi is typically outside the subject matter of your domain, so it doesn't really belong in the class model. By the same reasoning, though, it really shouldn't be defined in the action language either.

So we do need to find a way to define and use mathematical constants as well as special values like "" (empty string). And while a value like, pi doesn't change, the desired approximation might. We also note that the boolean `true` and `false` literals are generally helpful.

And then there is also the question of values for enumerated data types. It wouldn't be unreasonable to have logic in an activity that branches based on the current **landing gear** status like the following:

```
(/R1/Landing Gear.status == .retracted) ?
   do(x) : do(y)
```

And then there's the issue of instance creation. How do you create a new instance without supplying values for each of its attributes?

```
Color = "Blue"
Doors = 4
Speed = 0.0
```

You could argue that the initial attribute values could be gathered from a related specification class such as **Car Specification** with attributes like **Initial color**, etc. But sometimes the specification class solution can be a bit of overkill, especially in smaller applications.

So while there are important exceptions, it is STILL bad form to bury a literal value like **15.2** deep in some activity. And if the action language allows it, bad or lazy modelers are going to keep doing this kind of thing.

Solution: No literals in Scrall (pretty much)

Since there are other ways to handle the exceptional cases, literal values are illegal in Scrall. In particular, a statement like this…

```
desired temp = 350.0  // Not legal!
```

is not legal. We'll return to this example with a better solution, but first let's take into account all the other cases and exceptions.

Instance attribute creation values

First, consider new instance creation. The UML standard already suggests a solution with the initial value field on the class diagram. This is a value applied by default to the attribute of any new instance. This moves default initial values out of the action language and into the class model. When an instance is created, the model execution architecture will use those initial values if no other value is specified.

For example, if you want to ensure that every instance of **Dog** has its **Dog.Breed** attribute initialized with the value "Collie", just put that value in the appropriate class model's initial attribute value definition. Then, when you create a new instance of **Dog**, omit the **Breed** attribute like so:

```
new dog .= &R1 *Dog( Name: in.Supplied name )
```

And the breed will be set automatically to "Collie" since it wasn't specified. The rightmost part of the action links the newly created **Dog** to the local instance, presumably **Dog Owner**, on the **R1** association.

Boolean values

The values `true` and `false` are keywords in Scrall. That said, most of the cases where you might use them have been eliminated. That's because you can use the `set` and `unset` operations defined on the Boolean data type.

Let's say we have a boolean **Shuffle** attribute in a music application which either does or does not shuffle the play order in a song list.

```
song list.Shuffle.set // now its value is true
song list.Shuffle.unset // now its value is false
(song list.Shuffle) ? ...  // You don't need a literal to test the value
```

But, if you want, you can do this:

```
song list.Shuffle = true
(song list.Shuffle == true)? ...
```

There are situations where you will need the boolean value keywords such as:

```
=>> true    // return true
```

Special values

Each domain may require certain special values like `32 degF`, `pi`, `0.0` and so forth. You don't necessarily want to devote specification attributes to these values since they don't really describe your modeled subject matter. But if the values are not specific to your subject matter, they must be specific to something! You could argue, for example, that `32 degF` is specific to the temperature data type. Pi is a mathematical constant. While pi is an irrational number, in mathematics, it is always rounded down in the computer world to some rational value. Zero is a special integer which may be implemented in a variety of implementation types (double, unsigned, etc).

In all of these cases we can make use of a special operation called a "selector" defined on the appropriate type.

Type selectors and type operations

A type defines a set of values that can be assigned and a selector chooses one such value for that type.

Let's say that we want to select the number 2 from the system defined data type **Integer**. We could then define a selector operation named "2" invoked like this:

```
x = Integer(two) // Probably not a great idea
```

Here's the trick, though. You have to go in and define that selector for the relevant data type before it will work. So you shouldn't assume that a **two** selector has been defined unless you've added it yourself in the type definition.

For a more practical example, you could access pi like this:

```
pi = Real(pi)
```

For the temperature example:

```
t = Temperature(freezing h2o).DegF
```

Here we have a selector combined with a conversion operation. The selector is always parenthesized while the conversion operation is accessed via dot notation.

For zero, it depends on your data type:

```
d = Duration(immediate).Seconds // gives you zero seconds
```

If no value is supplied between the parentheses or if the parentheses are omitted, the default initial value defined for the type will be assigned. Given that it is zero for the base types **Rational** and **Integer**, you can do this:

```
rzero = Rational
izero = Integer
```

This trick also works if you want the current time:

```
now = Date.HMS  // current time is the default initial value
```

Note that the **HMS** operation is defined on the **Date** type and is applied to the default current time value. There is no need for a selector since we are assuming we get the default initial value, but we do specify the conversion operation to get the date in the desired format.

The same principle works with non-numeric values. Let's say that you want to create all new instances of **Folder** with the string "`Untitled`". Define that value as the initial default of the **Folder Name** type and you would do this:

```
newFolder name = Folder Name
```

Enumerated values

Since enumerated values are generally limited in number and are often used to test cases, it makes sense to express them in actions. So these literals are allowed in Scrall.

A type might be defined such as **Valve State** with values `{ open, closed }`

And then a variable **position** of that type would be used like so to decide which event to send:

```
position?
   .open : Close -> target valve
   .closed : Open -> target valve
```

Enumerated values are always prefaced with a `.` symbol in Scrall. In circumstances where the data type cannot be inferred from context, the data type must precede the value like this: `Valve State::closed`.

Structure of an activity

To relate syntax to semantics, it is best to visualize an xUML activity as a data flow diagram (see the Model Semantics section). Each bubble on the data flow diagram corresponds to a Scrall action in text. We can then consider the data/control inputs and outputs (arrows in and out) defined for that action. That's certainly not a formal definition and, absent a proper action language metamodel, it's about the best we can do for now.

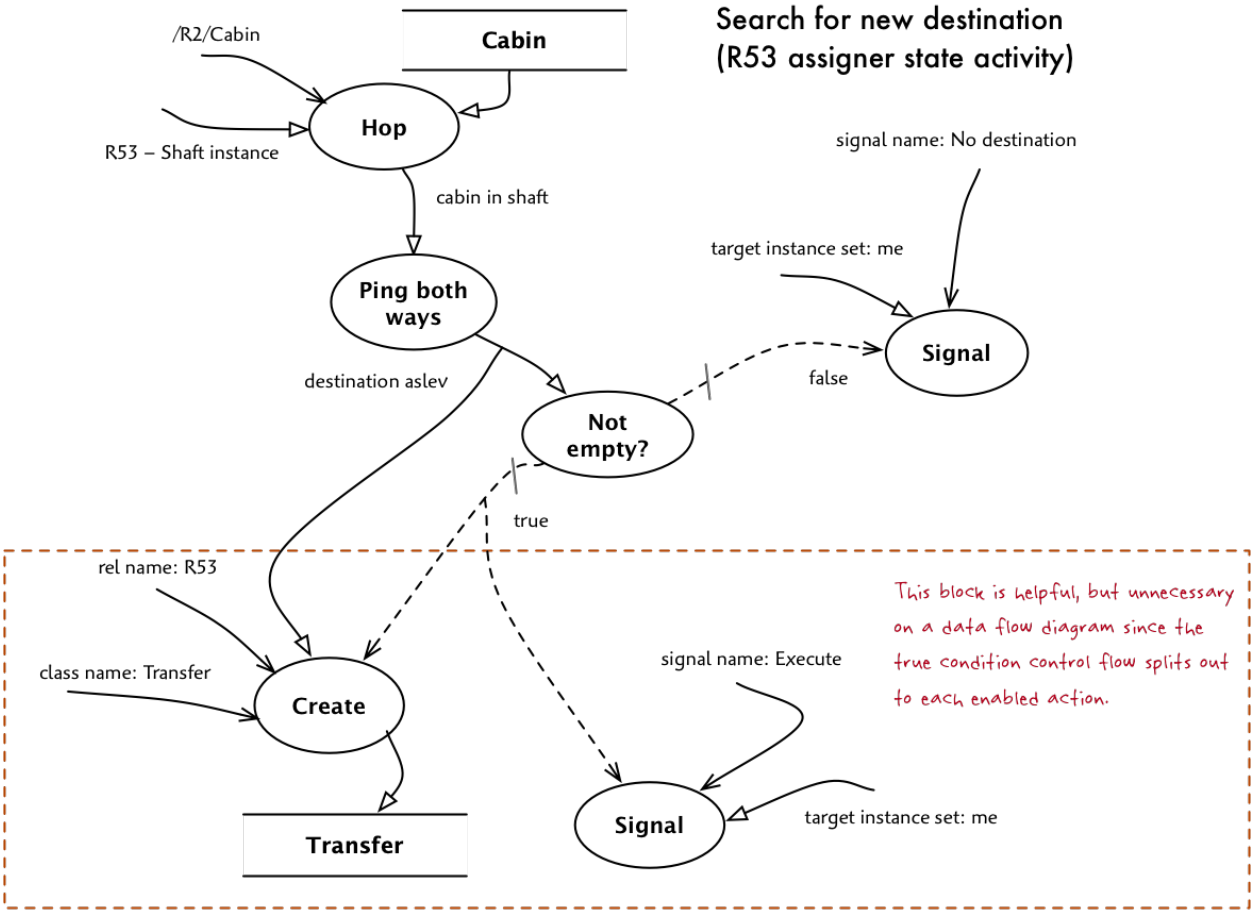
We'll dig a bit deeper to see how Scrall text elements correlate with graphical data flow diagram symbols.

Action block

An *action block* is a set of one or more actions within an activity that may be enabled. The actions sandwiched between an `?:` (if then) clause, for example, constitutes an action block. If the set consists of more than one action, it is generally surrounded by `{}` brackets. Read further to see what exactly is meant by "enabling an action block".

On a data flow diagram, there is no action block concept since you can just follow dashed arrows to see what upstream actions are triggered and then trace the outputs of those actions downstream to work out the complete activity subset triggered by a decision. With text, we need the help of brackets to visualize the same subset.

In the diagram below, two actions are enabled by a true condition. The dashed rectangle isn't needed on a data flow diagram, but it does indicate what we would enclose in brackets to form an action block.



This example shows a state activity from the Elevator 3 Case Study where a search for the next **Cabin** destination (**Accessible Shaft Level**: `aslev`) is initiated. One may or may not be found. Here is the equivalent Scrall:

```
// The Cabin in this Shaft is stationary and has no Transfer
cabin in shaft .= /R2/Cabin
destination aslev .= cabin in shaft.Ping both ways()
destination aslev? {
    Execute -> *Transfer &R53 cabin in shaft, destination aslev
    Transfer created ->
} : No destination -> me
```

Enabling an action block

The execution of an activity begins immediately, upon either state entry or explicit invocation depending on the activity type, state, method, etc. Each action in the activity may execute only when all of its inputs are available. An input to an action is either a data flow, as represented by a variable or a class model access, an input parameter designated in the activity signature, or a sequence token. Upon invocation, all

input parameters are considered available. The class model content is always available. A variable is available once it is initialized with a value. A sequence token is available upon completion of an associated action block.

If an individual action is enabled, it means that the action has all the data it needs and is ready for execution.

An action block is considered enabled when all input and sequence tokens are available to enable one or more triggering actions within the action block. A triggering action is an action within an action block that can execute as soon as the action block is enabled. All non-triggering actions in that action block must be enabled by data or sequence tokens originating within that same action block. In other words, once an action block is enabled, it is assumed that all of its content will ultimately be enabled.

Implicit data variable sequence

All sequencing within an activity is determined exclusively by data availability. The vertical order in which the actions are written, for example, has no relevance to the sequence of execution. To process an activity during runtime, it is necessary to convert an activity into a data flow graph to determine the essential sequencing. In most cases, it is possible to work this out by determining when data variables have been initialized and when they are accessed. If, however, a data variable is accessed more than once, it may be necessary to be more explicit about the data dependencies.

One action may alter instance data in the class model which serves as input to a subsequent action. Since no data variables are involved, it is necessary to explicitly sequence the actions. This is done on a data flow diagram using a dashed control flow. In action language, we can set a sequence token to represent the same idea.

Explicit control token sequence

A control token is any name between angled <> brackets and is set upon completion of one action block. When set, it is placed as the rightmost element after the end of an action block; either to the right of a single action or to the right for a closing } bracket. It can then be seen as an activating input by placing it as the leftmost element of some other action block.

See the example in the Action Semantics section for an example.

Conditional sequence

Activation of an action block may depend on the result of some prior evaluation. This can be accomplished by sandwiching the action blocks within an if-else action or through the use of guards.

The if-then-else statement follows the usual programming language format where:

```
(<predicate>)?
    <action block> : <action block>
```

For example:

```
(x > y)?
```

```
doA : doB
```

Statement

The concept of a statement and a line are purely textual grouping artifacts having nothing to do with data flow semantics. A statement is a segment of a data flow diagram typically starting with a flow or process and ending with another action or group of actions. A single statement may be spread across multiple lines of text.

A group of actions is collected together in an action block as indicated by `{}` braces.

Single line action

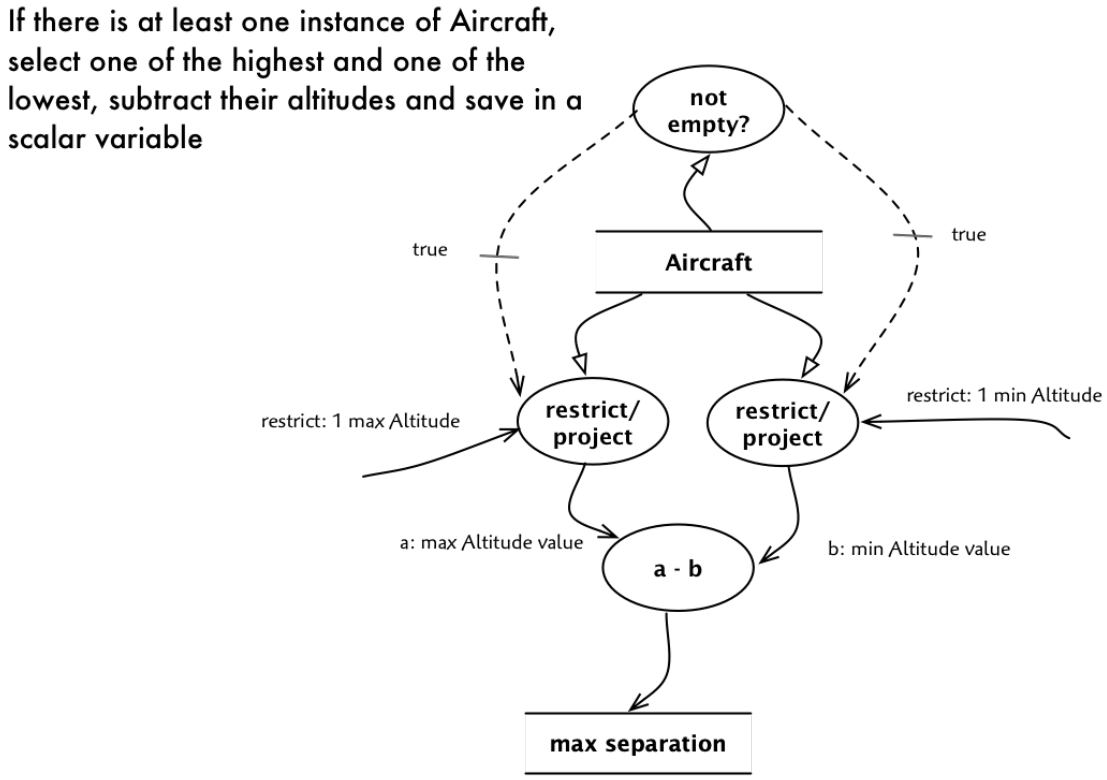
In the simplest case, an action may appear on a line by itself. A typical case would be an assignment:

```
current alitude = in.altitude
```

Multiple dependent actions on a single line

More than one data flow diagram bubble may appear in the same line:

```
Aircraft? max separation = Aircraft(1, +^Altitude).Altitude - Aircraft(1, -
^Altitude).Altitude
```



All of the actions in the data flow diagram above can be represented in a single line of Scrall.

As you can see on the diagram (and in the Scrall text) there is no data dependency between the two select/project actions so they could be executed in either order.

Multiple independent actions on a single line

If multiple actions are relatively terse and can execute independently of one another (with no data dependence among them) we can use the ; symbol to separate them on one line.

```
x = originX + length; y = originY + height
```

Note that the semi-colon not only separates the actions, it indicates that no data dependency exists among those actions.

An action spread across multiple lines

An action may require more than one line of text to express.

A long statement is split across lines using the back slash \ character. You don't need it if you split the line inside a parenthesized expression, an if-then-else structure, a signal action or any other action where the structure would be unambiguous.

Examples:

```
x = 1; y = 5  // two actions on the same line
a really long variable\  // Bad naming, but backslash needed
   name = some value
best cabin .= in service shafts/Cabin(1,
   ^-Estimated travel delay( Floor : /R4/Floor.Name, Calling dir: in.Dir )
) // action split across two lines using open parenthesis

Set destination( Shaft ID : Shaft, Floor name : Destination )
   => UI // action split using single back slash
```

A conditional group of single line actions

In the example below, the first line consists of single decision action. On a data flow diagram this would just be a bubble outputting mutually exclusive control flows. In Scrall, a variable is set and tested. An empty instance set variable evaluates as false, a table variable with zero rows/tuples evaluates as false, a false boolean variable will, of course, evaluate as false. A scalar type, on the other hand, must be explicitly tested to evaluate as false since its structure is opaque to our relational world.

Below, an expression is evaluated to decide which block of actions to execute. The second action block is enclosed in {} brackets and spread across multiple lines.

```
cabin in shaft .= /R2/Cabin
xfer .= cabin in shaft/R53/Transfer
```

```
call in progress .= xfer/R49/Floor Service( Direction: cabin in shaft.Travel di-
rection )


fwd dest .= cabin in shaft.Ping( dir: cabin in shaft.Travel direction )
fwd dest? destination aslev .= fwd dest :
    !call in progress? {
        // Search opposite the travel direction only if not servicing
        // a floor request in the current direction
        rev dest .= cabin in shaft.Ping( dir: Travel direction.opposite )
        rev dest? cabin in shaft.Travel direction.toggle
        destination aslev .= rev dest // may be empty
    }
```

This example is an excerpt from the **R53** assigner state model in the Elevator 3 Case Study. Its purpose is to search for a destination ahead of the Cabin in its current travel direction and, if none is found and there is no active floor call request (arrow button at floor), search behind the Cabin for a destination.

After attempting to select a forward destination, `fwd dest`, the value is checked and, if a destination was found it is saved as `destination aslev`. Otherwise, if no `call in porgress` was detected the bracketed action block, which looks in the opposite direction, is executed.

See the section on Conditional Execution under Control Flow for more about if-then-else processing.

Comments

Comments start anywhere in a line with the `//` characters. Anything after those characters will be treated as part of the comment up to the next newline. `/**/` style comments are NOT supported due to the usual problems these cause.

Accessing the class model

We access the class model to create or delete instances or to read or update attribute values. In the process, we may want to select a subset of instances (a restriction) of some class. Or we may want to navigate from one instance across one or more relationships to find a set of related instances. In more complex cases, we may want to select a subset of attributes (a projection). In even more complex cases we may want to grab attributes from multiple classes on a subset of instances connected together across relationships to construct tables. Ultimately we may may reduce all of this relational slicing down into the assignment a single scalar value.

In this manner, a small set of fundamental building block actions can be combined to perform powerful class model access operations. We'll define those building blocks in this section.

Selecting instances

To update or read data in the class model, it is first necessary to find the relevant instances. In fact, figuring out how to specify exactly the instances you want (or don't want) is the most challenging aspect of writing an activity.

Where to start?

The approach you take depends on where your search starts. There are a handful of potential starting points. For example, you might want to select a subset of instances within a known class. Or you might have a set of instances of some class already and want to traverse from those across one or more relationships to find related instances in the same or some other class. Finally, you might start from an assigner state machine and want to select one or more instances that it manages.

Let's explore how to proceed from each possible origin.

Starting from a class

We start with the name of some class and then apply criteria to select out a subset of instances belonging to that class.

The fundamental form for selection is:

```
<class-name>( <cardinality symbol>, <filter-criteria> )
```

So we have a class name followed by required parentheses containing an optional cardinality symbol and optional selection criteria. The comma is required only if both items are present.

Selection by cardinality

There are two cardinality symbols, `1` and `*` to select either one or all instances of a class. To select all instances, for example:

```
Airport(*)  // get all instances of the Airport class
```

You can assign the selected instance references to an instance set variable like so:

```
airports ..= Airport(*) // all instances assigned
```

But you don't have to assign the instance set to a variable. You might, for example, nest the selection inside some expression that expects a set of input instance references.

Since the "select all" case is quite common, it is assumed by default. So if you don't specify any cardinality, you get everything. So this also works:

```
airports ..= Airport // all instances assigned
```

The `1` cardinality symbol ensures that no more than one instance reference is produced. If no criteria is supplied, an arbitrary instance is selected or zero instances are returned if there are no instances in the class. Here we select an arbitrary instance:

```
some airport .= Airport(1)
```

If a class has no instances, the empty set is returned regardless of requested cardinality.

Using selection criteria

Criteria can be specified to filter the instance population of a class and return the corresponding instance references.

```
airports in my country ..= Airport( Country : my country )
```

Only those instances of **Airport** whose value for the **Country** attribute match the value of the **my country** scalar variable are selected.

In the context of a filter criteria expression, the `:` symbol reads as "is", "equals" or "matches" and the `,` symbol reads as "and". You can always substitute == and the `and` keyword, but these may add visual clutter, especially when multiple attributes are involved.

To improve readability `and`, `or`, and `not` are defined as keywords in Scrall. (You can also use `!` in place of `not`). These keywords are familiar and short and quick to type. So, you can name a variable `thisAndThat` or `this_and_that` or `this andthat`, but `this and that` is a really bad idea.

Supply an identifier to guarantee at most one instance selection:

```
origin airport .= Airport( Code : origin code )
```

where `Code` is considered unique among **Airport** instances, or

```
next point .= Point( x: xloc, y: yloc )
```

where each **Point** is understood to be unique by `x y location.`

In either case, zero or one instance will always be returned. Remember that the empty set can be returned if you select based on an identifier value.

Filter criteria can use the usual comparison operators if they are defined for the attributes involved in the comparison. For example, you can use these symbols: >, >=, <, <=, =, `!=`. Be careful, the data type of the compared values must support the desired comparison.

```
high aircraft ..= Aircraft( Altitude > high alt )
high and fast aircraft ..= Aircraft( Altitude > high alt and Airspeed > min fast
speed )
```

Min and max comparisons are represented by the ^+ and ^– unary operators. They can only be applied to attributes with data types that support ordering operations.

```
fastest aircraft ..= Aircraft( ^+Airspeed )
slowest aircraft ..= Aircraft ^–Airspeed )
allbut fastest aircraft ..= Aircraft( not ^+Airspeed )
```

Note the usage of the multiple instance reference assigner. Multiple aircraft may be flying at the same speed! If you only want one, specify 1 cardinality

More than one ordering comparison may be used and applied in the left to right order:

```
next tasks ..= Task( ^+Priority, ^–Time submitted )
```

The selection above is the set of Tasks with the highest priority and then, among those, the set with the earliest submission time. So the ordering is significant.

In the previous example, multiple instances may be selected. There could be many **Tasks** at the same **Priority**, submitted at the same time, for example. But it is possible to select an arbitrary instance among those selected:

```
next task .= Task(1, ^+Priority, ^-Time submitted )
```

Here, both cardinality and selection criteria are specified. In this case, zero or one instance will be selected, but the zero case would only occur when the cardinality of **Task** is also zero.

Returning at most one instance

For activities that return a value, such as a class method, you may want to return at most one instance.

Let's say, for example, the earlier example was in a class method that returns at most one task instance reference. In that case you could write this:

```
=>> Task(1, ^+Priority, ^-Time submitted )
```

The =>> symbol means "return". The calling action should be able to handle a returned empty set.

Methods in criteria

A method that returns a value may be used just like an attribute in a selection expression.

If a class method returns a scalar value of a type that supports ordering operations, the ^+ and ^- unary max min can be applied. Here's an example where the lowest duration returned by a method is used as the selection criteria.

```
cabin to call .= Cabin(1, ^-Estimated delay( Destination: Floor ) )
```

Here, each instance of Cabin invokes its **Estimated delay(Destination: Floor)** method to yield an orderable time duration value. A reference to the instance returning the least value is assigned to the instance set variable.

Starting from the local instance

Traversal from the local instance to one on the other side of an association can be accomplished by specifying a path using the hop / symbol (forward slash).

An instance of **Flot**, for example, can access its **Flot Specification** across an association like this:

```
my Spec .= /is specified by/Flot Specification
```

The relationship phrase, name or both can be used to unambiguously specify the navigation. For non-reflexive associations, the final path element should be the class name of the target instance set.

```
my Spec .= /R4/Flot Specification
```

```
my Spec .= /R4/is specified by/Flot Specification
```

In xUML, relationship names must be unique within a domain, but there is no such constraint on association phrases. So, if there is any ambiguity, you must supply the relationship name.

Multiple associations may be traversed in a single path.

```
my Bank .= /R4/Cabin/R2/Shaft/R1/Bank
```

Class names are not necessary, so you could type this:

```
my Bank .= /R4/R2/R1
```

For maximum clarity, always specify the destination class:

```
my Bank .= /R4/R2/R1/Bank
```

A smart editor with access to the class model can provide autofill assistance so that a relationship path can be entered in a consistent style with very few keystrokes.

The key principle is that every relationship path must be unambiguous.

Starting from an instance set

When you are not navigating from the local instance, you need to be explicit about the originating instance set.

```
bigdog breeds ..= Breed( Max height > big threshold )
bigdog owners ..= bigdog breeds/specifies/Dog/is owned by/Owner
```

In the above example that path expression begins with an instance set variable and then traverses from there across two associations to yield a set of related Owner instances.

If the originating instance set is empty, the empty set will be returned regardless of the remainder of the path.

Navigating a reflexive association

A reflexive association starts and ends on the same class. So navigation involves an originating instance set that yields another instance set on the same class.

An asymmetric reflexive association has a verb phrase on each side (Folder contains/is contained in). That is because the role differs depending on an instance's perspective with the multiplicity and conditionality on each side being potentially, but not necessarily, different. By contrast, a symmetric reflexive association (Territory/is adjacent to) has only one verb phrase since there is no distinction in perspective and, consequently, only one multiplicity and conditionality that is the same, by definition, from either perpsective.

Since an asymmetric reflexive association may be accessed in two directions to reach instances of the same class, it is essential to specify the relationship phrase in the desired direction. It is not enough to only specify the relationship name (Rnumber).

Assume, for example, that a number of aircraft are following one another in a holding pattern. Starting from some instance of Aircraft, to the one it follows, this action assigns that instance:

```
follow ac .= /R3/is following/Aircraft
```

Note that the class name at the end isn't really necessary to resolve the path. All you need is a relationship name and the phrase in the appropriate direction.

Also note that the `.=` assignment operator is used since at most one instance can be selected on R3 which is a 0..1:0..1 reflexive association.

If the reflexive association is symmetric, there is only one relationship phrase so there is no need to specify it. The relationship name is adequate. In the Risk game example we have the Territory *is adjacent to* (many to many) symmetric reflexive association. So you could write:

```
neighbors ..= /is adjacent to/
```

Now let's say that you want to find the nearest Aircraft in sequence at a higher altitude.

```
higher follow ac .= /OR3/is following/~/( its.Altitude > )
```

Note that we are hopping across an ordinal association as indicated by the "O" in the relationship name. (Instead of relying entirely on referential attributes to formalize the order, some attribute of type Ordinal, a numerical ordering type, sequences the instances). But you don't have to worry about this detail other than using the correct relationship name for our purposes here.

The repeated hop symbol `/~/`indicates continued traversal until either the criteria that follows is met or there are no more links to hop or a link back to the local instance is encountered. This last case prevents a cycle of references from being traversed endlessly.

The selection predicate will often compare an attribute of the referenced class with the same attribute of the local instance. To avoid confusion, the attribute of the referenced instance is prefaced with the keyword `its`, with that of the local instance optionally left implicit. In the above example, the criteria reads: "with a greater altitude than the local instance".

Now let's say that we want the next aircraft whose altitude is near the local instance's altitude within some higher/lower delta.

```
same level follow ac .= /OR3/is following/~/( (its.Altitude - Altitude).abs <=
delta )
```

You can see why the `its` keyword was important since we are subtracting the local instance's **Altitude** and we need to distinguish the two.

Note the use of the `abs` operation to get the absolute value of the difference. We use the dot notation instead of the more familiar abs() notation. It helps clarify that we are not performing a selection on some class named "abs".

Rather than closest, to find the leading aircraft that is the furthest number of hops away on OR3, use the repeated hop to end `/~|` symbol.

```
leading higher follow ac .= /OR3/is following/~|( its.Altitude > )
```

Here we traverse to the furthest rather than the nearest instance away that matches the filter criteria.

Alternatively, you can compare to the local attribute eliminating the need for the `its` keyword:

```
leading higher follow ac .= /OR3/is following/~|( Altitude <= )
```

Combined navigation and selection

You can hop around narrowing selection quite a bit in a single line. Consider the following action language that selects one or more elevator Shafts best suited to service a floor call.

```
in service shafts ..= /R1/Shaft( In service )
best cabin .= in service shafts/R2/Cabin(1,
^-Estimated travel delay( To floor : in.calling floor, Calling dir: in.Dir )
)
=>> best cabin/R2/Shaft
```

In the example above, we hop to all Shafts whose **In service** attribute is true and save them in the `in service shafts` instance set variable. Then we hop to all related **Cabins** sorting for the least returned value from the **Estimated travel delay** method invoked on each **Cabin** instance and assign one of them. The associated **Shaft** instance is then returned.

Navigating a generalization relationship

The path from a subclass to a superclass instance is straightforward. Let's take **R38** from the Elevator Case Study where a **Bank Level** is subclassed into a **Top**, **Middle** and **Bottom Level**. We also have **R29** between the superclass and the **Bank** class. Now consider an action within an instance of one of the subclasses, **Middle Level** let's say, where we want to find the related **Bank** instance, via the superclass.

```
myBank .= /R38/Bank Level/R29/Bank
```

Going the other direction, from a superclass instance downward, we need to ascertain the subclass instantiation. To do this we first perform a test and then continue navigation. In the case study example below, a method on the Bank Level superclass clears an up, down or up/down request based on its subclass.

```
R38? {
    .Top Bank Level:
        // If this is a top level and down is requested, clear the down request
        // set/unset are type specific operators defined on the Boolean type
        (in.dir == .down)? /R38/Top Bank Level.Calling down.unset
    .Bottom Bank Level:
        // If this is a bottom level and up is requested, clear the up request
        (in.dir == .up)? /R38/Bottom Bank Level.Calling up.unset
    .Middle Bank Level:
        in.dir? {
            // If this is a middle level and up is requested, clear up
            // If this is a middle level and down is requested, clear up
            .up: /R38/Middle Bank Level.Calling up.unset
            .down: /R38/Middle Bank Level.Calling down.unset
        }
}
```

When we test a relationship name, the result is an automatically generated enumeration value corresponding to one of the subclass names. So we can use a standard case statement to split out the possible navigation paths.

For more about the case action itself, see the relevant section under Control Flow.

Starting from an assigner state machine

An assigner state machine is not the lifecycle of a class instance. It is instead associated with a relationship for the purpose of managing competitive associations. So an activity in an assigner state cannot refer to a local instance since the concept has no relevance there. Note that the `me` keyword can still be used as the target of a signal in an assigner, but it refers to the state machine instance and not any class instance.

It is convenient, however, to have an implicit origin from which to navigate to class instances.

Single assigner

Consider the association R1 between Customer and Clerk where a single assigner establishes a "Clerk is serving Customer" 0..1:0..1 association. There must be an association class (because of the multiplicity) which we'll call Service that formalizes R1.

Now we have an assigner state model on R3. There is only one state machine instance for the entire association.

But you can't do this:

```
availableClerk .= /R1/Clerk(1, !Serving customer )  // ILLEGAL
```

This is illegal because no initial point for the navigation has been specified. Instead, you must do something like this:

```
availableClerk .= Clerk(1, !Serving customer ) // LEGAL
```

Here there is no relationship navigation at all and we are just selecting an arbitrary instance of Clerk whose **Serving customer** attribute is false.

This is also okay:

```
busyClerk .= Clerk(1, Serving customer )
[busyClerk] clerksCustomer .= busyClerk/R1/Customer
```

If an instance of a Clerk serving a customer is found, that Clerk's current Customer can be found by traversing R1 *from* the selected Clerk instance. So you can traverse associations in an assigner state model, but you must explicitly supply some instance reference as the starting point.

Multiple assigner

A single assigner manages competition on an association for all instances on each side of that association. In the Customer-Clerk example, any Customer instance might be associated with any Clerk instance on R1.

A multiple assigner manages competition among a subset of instances on each side of the association. Let's say that we add a Department class and relate it to both Customers via R2 and Clerks via R3 to reflect the fact that Customers and Clerks are in a particular Department. Now we want a separate assigner state machine instance per Department to manage only those Clerks and Customers in that Department.

So a multiple assigner designates some partitioning class, in this case, Department, and then supplies a separate state machine instance per partitioning class instance. Thus, we can use instances of Department as starting points for relationship navigation. So, you could do this:

```
availableClerks ..= /R3/Clerk( !Serving customer ) // LEGAL
```

Navigation implicitly begins from the associated Department instance across R3 to select all of that Department's Clerks who are not busy.

Reading an attribute

The dot notation is used to read or write an attribute.

If you are referencing the attribute of an instance in an instance set (variable or expression), you need to use the dot notation to access any of its attributes. For example:

```
cabin height = /R1/Cabin.Height
```

Attributes of the local instance need not be qualified. Assuming we have an instance of Aircraft and a single instance of some other Aircraft in a `target` variable we could write:

```
closing speed = target.Air speed – Air speed
```

The `Speed` element is understood to represent the local Aircraft instance as distinct from the target Aircraft's speed.

This means that no local variables may be named to match any local attribute. So, if you have a class called Aircraft with attributes **Tail number**, **Altitude**, **Airspeed** and **Heading**, you cannot use any of those names for local variables.

Assignment to a scalar variable

A scalar variable can be assigned an attribute value from an instance reference set that contains exactly one instance. Let's say that an instance of Shaft wants to save the height of its Cabin.

```
cabin height = /R1/Cabin.Height
```

If the RHS yields zero or many instances, there will be a runtime error. The above example is taken from the Elevator Case Study where there is an unconditional to-one association and it is assumed that all instances have been created at initialization. Therefore, there is no need to check in this case.

If, on the other hand, it is possible for the RHS to yield the empty set, you should verify that this is not the case before making the scalar variable assignment.

Here, an Aircraft instance has a 1:0..1 association with an assigned runway. So we must verify that the `my runway` variable is not empty before performing the scalar assignment.

```
my runway .= /taxi to/Runway
my runway? takeoff heading = my runway.Heading : norunway -> me
```

Or you could do this:

```
found runway = /taxi to/Runway
// implicit conversion to boolean
(found runway)? takeoff heading = my runway.Heading
(!found runway)? norunway -> me
```

Note the use of the = assignment operator instead of `.=`. Since the empty set is false and one or more instances true, we can perform an implicit conversion to a boolean scalar variable. There isn't much point in

doing this here since you could just evaluate the instance set as we did in the previous example. But, in a more complex activity, you may find a need for a boolean variable assigned in this manner.

Reading multiple attributes

You can assign multiple attribute values if a single instance is selected to the same number of scalar values. Each scalar value must be typed appropriately.

```
calling fnum, calling height = Floor( Name: in.calling floor ).(Number, Height)
```

The selection must find exactly one instance or a runtime error will result.

A safer approach might be:

```
calling floor .= Floor( Name: in.calling floor )

calling floor? calling fnum, calling height = calling floor.(Number, Height) : No
calling floor -> me
```

If, however, you are certain that one instance will be selected, say based on the requirement for an initial population, or by filtering input to your domain via a domain operation, for example, to ensure that the appropriate instances exist, then you may be satisfied to simply fail by not finding the instance and enjoy the simplicity in the first example.

Writing an attribute

The following action assigns a value to an instance's attribute

```
running task.Time to completion = remaining duration
```

Assuming that the local class is **Task** with a numeric **Priority** attribute having a type that supports the ++ operator, the following is legal:

```
++Priority  // increment the priority
```

Assuming **Commanded heading** is an attribute of the local class **Aircraft** you could do this:

```
Commanded heading = in.go this way  //incoming parameter assigned to local at-
tribute
```

Writing an attribute for multiple instances of the same class

You can set an attribute of multiple instances of the same class:

```
engines ..= /powered by/Engine
engines.Power setting = desired setting
```

Each instance of `engines` will assign the same value to its **Power setting** attribute. In the case of the empty set, there is no error and no values will be assigned.

Reading and writing multiple attributes of the same class

For the **Rectangle** class with attributes **X**, **Y**, **Length** and **Width** you can do this:

```
l, w = /Rectangle.(Length, Width)
```

The left right ordering indicated on the RHS is preserved on the LHS.

Or if you define a type called **Rect Size** with components **Length** and **Width** you could do this:

```
rect = /R6/Rectangle(ID: some rectID)
rect? {
    rsize::Rect Size = rect.(Length, Width)
    l,w = rs.(Length, Width)  //extract components
}
```

For this to work, the type **Rect Size** must provide an initialization operation that takes two values whose types that match the **Length**/**Width** types.

Looking at the class model, let's say that **Rectangle.ID** is an identifier. This selection then can yield only zero or one instance. We assign the scalar variables only if the `rect` instance set variable is non-empty and therefore holds a single instance.

Creation and deletion actions

When creating or deleting instances we need to consider how the action is synchronized and how it impacts the integrity of model relationships.

Synchronization

Creation can be triggered asynchronously by sending a creation event specifying the class of the instance to be created. Asynchronous deletion is achieved by addressing an ordinary event to the instance to be deleted where that event triggers a transition to a deletion state.

Scrall provides syntax both for signaling ordinary events and for sending creation events.

Synchronous creation and deletion are performed remotely by an action in some location other than that of the target instance's state model.

Scrall provides syntax for both synchronous creation and deletion actions.

Model integrity

The class model specifies various constraints on relationships that must be respected by the state and activity models to ensure model integrity while the models are executing. It is the responsibility of the modeler to ensure that all constraints are respected in the model activities. For example, in a 1:1 association when an instance on one side of the association is deleted, the instance on the opposite side should be likewise deleted or linked to a newly created instance. It is also the modeler's responsibility to ensure that all attributes have meaningful values at all times. There is no provision for assigning a null (lack of a value) to an attribute.

A referential attribute, in particular, must always hold a legal value. This means that you cannot create an instance with referential attributes if the referenced instance does not yet exist. It also means that you cannot delete an instance currently referenced by one or more referential attributes.

Say, for example, that you have a 1:1 association between classes **A** and **B** where **A** holds a reference to **B**. You must delete the **A** instance first and then the **B** instance immediately after and not in the reverse order. This ensures that you never orphan any referential attributes.

Creating an instance synchronously (with an action)

The create action creates an instance immediately and returns a reference to the newly created instance. It is synchronous because the creation begins and ends within a single action inside of an activity.

To create a new instance, use the new ∗ symbol with initial values supplied:

```
ac .= *Aircraft( Tail:t, Altitude:a, Heading:h, Location:l, Airspeed:s)
```

There are no "not applicable" or "null values" permitted in xUML. So you either create a complete instance or you don't. This is a necessary consequence of the underlying set theory. It is also a consequence of not assuming anything about the memory implementation.

If an attribute is omitted from the creation action, the attribute's specified initialization value indicated on the class model will be used. Failing that, a type specific initial value generator operation will be invoked. At least one of these initialization options must be available or a static error will result.

If a class references across an association, all referential attributes must be initialized. Rather than specify the referential attributes directly, each association and associated instance reference must be specified.

The following action is executed by an instance of **Owner** and provides itself as the necessary instance reference:

```
v .= *Owned Vehicle( License number:l ) &R1 me
```

Since you are relating to self, you can shorten it to:

```
v .= &R1 *Owned Vehicle( License number: l )
```

And if you don't need the variable, you can just do this:

```
&R1 *Owned Vehicle( License number: l )
```

If we create an instance that holds multiple references, each must be provided in a comma separated list. Here, a **Shaft** class references both a **Cabin** and a **Bank** class.

```
*Shaft &R4 cab, &R6 mybank
```

All non-referential attributes are presumably initialized by default values or type initialization values so the parentheses would be empty and may then be omitted.

The comma is necessary since it separates distinct <association>, <instance> pairs. So you read the statement as "Create an instance of **Shaft** and associate it with the **cab** instance on **R4** and also associate the **Shaft** instance on **R6** with the **mybank** instance. Without the comma, it would appear that you are linking Shaft to Cabin and then Cabin to Bank which is not the case.

```
*Shaft &R4 *Cabin, &R6 mybank
```

Here, a new instance of Cabin is created and provided as an instance reference.

Creating an instance asynchronously (with an event)

With this approach a creation signal is sent and, upon dispatch, the model execution domain creates the new instance and places it in an initial state as designated on the target class's state model. Any parameter values supplied with the creation event are made available to the new instance's creation activity.

If the new instance's class holds any references, a target instance reference must be supplied with each along with the corresponding association names.

This style of creation is called *asynchronous* since the sending instance can continue about its business, advancing through states, etc. without necessarily waiting on any feedback regarding the creation status.

To perform asynchronous creation, send a signal to a class name (not an instance set variable) and precede the class name with the * new instance operator.

```
New folder( Parent: parent id ) -> *Folder
```

The event target is a class name which establishes that this is a creation rather than a normal instance directed event.

All data necessary to create a complete instance of **Folder** must be available to the creation state. Though, not all data need be specified as parameters on the creation event if that data is available elsewhere, such as in attribute initialization values specified on the class model.

The **Create folder** event will be defined as a creation event on the **Folder** state model. But the * is helpful to clarify to the model reviewer that a new instance is created. It also encourages the model developer to be explicit about their intention. Consider a non-creation event sent to all instances of **Folder**:

```
Close -> Folder
```

In the above example a **Close** event will be sent to each instance of **Folder**. The fact that all instances are selected, you could have done this instead with the same result:

```
Close -> Folder(*)
```

Now let's say that we want to asynchronously create an instance of a class that holds more than one reference.

The syntax is similar to that for synchronous creation. You must specify a comma separated list of association name, instance reference pairs.

Recasting the Shaft example as asynchronous creation, it could look like this:

```
Create -> *Shaft() &R4 *Cabin(), &R6 mybank
```

Deleting instances

Like creation, deletion can be triggered by a signal or an action. Deletion is easier to specify than creation since you don't need to worry about initializing any attributes. You just need to be careful not to invalidate any references from other classes.

Synchronous deletion (with an action)

To delete an instance synchronously, use the delete `!*` command. Symbolically, you can remember it as "un-create". Let's say you have a set of Aircraft instance references:

```
!* ac
```

In the example above, all instances in the instance set variable `ac` are immediately deleted. It is best to use this approach with classes that do not have state models. If the class does have a state model, the modeler should ensure proper synchronization so that deletion occurs when no critical activities are being executed by the instances to be deleted. Furthermore, an error will occur if this action would invalidate any references from other classes.

As with most operations on an instance set variable, there is no special consequence or error in the case of an empty set.

Asynchronous deletion (with an event)

Deleting with an event is simply a matter of sending a normal event to the instance you want to delete. Presumably that event will cause a state transition to a deletion state in the target class. Since there's nothing special about a deletion event, there is no special Scrall syntax.

Creating an instance of an association class

Every association class holds at least two references, one on each side of the association it formalizes. Just like any class, an association class may hold references on other associations.

Consider the **Engagement** class which represents an interaction between a **Missile** and a **Target**. To create an instance of **Engagement** asynchronously you could do this:

```
Target acquired -> *Engagement(Closing speed: cspeed) &R1 the missle, the target
```

If it turns out that an **Engagement** also holds a reference to an instance of **Attack Profile** on the 0..*:1 R8 association, you will need to supply that instance and association as well:

```
Target acquired -> *Engagement(Closing speed: cspeed) &R1 the missle, the target
&R8 my profile
```

Creating an instance of a generalization relationship

Again, our prime concern is that all attributes, including referential attributes are initialized correctly. Since a subclass holds a reference to each of its superclasses, it follows that a subclass cannot be created without specifying its superclass instance. This can all be done in a single action, or be split apart with the superclasses being created first. In either case, it is important to ensure the integrity of all instances in a generalization, with no childless superclasses lying around.

```
*Off Duty ATC &R1 *Air Traffic Controller( ID:newid, Name:newname, Date of
birth:dob )
```

Subclass migration

Migration reclassifies some subclass instance of a generalization to an instance of some other subclass in the same generalization. Migration (also known as reclassification) both creates and deletes one subclass instance while maintaining a relationship to the same superclass instance. The source subclass instance is deleted while a new one is created in the target subclass. You can only migrate within the same generalization.

All attributes, including referential, in the new subclass must be assigned values as part of the migration action.

To specify migration, use the >> migrate symbol as shown:

```
atcOn >> Off Duty Controller( Time logged off : Date.Now hms )
```

In this example, all instances in `atcOn` migrate to the Off Duty Controller class with the **Time logged off** attribute set to the current time. (If you only want to migrate one instance, ensure that `atcOn` is assigned only one instance.

If no instance set variable is specified, the local instance is presumed.

```
>>On Duty Controller(Time logged in: Date.Now hms) &R3 my station
```

Here the local instance is migrated with both the attribute set and the new subclass instance is linked to the **my station** instance.

If no attributes need to be set, the simplest form is this:

```
cv .= >>Closed Valve()
```

In this case, the local instance of Open Valve migrates to Closed Valve. No new attributes, referential or otherwise, need to be set in the new subclass.

Furthermore, the action returns an instance set for each newly created subclass. In this case there is only one.

Multidirectional subclass

If the instance you want to migrate belongs to a generalization with a superclass that participates in more than one generalization, you may need to perform multiple migration actions.

Let's say that the superclass is **Valve** subclassed on R1 as **Open** and **Closed** and on **R2** as **Operational** and **Stuck**. And forget, for the moment, that this is probably a bad model. Now you attempt to open a valve which is **Closed** and **Stuck**. You succeed, and now you must perform two distinct migrations.

```
>>Open()
>>Operational()
```

Operations on instance set variables

Use the cardinality (how many are there) symbol ?? to obtain the cardinality of an instance set variable.

```
qty priority tasks = ??next tasks
```

The result is a positive integer.

Set operations

Basic set operations such as union, intersection, subtraction (set complement) are supported on instance sets of the same class.

Here the variables r, a and b are instance sets of the same class:

```
r ..= a ^ b // intersect
r ..= a + b // union
r ..= a - b // subtract
r ..= a % b // symmetric difference
quantityOfR = ??r // total number of instance references
```

Super and subset operations yield a boolean result:

```
( c < d) {
   // is c a proper subset of d?
   // if so, take action here
}
```

The symbols are <, <_, > and _> for proper subset, subset, proper superset and superset. The set operators might look a bit strange, especially since some of them double as scalar value comparison operators, but they are easy to type and a modern text editor can replace these symbols with the corresponding math set characters ⊂, ⊆, ⊇, ⊃ .

Here are some more examples:

```
// Grab some aircraft instance references
fastest aircraft ..= Aircraft( Airspeed > min speed )
highest aircraft ..= Aircraft( Altitude > floor alt )


// Now let's do some set operations on them
slow aircraft ..= Aircraft - fastest aircraft // set complement from class popula-
tion
high and fast aircraft ..= fastest aircraft ^ highest aircraft // intersection
fastest low aircraft ..= fastest aircraft - highest aircraft  // set complement
my aircraft .= /Aircraft( Tail number: my tail number )
```

```
my aircraft and slowest aircraft ..= my aircraft + slow aircraft  // single in-
stance in union

qty of slow aircraft = ??slow aircraft
```

Operations on tables

A table can be constructed from a class or from a table definition comprising a set of attribute type pairs.

Table operations are best defined in C.J. Date's writing on relational theory. The two key books to read are An Introduction to Database Systems (latest edition is eighth as of this writing) and Relational Theory for Computer Professionals. Here we'll assume that you've read these or are somewhat familiar with the operations with only a very light treatment and a few simple examples.

Much of what you want to accomplish can be performed with instance sets, so you rarely need to explicitly construct tables. If you are not a relational algebra wizard, don't worry. But Executable UML is rooted in relational theory and power modelers can sometimes specify complex computational activities more easily with table operations, as opposed to resorting to needlessly platform specific data structures.

Creating a table from a class

Any selection on a class that projects over a subset of that class's attributes will generate a table value that you may assign to a table variable using the #= assignment operator.

The selection format for table values is:

```
<class or table>( <selection> ).( <projection> )
```

Here are some examples:

```
t1 #= Aircraft.(Altitude, Speed)  // creates a table

t2 #= Aircraft(1, +^Altitude).(Altitude, Speed) // One row max table with max Al-
titude

t3 #= Aircraft(Tail Number: in.ID ) // One row max table with all attributes of
Aircraft

rowqty = ??t3  // Number of rows in t1
```

You need not perform an assignment to create a table. You might nest a table construction or return a table as an output parameter. If you select all attributes, you will need to clarify that you want to generate a table rather than an instance set. To do this, preface your expression with the # symbol. For example:

```
=>> Aircraft // returns all Aicraft instance references


=>> #Aircraft // returns a table of all Aicraft data
```

Here is the general table construction format:

```
<class name>(<instance selection>).(<attribute selection>)
```

Here are some example expressions that construct a table:

```
Aircraft(*).(Heading, Airspeed)  // Two column table
Aircraft.(Heading, Airspeed) // Same table (all implied)
Aircraft.(-Tail number) // Table has all attributes except the Tail number
Aircraft.(-Heading, -Airspeed) // Table has all attributes except Heading and Air-
speed
#Aircraft(1) // One arbitrary row, all attributes implied
#Aircraft(1).(*) // Same table
#Aircraft(*).(*I1) // Instance reference table on ID1
#Aircraft(*).(*I2) // Instance reference table on ID2
#Aircraft(1).(-*I1) // All attrs except I1 of some instance
Aircraft(Tailnumber : in.ID).Airspeed // Airspeed table
Aircraft(Tailnumber : in.ID) // Yields zero or one instance reference
x .= Aircraft(Tailnumber : in.ID) // Assigns instance references
t #= Aircraft(Tailnumber : in.ID).Airspeed // Zero or one airspeed in a table
t2 #= Aircraft(Tailnumber: in.ID) // Zero or one row table
```

So we can see that the – symbol excludes an attribute from a projection. Also the symbol `*In` refers to all attributes of identifier n. And `-*In` gives you all attributes except those in identifier n. And the * symbols selects all attributes when it appears in a projection

You can also produce some weird tables that are set-theory correct and often useful for true/false decisions:

```
Aircraft(*).(0) // No attributes, zero or one row
// True if one row, false if zero rows

Aircraft(0).(0) // No attributes, zero rows
// False

Aircraft(0).(*) // All attributes, zero rows
// False
```

To get a sense of how these might be useful, consider some examples:

```
speeds above ceiling #= Aircraft(Airspeed > ceiling).(0)
// returns table with zero attributes and zero or one row
```

```
speeds above ceiling?
    Speeds above ceiling -> : No speeds above ceiling -> me
```

You can never have duplicate rows in a table. So if six different speeds are found by the selection criteria, but zero attributes are returned we get six duplicate rows which yields a single row. If none are selected, then you get zero rows. Thus only two possible cardinalities can result when zero attributes are specified in the attribute projection expression.

Unless the above is part of more intricate table actions, there is no need to resort to table operations. You could get the same result with the less obscure:

```
Aircraft(Airspeed > ceiling)?
    Speeds above ceiling -> : No speeds above ceiling -> me
```

(The abbreviation of omitting the target instance in the "then" part when it matches the target in the "else" part is described in the conditional execution section).

Creating a table with a definition

Alternatively, you can create a table with a set of attribute-type pairs:

```
t4 #= #[ x: xloc, y: yloc ]  // Table with one row
t5 #= #[ x:Rational, y:Rational ] // Table with zero rows
```

In this example `t4` is assigned a table with two columns and one row. The variables xloc and yloc implicitly type the header elements. Table variable `t5` is assigned a table with two attributes each of type **Rational**. Since no values are supplied, the type designates must be explicit.

To create a table with multiple rows, use the `;` symbol to separate the rows:

```
t6 #= #[ x: xloc1, y:yloc1; x: xloc2, y: yloc2 ]
```

To initialize the relational true and false equivalents (see C.J. Date's writings on tabledum and tabledee) do this:

```
tabledum #= false // zero attributes, zero rows
tabledum2 #= #[] // same result as above
tabledee #= true // zero attributes, one row
tabledee2 #= #[1] // same result as previous
```

Two methods are provided to keep the syntax consistent. To keep your actions consistent and readable, choose the one method that suits you and stick to it.

Converting a table into a class

You cannot convert a table directly into a class since there are usually referential attributes involved and it would be quite easy to violate referential integrity by populating a table with invalid references.

But you can convert a table into a set of instance references and then assign those to an instance set variable like so:

```
tlow #= Aircraft(Altitude < floor)
tids #= tlow.(Tail number)
low flying aircraft::Aircraft ..= tids
```

We need to specify a table whose header is a superset of at least one of the target class's identifiers. In this case it just means that the table **tids** must have **Tail number** in its header.

Note that we need the `::` type definition symbol to ensure that the instance set variable is defined on the correct class.

Set operations

The following operations are defined only when each operand shares the same exact table header.

```
t #= q ^ r // intersect
t #= q + r // union
t #= q - r // subtract (subset of q without r)
t #= q * r // multiply (cross product)
t #= q % r // symmetric difference (union minus intersection)
quantityOfR = ??t // total number of rows
```

Set comparisons

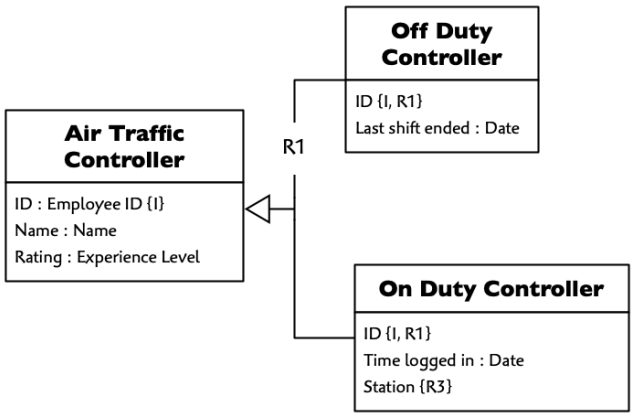
These work just like the instance set comparison operators. In all cases the comparison operands must have the same table header.

```
<, <_, >, _> operators are proper subset, subset, proper superset, superset
if (t1 < t2) {...} // t1 is a proper subset of t2
```

Join

Given two tables t1 and t2 with zero or more attributes in common (same name, same type), the join operation yields a table where each row in t1 is matched with every compatible row in t2. For now, let's ignore the case where there are no attributes in common. A row is compatible if values match for all of the attributes shared by the two tables.

You can imagine the join symbol ## as depicting two tables joined together.




```
t #= Air Traffic Controller ## On Duty Controller
```

As a result of the join, `t` will be assigned a table with columns **ID**, **Name**, **Rating**, **Time logged in** and **Station**. Since the attribute **ID** has the same name and type in each class, each row represents all the common and specific data for an on duty air traffic controller.

If we only wanted a table of log in times organized by ID later than some specific time, we could do this:

```
late times #= (Air Traffic Controller ## On Duty Controller( Time logged in > some
time )).(ID, Time logged in)
```

By surrounding the join result in parenthesis we can add a projection expression after a dot just as we would with any table.

If you join two tables with no attributes in common, each row in the first table will be compatible with every row in the second table. Consequently, you end up with the cartesian product.

Let's say we have a class called Assembly Station with attributes ID and Location and another class called Manufacturing Task with attributes Name and Duration.

```
max assignments = ?(Assembly Station ## Manufacturing Task)  // Total possible
task-station assignments
```

Since there are no attributes in common, we end up with a table having attributes ID, Location, Name and Duration where each instance of Assembly Station is matched with every instance of Manufacturing Task to yield a table of all combinations and then we take the cardinality of that resulting table to get **max assignments**. Note that the cartesian product table was an intermediate result that was not assigned to any table variable.

Rename

It is common practice to name a referential attribute and its reference target differently. To join two such class tables together we need to rename one of the attributes so that the names match up. We use the rename >> operator for this purpose.

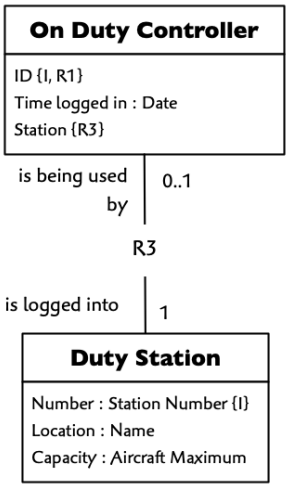
For example, let's say we have an **Arrival** class with attributes **Aircraft** and **Airport** representing all scheduled arrivals. Now we want to find all **Aircraft** not scheduled to arrive anywhere. We could do this:

```
non_arriving_ids #= Aircraft.Tail number - Arrival[Aircraft >> Tail number]
```

The complete table access format including rename is <table>[fromName >> toName, …](<selection>).(<projection>). You can omit any or all braced or parenthesized components if there is nothing to specify. The dot, however, is required when any projection is specified.

We use the >> rename operator on Arrival.Aircraft to yield a table whose header matches the projection on Aircraft.Tail number. Then we subtract the set of Arrival tail numbers from the set of all Aircraft tail numbers to yield all Tail numbers not scheduled to arrive.

Here's one such example:



To create a table of ATC ID's matched with station locations, we could do this:

```
atc locations #= (Duty Station ## On Duty Controller[Number >> Station].(ID, Loca-
tion)
```

The common attribute in the join has a different name in each class. Either Station should be renamed as Number or vice versa. In the example above the Number attribute is renamed. Then, we project over ID and Location.

Extend

The extend operation adds an attribute to a table and populates it with values according to some formula.

The rename >> symbol becomes an extend operator if it has nothing on the left inside of a projection expression. Let's say we have a table of Celsius temperatures and we want to add a column of equivalent Fahrenheit values.

```
ftemps #= ctemps[>>Fahrenheit(Celsius.toFdeg)].(Fahrenheit)
```

We start with the ctemps table which as only one column named Celsius. It is extended by adding a Fahrenheit column who's value will be the Celsius temperature with the toFdeg conversion invoked. We assume that the temperature type provides this conversion operation.

Aggregation

Aggregation is some operation that can be applied to a set of scalar values of the same type yielding a single scalar value result. For example, the sum operation adds up all of the scalar values in a column. Other common possibilities are min, max and avg (average). Aggregation is not a relational operation since it yields a scalar value rather than a relation.

So it is critical that any given aggregation operation be supported by the type it operates on. It makes no sense, for example, to take the average of a set of values of type **Name** since that type does not support addition. Many aggregation operations require at least one input value so, to avoid a run-time error, you must check for the empty set.

Scrall does not provide a predefined list of aggregation operations. These must be defined (or supplied automatically) as part of the type definition system since that is where supported operations are managed. Scrall merely provides a syntax for invoking an aggregation function on an attribute or table column.

Assuming an avg aggregation operation has been defined and is supported on the Altitude type, as one would expect since Altitude would be based on the Rational type, this works:

```
avg altitude = Aircraft(*).(Altitude).avg // same result
```

When the dot notation is a suffix to a projection expression on one or more attributes it must designate an aggregation.

Since there is only one attribute, it can be shortened to get the same result since all attributes and all instances are implied:

```
avg altitude = Aircraft.Altitude.avg
```

If there are zero instances of Aircraft, the assignment will fail with a run-time error since avg would need to divide by zero and min and max are not defined for empty sets. (You CAN define max so that it approximates negative infinity as the minimum system defined value, but it's unclear that this would be useful). So, if you are not certain to obtain at least one instance with an aggregation operation that isn't defined on the empty set, check first as shown:

```
Aircraft? avg altitude = Aircraft.Altitude.avg
```

Above, we create a boolean condition as indicated by the ? symbol on the right of an implicit "select all" on Aircraft. If at least one is found, the average is computed. (Technically we need only do an Aircraft(1) select, but we'll assume that the code generation is smart enough to figure this out since the selection is not assigned and merely tested).

To produce a table of heading and altitude averages, do this:

```
Aircraft? ha avgs #= Aircraft.(Heading, Altitude).avg
```

To get the maximum altitude of all aircraft faster than a certain speed:

```
Aircraft? highfast alt = Aircraft( Airspeed > s ).Altitude.max
```

Since there is a max symbol defined, you can do this which produces the same result:

```
Aircraft? highfast alt = Aircraft(1, Airspeed > s, +^Altitude).Altitude
```

You can specify an operation on the projected attribute as input into the aggregation like this:

```
totalCapacity = Duty Station.(Capacity * scale).sum
```

Rank

The rank operation adds a column with an ascending or descending order based on some value.

```
rankedAltitudes #= Aircraft[>>Lrank:^-Altitude]
```

The resulting table includes all of the Aircraft attributes with an added column named **Lrank** filled with an integer rank value for each **Altitude** value. The rank values start at 1 for the lowest altitude and increasing for higher altitudes.

The `:` and `^-` symbols have a different meaning in the context of a projection expression than they do in the restriction expression. Specifically, `:` serves as a delimiter and `^-` and `^+` mean descending and ascending, respectively.

The operation can be useful when you want to find the n highest or lowest aircraft. For example:

```
ranked by altitude #= Aircraft[>>Hrank:^+Altitude]
n highest rows #= ranked by altitude(Hrank < n)
n highest aircraft #= n highest rows.(-Hrank)
```

First we add the **Hrank** column with altitudes ranked with 1 representing the highest altitude. Then we grab all Aircraft with a ranking less than **n**. Finally, we drop the **Hrank** column.

We can dispense with the intermediate variables and use a single expression to produce the same result:

```
N highest aircraft #= Aircraft[>>Hrank:^+Altitude](Hrank < n).(-Hrank)
```

In fact, we could go further and convert the table back into an instance set value like so:

```
High fliers::Aircraft ..= N highest aircraft.Tail number
```

This eliminates the need for dropping the Hrank column, so the expression yielding an instance set becomes:

```
High fliers::Aircraft ..= Aircraft[>>Hrank:^+Altitude](Hrank < n).Tail number
```

Image

The image operation is best explained in C.J. Date's book: Relational Theory for Computer Professionals, Chapter 5. It eliminates the need for a summarize operation.

Borrowing with a few stylistic adjustments, here are C.J. Date's case study tables for reference:

Suppliers

Number	Name	Status	City
S1	Cartman	20	Denver
S2	Phillip	10	Toronto
S3	Terrance	30	Toronto
S4	Kenny	20	Denver
S5	Kyle	30	Cupertino

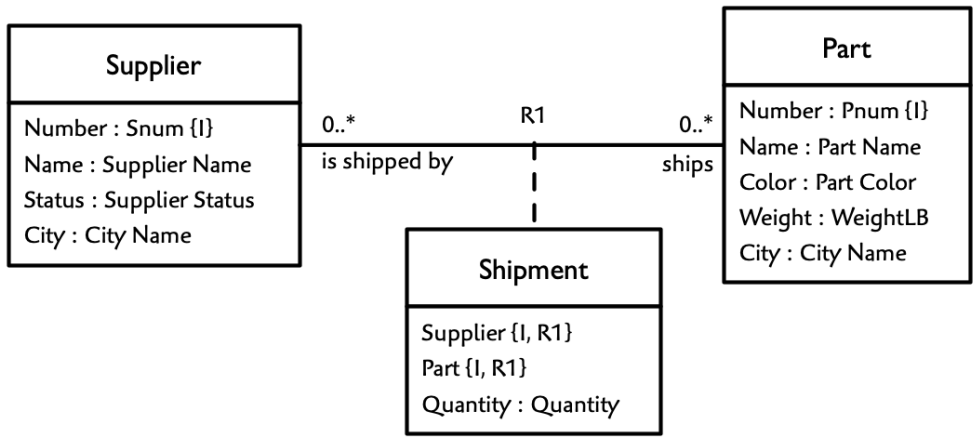
Parts

Number	Name	Color	City
P1	Cover	Rose	Denver
P2	Slide	Space gray	Toronto
P3	Button	Slate	Montreal
P4	Button	Rose	Denver
P5	Plate	Slate	Toronto
P6	Charger	Rose	Denver

Shipments

Supplier	Part	Quantity
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

And here is the corresponding class diagram:



The image of Supplier number S4 in Shipment would be:

Image of S4

Part	Quantity
P2	200
P4	300
P5	400

Which is equivalent to:

```
imageS4 #= Shipment(Supplier : s4).(-Supplier)
```

To see how the image operation is useful, we use it to find all Suppliers that ship all Parts using the `!!` image operation:

```
ships all the parts #=
    Supplier[Number >> Supplier](!!Shipment.Part number : Part.Part number )
```

So first we rename the Supplier table so that its Number attribute matches the Shipment.Supplier attribute.

Then we take the image of each Supplier row in the Shipment table projecting only on Part number. This gives us a one column table of distinct shipped Part numbers. We match this against the set of all Parts which is just the Part table projected on Part number.

Input values

Values can be passed into an activity with an input parameter designated with the `in.` prefix. On a data flow diagram an input parameter appears as a data flow with no source.

```
next destination = in.Floor
```

If the above example is in a state activity, the **Floor** parameter packaged with the event that triggered the incoming transition is assigned to the scalar variable on the LHS.

Variable types that can be passed as input parameters depend on the type of activity as follows:

State activities: Table and scalar variables only (instance references are not guaranteed to be valid during the transmission of a signal).

All non-state activities: All variable categories

Since a class method is invoked either by a domain operation or a state activity, all instance references and model data should remain valid assuming the actions are sequenced as necessary.

Signatures

In all activity types, parameter names are specified using formal : actual pairing and never with implicit ordering. Consider an event specification defined as follows:

Set new dest(Destination : Name, Direction : Vdir)

Two parameters, **Destination** and **Direction**, with data types **Name** and **Vdir** are required in the **Set new dest** event specification. To generate a corresponding event, supply values of the appropriate types paired up with the parameter names in any order.

```
Set new dest( Direction : in.Dir, Destination : in.Dest floor ) -> me
```

No need for name doubling

Note how the parameter names and supplied variable names closely match? As a convenience, and to avoid name doubling clutter, you can drop the formal parameter if the name of the actual parameter matches the formal parameter name. Let's change the event specification to this:

Set new dest(Dest floor : Name, Dir : Vdir)

Now the formal parameter names match the actual parameter names the action can be shortened like this:

```
Set new dest( in.Dir, in.Dest floor ) -> me  // less clutter
```

The action language is also easier to read and less trouble to write. If the actual parameter and formal names don't match, a static error results.

Declaring output values

A method or operation can output a scalar value, a table or a set of instances. The declaration of a method or operation signature is not performed with action language. Instead, you show it in the lower compartment of whatever model element provides the method or operation (class, external entity, domain operation) and also in a comment at the top of your activity.

We use the term "output" since it more accurately reflects our data flow thinking (an activity has inputs and outputs) than the programming concept of a "return value".

Output a scalar value

On the class diagram, use the UML method signature style to define return values:

Computer area(length : Distance) : Area

Output a table

Name the table and preface name with **#** followed by a heading consisting of attribute name : data type pairs in parentheses:

Stimulus Occurrence.Match pattern(ext system, stimulus, param values) : #(Name : Pattern Name)

This signature outputs a table of **Name** attributes typed **Pattern Name**

Output a set of instances

Name the class and preface name with **.** or **..**

(This avoids collision with a class name and a data type name or a table name)

For example:

Behavior.Next task() : .Task

Outputs zero or one **Task** class instance references.

Or:

Behavior.All tasks(): ..Task

Outputs a set of zero, one or many **Task** instance references.

Output parameters

Zero or multiple scalar values, table values and instance references may be output by synchronously invoked activities such as domain operations, synchronous external entity operations and class methods.

Outputting a single scalar value

Here is a class method signature that returns a single scalar value of type **Duration:**

Aircraft.ETA(waypoint : Waypoint ID) :: Duration

It might be used inside another activity like so:

```
current wait time = my aircraft.ETA( waypoint : /is visited next/Waypoint.ID )
```

Outputting multiple scalar values

To output multiple scalar values you produce a single tuple relation where the name of each output parameter is an attribute in the relation.

The method below takes as input a **Tail number** and returns the **angle** and **distance** to the **target**.

Aircraft.Trackto(target : Tail Number) :: (angle : Degrees, distance : Meters)

The corresponding action language within the method looks like this:

```
// compute using in.target input parameter and
// assign results to data variables a and d


=>> ( angle: a, distance: d )
```

The =>> symbol defines an output within an activity.

Outputting a table value

Or you could output them as a single row table:

```
return #( angle: a, distance: d )
```

Now the calling activity can retrieve these values like so:

```
track #= /R8/Aircraft.Trackto( target : nearby aircraft.ID )
a = track.Angle; d = track.Distance
```

This would also work:

```
a, d = /R8/Aircraft.Trackto( target : nearby aircraft.ID )
```

Or even:

```
track #= /R8/Aircraft.Trackto( target : nearby aircraft.ID )
nearby aircraft.(Angle, Distance) #= track  // assign results to attribute values
```

The name doubling simplification used for input parameters also applies to output parameters. If the variable names in a class method or domain operation match the returned attribute names, there is no need to double them. Therefore, this would also work with the above example:

```
=>> ( angle, distance )  // variable names match output parameter names
```

Outputting instance references

It is okay to output one or more instance references. For example, for some class method typed **Aircraft**, where **Aircraft** is a class, you might see this at the bottom of the method activity:

```
// Returns all lowest flying aircraft instance references


=>> Aircraft( ^-Altitude )
```

Outputting a boolean evaluation

If you use the ? symbol a suffix on an instance set or table value, a boolean value will be output.

```
// select all related runways, if any, where "Clear of traffic" attribute is true


=>> /R21/Runway(Clear of traffic)?
```

Without the ? suffix, the output would be an instance set.

Control concepts

Within a single activity, control defines an explicit order in which actions may execute or supplies a predicate evaluation to decide which actions may execute. A set of one or more actions defined to execute under the same conditions are grouped as an action block. Explicit sequence is determined by setting and using sequence tokens. Conditional activation is established by a traditional if-then-else clause or through the use of boolean scalar variables used as guards.

Flow of control

Control is visualized in a data flow diagram as a dashed line that delivers either a sequence token or a diamond that splits out a true/false result. For our purposes, the dashed line is represented by setting and delivering a sequence token. The decision diamond is handled by either with an if-then-else construct or by setting one or more boolean scalar variables for later reference.

Execution order

All sequencing within an activity is determined exclusively by data availability. The vertical order in which the actions are written, for example, is not relevant to action sequencing. To process an activity during runtime, it is necessary to convert an activity into a data flow graph to determine the required sequencing.

The execution of an activity begins immediately, upon either state entry or explicit invocation depending on the activity type, state, method, etc. Any input parameters and any data in the class model is immediately available.

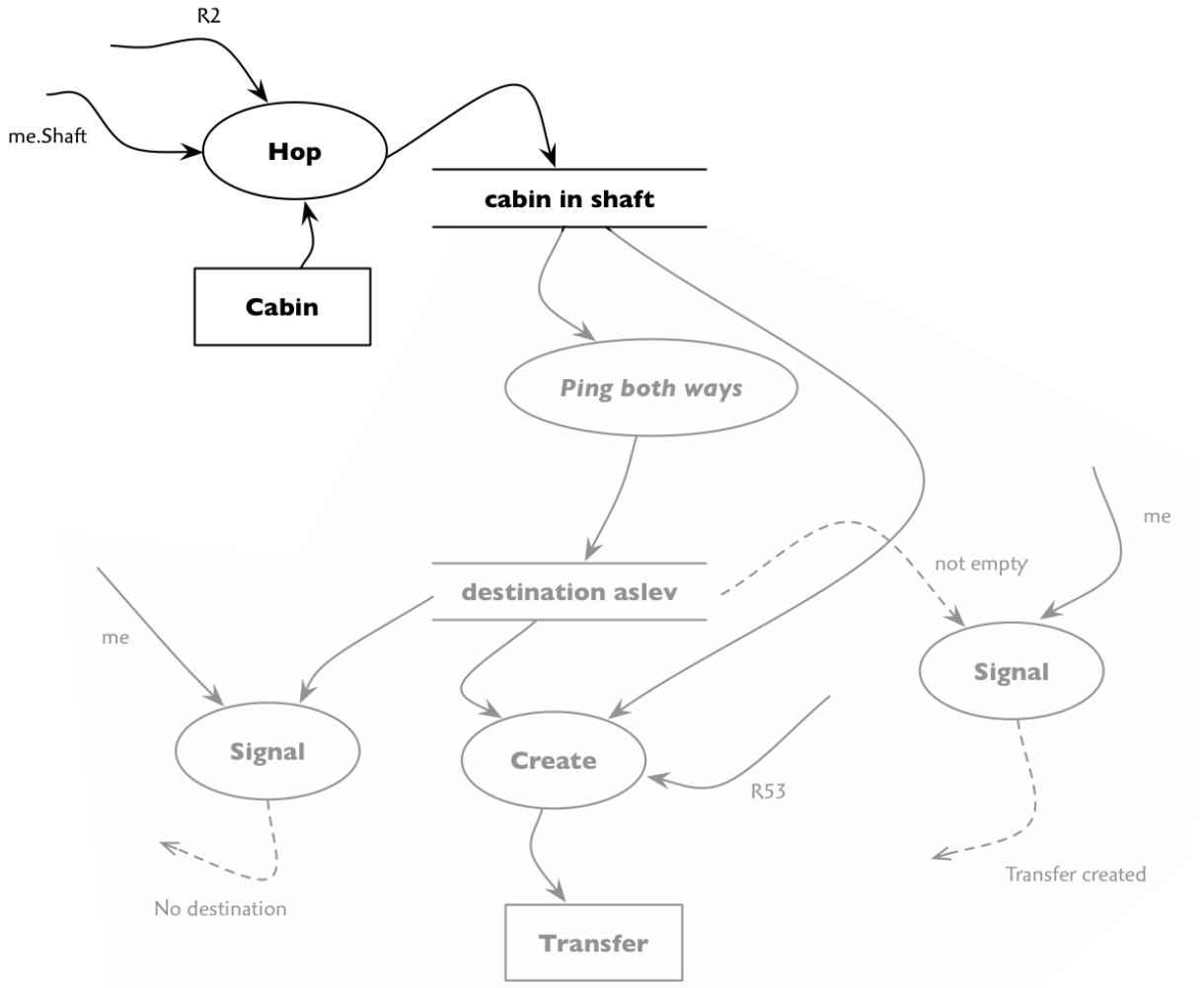
Each action in the activity may execute only when all of its inputs are available. An input to an action is either a data flow, as represented by a variable or a class model access, an input parameter designated in the activity signature, or a sequence token (more about those later).

When an activity first starts, any action whose input consists entirely of input parameters or class model accesses may execute immediately. Any action requiring a sequence token, temporary variable value or output from some other action must wait until these items are made available. On a data flow diagram we can see the order in which inputs will become available and, thus, determine which actions can proceed concurrently and which must be sequenced.

Let's examine the data flow diagram for the following action from the **Searching for new destination** state in the **R53 // Shaft** assigner state model.

```
// The Cabin in this Shaft is stationary and has no Transfer
cabin in shaft .= /R2/Cabin
destination aslev .= cabin in shaft.Ping both ways()
destination aslev? {
    Execute -> *Transfer &R53 cabin in shaft, destination aslev
    Transfer created -> me
} : No destination -> me
```

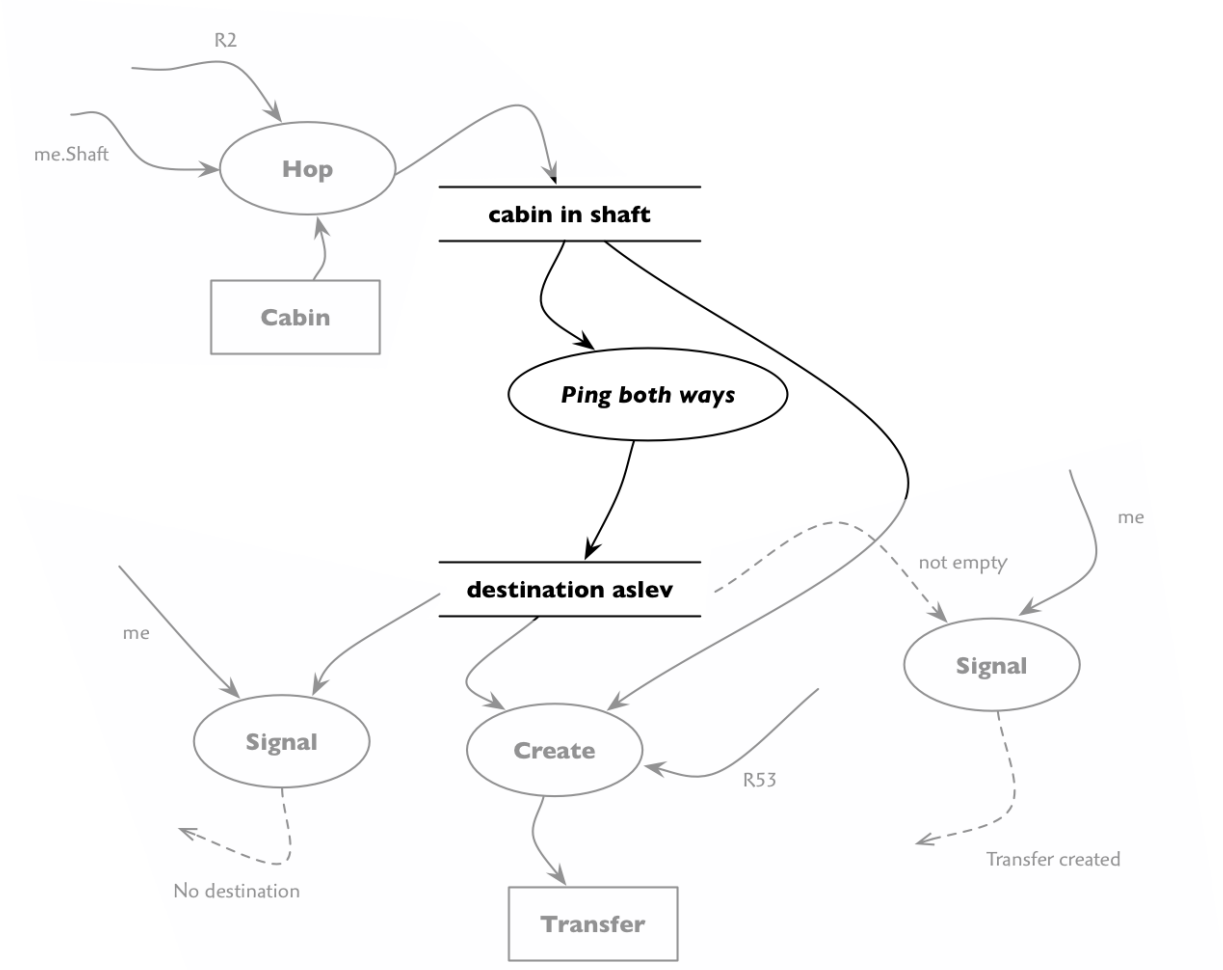
We can view this as the following data flow diagram:



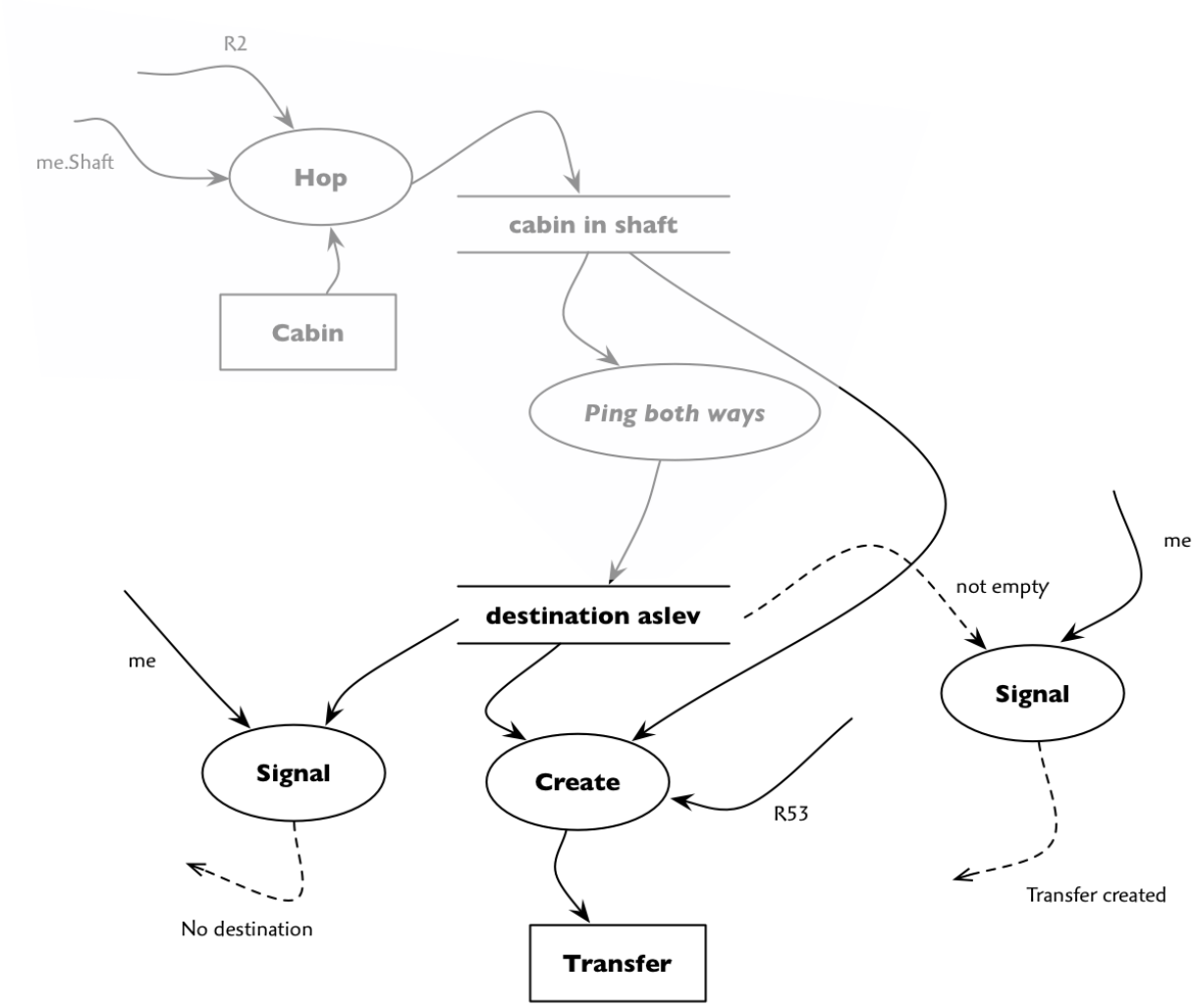
On this diagram, classes are shown as rectangles and temporary variables appear between parallel lines. Solid lines with arrow heads represent the flow of data. A write access points into a class or store while a read access points into an action.

Upon invocation, all input parameters are considered available. The class model content is always available. A variable is available once it is initialized with a value. A sequence token is available upon completion of an associated action block.

The data flows with no origin are either input parameters or data supplied or implicitly assumed in the action text. The R2 relationship name is taken from the **Hop** action's path name text. The `me.Shaft` input is implicitly assumed by context (me, in this case, is the partitioning instance for a multiple assigner, which will be some instance of Shaft). The Cabin class is specified in the path name text and we assume all class model data is available. Therefore, the **Hop** action can execute immediately when the activity starts. Note that all of the faded out actions require data which is not immediately available.



The next action to execute will be the **Ping both ways** method of the Cabin instance selected by the **Hop** action. This method takes no input parameters, but it does produce an instance of Accessible Shaft Level as its output which is assigned to the **destination aslev** instance set variable.



At this point all remaining downstream actions may execute, though there is a conditional dependency determining which set of actions will execute. If no destination was selected by the previous method action and **destination aslev** is empty (false), only the signal action on the left sending the **No destination** signal will execute. Otherwise, the signal action on the right sending the **Transfer created** signal will execute. In either case, the Create action executes, but it will have no effect if the **destination aslev** variable contains the empty set.

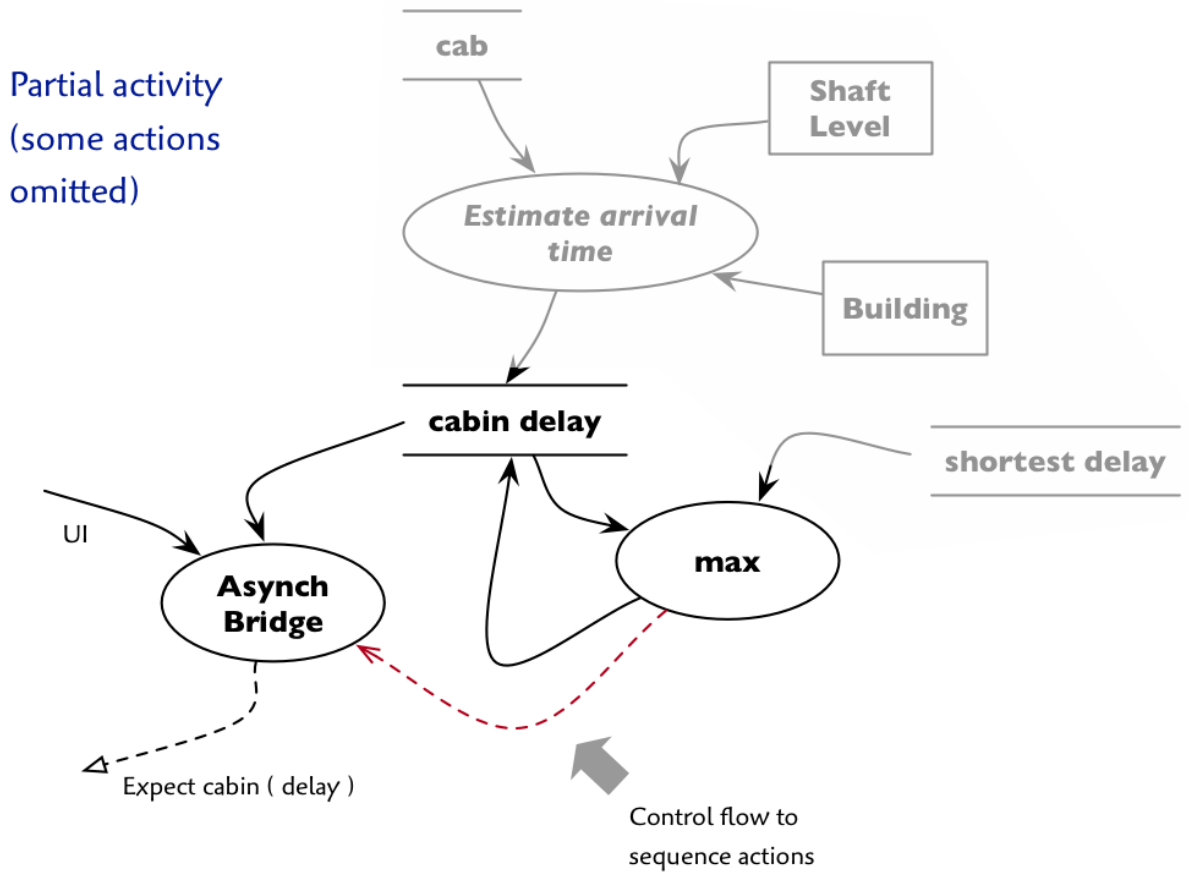
Sequential execution

In most cases, it is possible to work out data dependencies by determining when data variables have been initialized and when they are accessed. If, however, a data variable is accessed more than once, it may be necessary indicate the access order. For example, consider this excerpt from an activity:

```
// BAD ACTIVITY, Don't do this!

cabin delay = cab.Estimate arrival time()
cabin delay = max( cabin delay, shortest delay )
Expect cabin( delay: cabin delay ) => UI
```

In the above partial activity it is unclear whether the **Expect cabin** bridge will send the original value or the adjusted value for **cabin delay**. If we drew this as a data flow diagram, we would add a control flow to ensure a correct sequence, as shown:



The highlighted control flow ensures that the bridge action does not execute until the max function action completes.

In Scrall we use indicate a sequencing control flow by means of a sequence token. A sequence token is named and then set by some action. One or more other actions can then require that token as input before proceeding. The action language is updated as shown:

```
// Now it is okay

cabin delay = cab.Estimate arrival time()
cabin delay = max( cabin delay, shortest delay ) <1>
<1> Expect cabin( delay: cabin delay ) => UI
```

In this above text we've defined a sequence token simply named `1` and then specified that it be set by the second assignment action. We've also specified that the same token is a required input for the bridge action.

A sequence token appears between angled `<>` brackets and can be named like any variable. So we could have called it `<delay trimmed>` instead.

If you want to specify that multiple actions require the same sequence token as input you can group them together as an action block using the `{}` brackets just as you would in an if-then-else structure. An action can set at most one sequence token by placing it to the right of the action.

An action, or set of actions, can be enabled by one or more sequence tokens. If there are multiple, they must all be enabled in order for the sequenced action to proceed. This is what you would expect if you saw an action on a data flow diagram with multiple incoming control flows.

Rather than use a sequence token in the above example, you could have done this:

```
// Better yet

cabin delay = max( cab.Estimate arrival time(), shortest delay )
Expect cabin( delay: cabin delay ) => UI
```

This is probably the better way in this example. In fact, you could compact everything into the bridge action and eliminate the temporary variable. The guideline is to use the simplest, easiest to read formulation for the given activity. Readability is always the primary goal.

Conditional execution

An action or set of actions may or may not execute based on the result of an upstream decision. Scrall provides several ways to handle this situation.

For boolean decisions the familiar if-then-else construct is available in the form of a `? :` style ternary operator much like it is in programming languages like C or Python. Consequently, "if", "then" and "else" are not keywords in Scrall though it is still probably confusing to use those words in variable names. Additionally, boolean scalar variables can be set and named conveniently to explain what a true/false condition actually means. "Cabin going down", for example. In many situations this solution better reflects the data flow semantics and is easier to read than an elaborate if-then-else construct.

Case/switch based decisions where an enumerated type value enables an action block are handled by the the same `? :` syntax used for if-then-else since there's actually little difference to the decision other than having more than two enabling choices.

Guards

With data flow semantics we don't think so much in terms of the programming language style if-then-else. Rather we imagine an action that evaluates an expression and then emits one or more mutually exclusive control outputs. Each output may enable any number of downstream actions.

Single control output

In the simplest case we have the notion of a "guard". A guard is an enabling condition defined as boolean expression. If the expression evaluates as true, the associated action block executes.

The guard must be placed on the leftmost side of a statement, with the action block to the right as shown:

```
(altitude > target.Altitude)? look down -> me
```

Parentheses enclose the guard so that it is easily distinguished from the enabled action. The leftmost placement along with the suffix `?` symbol establishes the expression's role as a guard.

Named control output

A boolean variable often serves as a more readable guard. This is especially the case when you want to use the same guard multiple times within an activity. First set a boolean scalar variable and then you use it as a guard.

```
higher = (altitude > target.Altitude) // set guard


// somewhere further down in the activity, use the guard
higher? look down -> me
```

Parentheses are not required when you use a single variable as a guard. But they are when you do not!

Two control outputs

To specify two mutually exclusive control outputs, add the `:` symbol and a second action block after it.

```
lower? look down-> me : look up-> me
```

The choice between two alternate signal actions with the same sender is so common that you can imply both targets are the same by omitting the first one `:`.

```
lower? look down -> : look up -> me
```

You can use this abbreviation only when the "then" and "else" action each consists of a single signal action. And this is the only case where you may omit the target of an event action.

Two named control outputs

It may be helpful to name both true and false conditions. You can do it like this:

```
higher!lower = (altitude > target.Altitude)
```

The first (leftmost) variable is assigned the result of the expression and the second variable is the negation (not) of the first variable. Then you can do this:

```
higher? {
    // do higher stuff
}

// then after many lines of action language later...

lower? {
    // do lower stuff
}
```

Of course, you can just make the negation explicit if that reads better:

```
!higher ? {
    // do lower stuff
}
```

Multiple named control outputs

Sometimes, though, there are more than two possible conditions to consider:

```
// Location of floor relative to this elevator cabin

(above, across, below) = (Height <, ==, > dest floor.Height)

above? { // above action block }
across? { // across action block }
below? { // below action block }
```

Each comparison on the right yields a value assigned to the corresponding variable on the left. The three guards can be used further down.

Finally, you can combine both approaches:

```
(above, across!away, below) = (Height <, ==, > dest floor.Height)

above? { // above action block }
across? { // across action block }
```

```
below? { // below action block }
away? { // not across action block }
```

Cases

The ? operator, when appended to a scalar variable typed with an enumeration, designates a decision based on the variable value, much like a case/switch statement that you might find in a programming language.

Since Scrall does not support literal values, cases are only represented by enumeration values. Consequently, you must specify a case for each possible value of an enumeration type.

```
vehicle traversing intersection ?
    traffic signal? {
        .red : take photo -> camera
        : conserve power -> camera  // yellow and green
}
```

In the above example, there is one explicit case **red** and then an "other" case which designates all other possible enumeration values. The **traffic signal** variable must have an enumeration type. You could also have done this for the second case:

```
.green, .yellow : conconserve power -> camera
```

It is generally best practice to avoid use of the "other" case since its use makes it unclear that all other options were fully considered by the modeler.

Here, all possible values are made explicit:

```
valve position? {
 .open : close -> my valve
 .closed : open -> my valve
}
```

In the following example cases and guards and sequence tokens are combined:

```
dir? {
 // update attribute value only if input dir is different
    .up:
        !Calling up? Calling up.set <new call>
    .down:
        !Calling down? Calling down.set <new call>
}


<new call> /R38/Bank Level.Request cabin()
```

Each case uses a local attribute value **Calling up** or **Calling down** as a guard. If the input value and attribute value match, the local attribute is set to true (via the .set boolean operation) and the **new call** se-

quence token is enabled. Otherwise, no change occurs and no further action is necessary. If the sequence token is enabled, the **Request cabin()** method is invoked.

If `'dir'` is not of type **Direction**, there will be a static error. Therefore, there is no need for a default or fall-through case.

Boolean case selection

Consider cases on an enumeration with only two possible outcomes:

```
valve position? {
 .open : close -> my valve
 .closed : open -> my valve
}
```

Now let's say that we change the switch variable to a boolean scalar named **valve open**. You can then use a shorter guard expression:

```
valve open ? close -> : open -> my valve
```

Either formulation works.

Subclass cases

A superclass instance's activity may behave differently depending on the associated subclass instance. The ? symbol can be postfixed on a generalization relationship name to generate a case for each modeled subclass name on that relationship.

You can then switch based on the subclass as shown:

```
R1? {
   .Fixed Wing Aircraft:  // must be a subclass name on R1
       // land on runway
   .Rotary Wing Aircraft:
       // land on helipad
}
```

If no instance is specified, the local instance is assumed. Otherwise, make the source instance explicit:

```
some aircraft/R1? {
   // subclass cases
}
```

Signaling an instance set

The signaling of an event is expressed with the –> symbol. Here a signal is sent from a **Cabin** instance to its associated **Door** instance.

```
Unlock –> /R4/Door
```

One signal is sent to each instance in the target selection.

```
Take out of service –> my bank/R1/Shaft // All Shafts in the Bank are deactivated
```

The `me` keyword is used when the target of the signal is the local instance.

```
Continue –> me  // Move self on to the next state
```

You need to first select the destination instances, via navigation, criteria based selection or both so that you can address the signal. If you end up with the empty set in your selection, no signal will be sent. You will often want to specify some action to be executed in the case where zero target instances were selected. You will need an instance set variable to test the cardinality as shown:

```
ac to land .= Aircraft( Tail number: in.aircraft to land )
Land ( Runway: designated runway ) –> ac to land
!ac to land ? No aircraft to land –> me
```

The above works, but here is a more readable, and hence preferable, formulation:

```
ac to land .= Aircraft( Tail number: in.aircraft to land )
ac to land ?
    Land ( Runway: designated runway ) –> ac to land :
    No aircraft to land –> me
```

Sending a delayed signal

You can specify a duration to wait before dispatching an event to the target instance using the at `@` symbol:

```
Off -> my heater @ heating interval
```

Here, the variable `my heater` (note that `my` is NOT a keyword) gets the event after the `delay` duration (as close to it as the MX can manage). To send a signal at a specific time, you can also use the `@` symbol and supply a time value instead of a duration.

```
On -> the air conditioner @ time to cooldown
```

In this example, a time value is stored in the `time to cooldown` variable. To avoid confusion for the readers of your action language, name your times and delays appropriately as in the examples above.

An error results if the scalar value after the `@` is not a time or duration based data type. No, you can't just use Integer!

Canceling a delayed signal

Use the cancel signal `!->` symbol to cancel a delayed signal:

```
On !-> the air conditioner @ time to cooldown
```

Signaling an assigner state machine

A single assigner is a state machine that manages competition on an association. Since there can be at most one instance of a single assigner per association, you can just address the signal to the association name like so:

```
Waiting for clerk -> /R1
```

A multiple assigner provides one state machine instance per instance of some designated partitioning class. Extending the Customer-Clerk example we might manage competition separately within each Department. So the Department class is designated as the partitioning class and we have one state machine per Department instance. Therefore, you must provide both the association name and the Department instance to find the correct assigner state machine as shown:

```
my department .= /R3/Department
Waiting for clerk -> /R2(my department)
```

In a state activity of either kind of assigner, the `me` keyword refers to the local state machine instance and not to any class instance.

For each

Scrall's basis on data flow and relational semantics eliminates the need for an explicit "for each" action.

In the following example, all **Dogs** past a certain age are classified as 'old' by setting a status attribute.

```
Dog( Born < in.date).Status = .old
```

Consider this action language from the Elevator case study. Here a **Bank Level** instance (**Floor** in a **Bank**) is responding to an up/down floor call button by selecting the most suitable **Cabin** to respond to the call. Here is what it might look like in xtUML's (BridgePoint) OAL (object action language):

```
shortest_delay = 0.0;  // seconds
first_cabin = true;
param.OUT_Shaft = "";

// Select one of the cabins with the shortest delay
select many bank_Cabins related by my_Bank->Shaft{R1}->Cabin{R2};

for each this_Cabin in bank_Cabins
  select one its_Shaft related by this_Cabin->Shaft{R2};
  if (its_Shaft.In_service)
    cab_delay = this_Cabin.Estimate_travel_delay(
     Floor:my_Floor.Name, Calling_dir:param.Dir
    );
    if ((cab_delay < shortest_delay) or (first_cabin))
      shortest_delay = cab_delay;
      param.OUT_Shaft = its_Shaft.ID;
    end if;
  end if; // in service
  first_cabin = false;
end for;

if (param.OUT_Shaft != "")
  return true;
else
  return false;
end if;  // OUT_Shaft will be "" only if all Shafts are out of service
```

In Scrall, you could replace all that with two lines:

```
inservice cabins ..= /Bank/Shaft( In service )/R2/Cabin
```

```
=>> inservice cabins(1, ^-Estimate travel delay(
 Floor: /Floor.Name, Calling dir: in.Dir).Shaft
```

The first line selects all **Cabin** instances whose **Shaft** is inservice.

The second line invokes the **Estimated travel delay** method of each selected **Cabin** instance and chooses one with the least delay reporting its **Shaft** attribute value (which happens to be a referential attribute to a **Shaft ID**. Done!

In fact, the whole thing could have been done with one line, but at some point you violate the principle of making the activity easy to read. In that vein, the two line solution could easily be expanded to three with the introduction of an additional variable (**best cabin**) to make it easier for humans to parse.

But the OAL solution returns an **ID** value because it is not possible output instance references. In Scrall, we would probably just return the selected **Cabin** instance references which, could be more than one since two **Cabins** might have the same estimated delay.

```
=>> /Bank/Shaft( In service )/R2/Cabin(
  ^-Estimate travel delay( Floor : /Floor.Name, Calling dir: in.Dir )
 )
```

Now the calling action can be:

```
servicing shaft .= Choose shaft( 1, in.Dir )
```

Iteration

It may be necessary to apply an action block successively to an ordered set of instances. A series of **Waypoints** may be serialized to logging domain. Here it may be important that **Waypoints** be serialized in the order they should be followed. Or in a signal IO domain it may be necessary to write a series of output values in a particular order.

In both cases it is necessary to specify the instance order somehow and then to specify the actions to be repeated for each ordered instance.

Specifying an ordered instance set

There are two ways to define an ordered set of instances. You can use a reflexive association and specify an initial instance. Or you can specify one or more attributes each in either ascending or descending order.

```
<< /OR1/next >> {
   DRIVER.Write( PointID : ID, Value )
}

<< Aircraft( +^Altitude, -^Airspeed ) >> {
   LOG.LogData( Item1: Altitude, Item2: Airspeed )
```

```
}
```

If no order is specified, the implementation will choose whatever order suits it.

```
<< Aircraft >> {
    LOG.LogData( Item1: Altitude, Item2: Airspeed )
}
```

In the above example, all instances of **Aircraft** will be logged, but in no particular order.

Domain interactions

Interactions initiated by external domains invoke domain operations in the local domain. Interactions initiated by the local domain outward are mediated by operations on external entities. An external entity is a model element that serves as a proxy for an external domain.

Both external entity and domain operations are specified with action language.

Domain operations

A domain operation is defined much like a class method with input and output parameters. Domain operations are invoked synchronously so it is possible to produce output values.

The same action language used to yield output values in a class method activity can be used in a domain operation.

External entity operations

To interact with an external domain either a synchronous or an asynchronous operation may be invoked on an external entity.

An external entity serves as a proxy for the target domain. It may, or may not be named the same as that domain. For example, the ROBOT external entity might actually be connected to a Signal IO (SIO) domain.

An asynchronous operation does not produce any output parameters since it implies a deferred response from the external domain. To invoke this kind of operation, say from a state activity, an outgoing asynchronous signal is sent using the => symbol like this:

```
Open door( Shaft ID : Shaft ) => SIO
```

A synchronous operation may produce output parameters just like a class method or domain operation.

```
press status = SIO.Above inject pressure()
```

Here, the **Above inject pressure()::Boolean** operation is invoked on the SIO external entity. It completes and assigns a boolean result to the **press status** variable.

Time

The current system time is available via the system supplied Date type. This type specifies “now” as the default value to use whenever a new variable is initialized to that type. A conversion operation can be specified which provides “now” in the desired format.

This type supports a variety of conversion operations which extract the desired time components as tuples such as H, M, S, ms, HMS, MDY, DMY, etc. So you can do this kind of thing:

```
current time = Date.HMS
today = Date.MDY
```

It is important to avoid naming collisions with system supplied types. At the moment these are Integer, Rational, Boolean, String, Nominal, Ordinal and Date. Since most types used by the modeler will be constrained versions of the system types (Pressure, instead of Rational, Tail Number instead of String) or base types such as Boolean, this should not be much of an encumbrance.

Assertions

Especially during debugging, but even in production code, assertions make it possible to catch a bug and stop further execution while providing useful diagnostic information.

The form is:

```
>>> <predicate> "<message>"
```

From the **Building** class in an elevator application

```
>>> /R48/Floor "Building {Name} has no floors"
```

If the above action is executed by an instance of **Building**, and there is not at least one related **Floor** instance, a fatal exception will be thrown along with default diagnostic information.

The default diagnostic information indicates the exact location of the assert statement that triggered the exception and the predicate that failed. The values of any attributes or variables can be supplied between {} brackets. Since they appear between quotes, the variable brackets won't be confused with action grouping brackets.

Naming conventions

Here are some naming conventions for xUML model element and variable names.

Symbol names

Symbols include variable, attribute, class, parameter, event, method and operation names.

Internal spaces allowed

Symbol names follow the usual alphanumeric conventions with the exception that the space character may be used as a delimiter. So the name `incoming aircraft`, for example, is a legal name.

System symbol names

Values and functions provided by the model execution platform are preceded by an underscore.

Naming conventions

For ease of reading, a number of model naming conventions are recommended, but not enforced.

Variable names – Lower case: `my aircraft .= /R1/Aircraft`

Class names – All initial caps: `Air Traffic Controller`

External entity names – All caps: `UI`, `ALARMS`

Attribute names – First initial cap: `Maximum altitude`

Parameter names – All lower case: `(landing runway, radio freq)`

Data types – All initial caps: `Compass Direction`

Relationship phrase names – Lower case: `/R3/is taking off from`

Event, operation and method names – First initial cap:

- Event: `Land planes -> Air Traffic Controller`
- Operation: `UI.Send warning( Message )`
- Method: `Off Duty Controller.Check break duration()`

Transitory states – First initial cap: `Checking temperature`

Wait states – All Caps: `RAISING LANDING GEAR`

(Note that state names are never referenced in Scrall, but it is helpful to see all of model naming conventions in one place).

Case insensitive names

To avoid confusion and bad naming practices, symbol names are case insensitive.

Keywords

There are not many keywords in Scrall by design. Consequently, you have a lot of freedom when it comes to naming things, but not total freedom. Avoid using the following keywords in names.

```
its // relative to nonlocal instance in reflexive hop
me // for sending signals to self
and, or, not // in comparison/selection predicates
in // marks input parameter before .
true, false // for convenience
```

Symbols

As in most development languages, symbols are used for common math, logic and comparison operations. Symbols also take the place of keywords so that names can be more readable, expressive and include whitespace.

On the downside, there are a lot of symbols to memorize. As much as possible, visual mnemonics are employed to make this task easier. For example, # means table and . means instance/object.

Symbol	Context	Usage
*	*	create instance
	restriction	select all instances
	projection	select all attributes
	*I projection	project all identifer attrs
	computation	type specific op (multiply)
=	all	scalar value assignment
#=	all	table value assignment
#.=	all	assign at most one tuple
.=	all	assign at most one instance
..=	all	assign 0, 1 or many instances
0	restriction	select no instances
1	restriction	select at most one instance
*In	projection	project all In attributes
-attr	projection	select all but attr
~	path	repeated hopping
->	all	send signal
@	all	time of signal delivery
=>	all	asynch bridge
:	restriction	equals, matches
:	projection	rank delimiter
!=, ==, <, >, <=, >=	all	comparison
	after action same line	set sequence token
	before action same line	token enables action
<>, |<>|, <>|, |<>	all	between inclusive/exclusive compare
<, <, >, >	between table values	set comparison
//	all	comment
&	&R	relate instances
&	before assoc class	initialize assoc class attrs
!&	all	unrelate
[ ]	[boolean scalar] rightmost	set guard
[ ]	[boolean scalar] leftmost	enable action block

!	(!boolean scalar)	not
;	all	sequences actions with implicit seq tokens
/	all	hop across relationship
^+, ^-	before item in restriction	greatest, least
^+, ^-	after : in projection	rank ascending, descending
^>, <^, >, <	all	next greater, next lesser
^, +, *, - –	all	set operation on table value
+=, -=	all	combine assignment
#!!	all	image operation (see c.j. date)
^, +, -	between table values	set operations
#[ ]	#= assignment	table constructor
{ }	all	action block
( )	rhs boolean assignment	boolean expression
( )	in computation	math precedence
( )	after class name	restriction
( )	.( )	projection
( )	signal action, method, bridge	parameter list
.	class.attribute, table.attribute	attribute access
.	.enumvalue	enumerated element value
::	data type::variable	declare data type
??	before instance set or table variable	cardinality
>>	projection a >> b	rename attribute
>>	projection >>c.(formula)	extend
>>	migration	migrate instance
+	+ in projection	extend operator
=>>	in non-state activity	return value
?	after boolean expr	if-then
?	after enum value	switch

Bibliography

Executable UML

Models to Code (With No Mysterious Gaps), Leon Starr, Andrew Mangogna, Stephen J. Mellor, Apress 2017, ISBN 978-1-4842-2217-1

Executable UML: A Foundation for Model Driven Architecture, , Stephen J. Mellor, Marc Balcer, Addison-Wesley 2002, ISBN 0201748045

Elevator 3 Case Study (to be published in June 2019) and available from modelint.com

Relational Theory and Predicate Logic

Relational Theory for Computer Professionals, C.J. Date, O'Reilly 2013

Applied Mathematics for Database Professionals, Lex de Haan, Toon Koppelaars, Apress 2011

Databases, Types, and The Relational Model: The Third Manifesto, 3rd edition, Addison-Wesley 2006, ISBN: 0-321-39942-0

(available at www.thethirdmanifesto.com)

Open Source Code Generation

Andrew Mangogna's designs and writings have been greatly influential and, most importantly, key to providing a reality check so that there is a clear path to code from Scrall.

Micca - xUML Translation Tool (wade through the documentation!)

http://repos.modelrealization.com/cgi-bin/fossil/mrtools/wiki?name=MiccaPage