Dispel Tutorial Documentation

Release 0.8

Paul Martin

August 31, 2012

CONTENTS

This tutorial introduces the Dispel workflow language, a strongly-typed imperative programming language used to construct logical descriptions of streamed-data workflows to be executed on distributed architectures. It describes many of the language's key features and tries to explain some of the thinking behind various design decisions.

CHAPTER

ONE

GETTING STARTED

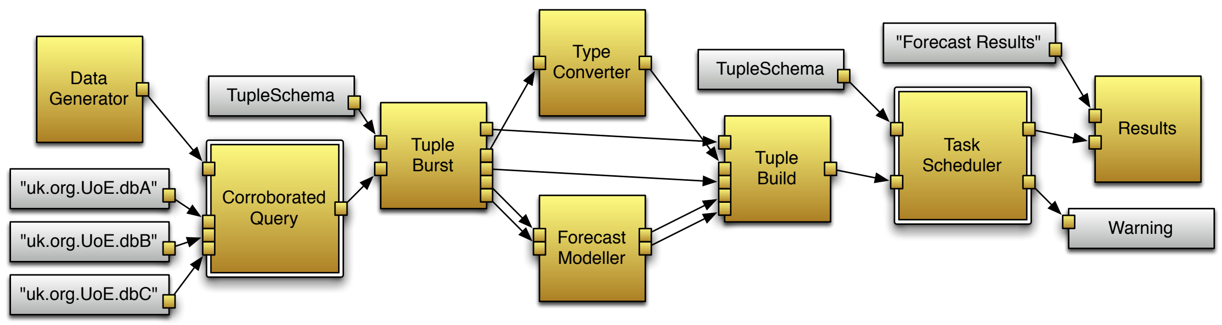
Dispel is a strongly-typed imperative language for generating executable workflows for data-intensive distributed applications, particularly (but not exclusively) for use in computational sciences such as bioinformatics, astronomy and seismology. It has been designed to be a portable *lingua franca* by which researchers can interact with complex distributed research infrastructures *without* detailed knowledge of the underlying computational middleware, all in order to more easily conduct experiments in data integration, simulation and data-intensive modelling.

Dispel was created as part of the ADMIRE project, which sought to promote a model for advanced data mining and integration which insulates the computational scientist or domain expert from the specifics of how individual computational services are implemented or how data is moved between physical resources.

1.1 Core concepts

A *workflow* is simply a decomposition of a task into a number of sub-procedures which, when linked together, describe how that task can be carried out. Such workflows can be understood in terms of *data-flow*, wherein the output of one sub-procedure feeds directly into the next sub-procedure. If a task involves a continuous processing of data over some period of time, then each sub-procedure can begin enactment as soon as data produced by prerequisite sub-procedures starts to emerge. If these sub-procedures can be distributed amongst a number of different actors and a means to efficiently deliver data between actors can be provided, then the workflow can be effectively parallelised, and results can start being produced almost immediately regardless of the ultimate size of the base task.

A Dispel workflow is a decomposition of an application into a number of processing elements connected together via channels though which data can be streamed. Each *Processing Element* (PE) encapsulates some sub-procedure deemed pertinent to the task at hand and takes some number of input connections as well as some number of output connections. As a PE consumes data through its inputs, some operation is performed within the element which results in a number of new streams of data produced through its outputs. The rate of data consumption and production depends on the behaviour of the PE and its neighbouring elements.



An example of a Dispel workflow.

A Dispel script describes how to construct a workflow; scripts must be executed by an interpreter capable of mapping the resulting workflow onto a suitable *enactment platform*, being a collection of services and middleware distributed onto physical resources which can actually handle the execution of workflow sub-procedures. Such an interpreter is usually provided by a remote *gateway* which can execute any Dispel script submitted through it on an enactment platform provided by the (distributed) system to which it is attached. The gateway serves to conceal the vagaries of implementation and physical topology of resources from the casual user, instead presenting a library of PEs implementable by the enactment platform which can be enlisted in the construction of workflows. In order to provide a common library to Dispel users, the gateway usually defers to a central *registry* (though other arrangements are possible), whilst implementation code for registered PEs on a given enactment platform is kept in a nearby *repository*. Provided that a valid workflow has been submitted, a gateway will automatically map workflow components to services and processes implemented by the enactment platform, delegate their execution to suitable pre-configured computational resources and then choreograph the execution of the distributed 'complete' workflow.

1.2 Gateways

In order to use Dispel, there must be a service able to interpret Dispel scripts and map them onto an actual enactment platform. Whilst such a service can be configuration on a user's own machine for simple tasks, the scenarios envisaged for Dispel generally involve submission of scripts to a remote gateway which acts as an interface onto some kind of distributed system — such a distributed system could be a federation of computational resources including large data archives and high-performance compute clusters.

An ADMIRE gateway maps Dispel requests onto OGSA-DAI workflows — these workflows describe a concrete distributed implementation of the logical workflows described by a Dispel script by replacing individual PEs with compositions of OGSA-DAI activities. An ADMIRE gateway can be stand-alone, or part of a federation of gateways; a gateway can be configured to refer to an independently-hosted registry and repository, or have its own in-memory private registry for stand-alone use.

To execute the Dispel examples given in this tutorial, you must either use an existing gateway, or install a new one. The installation and deployment of a gateway is described here; the ADMIRE gateway is a Java Servlet (JRE version 1.6+), typically hosted using Apache Tomcat (version 6+). It uses a RESTful HTTP interface for the submission of Dispel and the retrieval of workflow state.

CHAPTER

TWO

BUILDING SIMPLE WORKFLOWS

A basic Dispel script consists of a series of statements handling the importing of useful processing elements from the local gateway's registry, the instantiation and configuration of those elements and either the registering of new composite processing elements or the submission of a workflow to be executed by the gateway (or both). This section shall introduce some of the core functionality of Dispel, demonstrating how to construct simple workflows for submission.

2.1 Skipping 'Hello World'

Processing elements (PEs) describe the principal components which make up any workflow. An active gateway will provide a number of fundamental PEs which are commonly used in various data-intensive applications; more may be added by data analysis experts and software engineers. Such experts can also use Dispel itself to define *composite* PEs, building an increasingly sophisticated array of pre-fabricated components for users to exploit in their workflows.

Available PEs are described within the registry associated with the gateway to which a script is submitted. We can import a PE description from the local registry by invoking the `use` directive:

```
use dispel.db.SQLQuery;
```

The above command extracts a PE named `dispel.db.SQLQuery` from the registry; it also imports the identifier `SQLQuery` into the local namespace, which means that we can drop the prefix '`dispel.db`' when referring to this PE within this Dispel script. An `SQLQuery` converts queries written in SQL into responses returned by a selected database. Every useful PE has some combination of input and output connections — a few may act as sources (only outputs) and a few may act as sinks (only inputs), but the majority will have at least one input and one output. `SQLQuery` has two inputs and one output; `expression` (into which queries may be fed), `resource` (into which the location of the database to be queried must be fed in tandem with every query) and `data` (from which the results of any queries will emerge).

In order to make use of any imported PE however, we must first create an instance of the PE. A new instance can be created by use of the `new` directive:

```
SQLQuery query = new SQLQuery;
```

Here we are defining a *PE instance* named `query`, which is of course an instance of the `SQLQuery` PE. To do anything useful, we need to connect each input of `query` to a suitable data stream. Such data streams typically come from the output of other PEs, but we can also feed in data directly from a script:

```
|-"SELECT * FROM littleblackbook WHERE id < 10"-| => query.expression;
|-"TutorialDatabase"-| => query.resource;
```

In this case, both inputs (`query.expression` and `query.resource`) read in text strings, one describing an SQL query and the other identifying a database (in this case via the identifier by which it is known within the distributed system associated with the gateway).

Theoretically, this is enough to describe a 'useful' workflow. We need only invoke the `submit` directive, and the gateway will be able to provide an implementation of `SQLQuery` with which to query the referenced database with the specific query given:

```
submit query;
```

Of course, since we didn't connect the output of `query` to anything in particular, we can only actually find out the result of our query if we are able to somehow inspect `query` directly as it executes, which we can assume to be unlikely and (in many cases) undesirable. Usually, what we want to do instead is channel the output of our workflows towards a sink PE which can report back its input, whether (for example) by saving it to a file in a known location or by directly visualising it using a portal widget or even in a terminal display. So instead of simply submitting the workflow as is, we direct the output of `query` towards an instance of the `Results` PE and submit that instead. PE `dispel.lang.Results` has two inputs — `input` which takes data to be recorded, and `name` which associates a name with the data recorded for ease of reference. In the complete Dispel script below, `query.data` is connected to `results.input`, storing the result of the given query somewhere which can be directly accessed after the workflow is enacted.

```
// Import PEs from the local registry.
use dispel.db.SQLQuery;
use dispel.lang.Results;

// Create instances of PEs.
SQLQuery query   = new SQLQuery;
Results  results = new Results;

// Construct workflow and feed in data.
|-"SELECT * FROM littleblackbook WHERE id < 10"-| => query.expression;
|-"uk.org.UoE.dbA"-| => query.resource;
|-"10 entries from the little black book"-| => results.name;
query.data => results.input;

// Submit the entire workflow.
submit;
```

The entire workflow is submitted by simply invoking `submit` without any additional parameters. Submitting a single component (such as `query` or `results`) will also submit any workflow which the component is part of.

2.2 Describing workflow input

In the previous section, we fed data into an instance of the `SQLQuery` PE directly, but only fed in a single set of inputs. We did this using what is referred to as a *stream literal*. A stream literal describes the content of a data stream explicitly as a sequence of data elements drawn from the Dispel script itself, rather than as the output of a PE instance:

```
|-input1, input2, ...-|
```

This sequence of elements can consist of any number of arbitrarily complex data structures built from a number of elementary data types (booleans, integers, character strings, *etc.*) as dictated by PE requirements and the whims of the workflow designer. For now however, we shall concentrate on simple inputs of text strings.

It is trivial to describe a list of homogeneous inputs as a stream literal — say we want to define not one but three queries to stream into an instance of the `SQLQuery` PE. We might write the following:

```
String query1 = "SELECT name FROM littleblackbook WHERE id <= 10";
String query2 = "SELECT name FROM littleblackbook"
              + "WHERE id > 10 AND id <= 20";
String query3 = "SELECT address FROM littleblackbook"
```

```
              + "WHERE name = 'David Hume'";

|-query1, query2, query3-| => query.expression;
```

This is not enough however. An instance of `SQLQuery` takes not one but two inputs, and consumes data from each input at the same rate. Thus, for each query made to a PE instance like `query`, a reference to a database to query must be given. If we wish to direct each query to a different database, then we could legitimately write the following:

```
String database1 = "uk.org.UoE.dbA";
String database2 = "uk.org.UoE.dbB";
String database3 = "uk.org.UoE.dbC";

|-database1, database2, database3-| => query.resource;
```

If we want to direct all queries to the *same* database however, then it seems rather inefficient to repeat the same string multiple times manually. Fortunately, we can use a `repeat` expression to write this more elegantly:

```
|-repeat 3 of "uk.org.UoE.dbA"-| => query.resource;
```

In fact, if we know that we shall always be feeding the same input to a particular PE instance, then we can use the keyword `enough` to essentially lock the input to a given value no matter how much data flows through the PE instance's other inputs:

```
|-repeat enough of "uk.org.UoE.dbA"-| => query.resource;
```

This is especially useful if other inputs are connected to the output of 'black box' PEs which produce an unpredetermined volume of data for processing — such as in the example below:

```
// Import PEs from the local registry.
use dispel.db.SQLQuery;
use dispel.tutorial.PrecociousChild;

// Create instances of PEs.
PrecociousChild child   = new PrecociousChild;
SQLQuery        query   = new SQLQuery;
Results         results = new Results;

// Construct workflow and feed in data.
child.output => query.expression;
|-repeat enough of "uk.org.UoE.dbA"-| => query.resource;
|-"Adult responses"-| => results.name;
query.data => results.input;

// Submit the entire workflow.
submit;
```

It is worth noting that in the above script, the input to `results.name` is *not* repeated — this is because unlike instances of `SQLQuery`, instances of `Results` only read data through their `name` input once. If we want `query` to behave in the same way, we will need to adapt `SQLQuery` to better suit our purposes. Another notable omission in the above script is any statement to import the `Results` PE — this is because actually all PEs in `dispel.lang` are automatically imported. Thus we will omit any explicit import in all further examples.

2.3 Defining new types of processing element

It might be felt that having to remember to ensure a continuous supply of identical inputs to a PE is undesirable, and that it should be possible to configure a PE which, for example, will always refer to the same database for a given stream of queries. It is possible to do such a thing in Dispel by adapting the functionality of existing PEs. These customised PEs can then be inserted into a workflow immediately, or registered in order to be used (and reused) later.

There are two ways to adapt a PE to serve our purpose. One is to modify the behaviour of the connection interfaces associated with a PE, which we shall defer to later in this tutorial; the other is to wrap one or more PEs within a new PE. In order to produce a new PE, it is necessary to define its abstract behaviour. Since we consider PEs in Dispel essentially as black boxes taking in a set of inputs and producing a set of outputs, it is only to be expected that we classify PEs by their connections. Formally, a *connection* is a link between two *connection interfaces* — we have made several such connections already:

```
child.output => query.expression;
|-repeat enough of "uk.org.UoE.dbA"-| => query.resource;
query.data => results.input;
```

The connection operator => connects the left interface (for example `child.output`) to the interface on the right (in this case `query.expression`). A stream literal is therefore an *ad-hoc* connection interface. Note that stream literals can only be found on the left-hand side of connections — we cannot feed the output of a PE into a stream literal, nor can we connect two stream literals together.

An *external* connection connects an output of one PE (or stream literal) to the input of another. A given output can be connected to many different inputs, though each input can only receive data from one source — if we wish to merge multiple outputs into a single input, then we must use a suitable PE which knows how to interpolate the outputs correctly, like `dispel.core.Combiner`. Meanwhile, an *internal* connection connects all inputs of a PE to all outputs of the same PE — in other words, we can describe a PE abstractly by the internal connection made between its interfaces with outside data streams. Thus, when we define a PE, we define it by describing such an internal connection in the following format:

```
PE( <Connection input1;  Connection input2;  ...> =>
    <Connection output1; Connection output2; ...> );
```

Consequently, we can define the PE `SQLQuery` as so:

```
PE( <Connection expression; Connection resource> => <Connection data> );
```

Whilst for a source PE like `PrecociousChild` (used in the previous script):

```
PE( <> => <Connection output> );
```

Such definitions can quickly get cumbersome however, which is why we use aliases like `SQLQuery` and `PrecociousChild`. It is possible to define new aliases by using a `Type` declaration. Say that we wish to define a new PE which, like `SQLQuery`, reads queries from a data stream and produces a list of responses, but unlike `SQLQuery`, always refers to a specific database. We can define the abstract PE type as so:

```
Type SQLToTupleList is PE( <Connection expression> => <Connection data> );
```

From now on, whenever we want to create a new PE which takes in an expression and produces data, we can refer to it as a sub-type of `SQLToTupleList`. An `SQLToTupleList` is just an abstract PE however — to have a PE which we can instantiate and use, we need PEs with internal architecture which can actually be implemented using concrete computational components. This is where the notion of a PE constructor comes in.

2.4 Deriving new workflow elements

A PE constructor is a particular type of function which returns implementable descriptions of abstract PEs. Consider the constructor `lockSQLDataSource`:

```
PE<SQLToTupleList> lockSQLDataSource(String dataSource) {
    SQLQuery query = new SQLQuery;
    |-repeat enough of dataSource-| => query.resource;
    return PE( <Connection expression = query.expression> =>
               <Connection data = query.data> );
}
```

Like any other function which might be found in a functional or imperative programming language, it has a function head and a function body. The function head consists of a return type (in this case `PE<SQLToTupleList>`), a function name (`lockSQLDataSource`) and an ordered set of parameters (here, only one — `String dataSource`) The function body is a set of statements ending with a `return` directive. A function which returns a description of a PE must return a PE internal connection which matches that of the abstract PE given in the function head (in this case, an input `expression` and an output `data`).

In the case of `lockSQLDataSource`, the function describes how to make implementable `SQLToTupleList` by taking a new `SQLQuery` instance named `query`, and then locking the `resource` connection interface within that PE instance to the string value passed to it. It would then simply return a version of `SQLToTupleList` wherein the `expression` and `data` interfaces are attached to those of `query`. Using this function, successive implementable versions of `SQLToTupleList` can be defined by invoking `lockSQLDataSource` with a suitable instance of parameter `dataSource`:

```
PE<SQLToTupleList> TutorialQuery = lockSQLDataSource("uk.org.UoE.dbA");
```

Having created the new implementable PE `TutorialQuery`, it is now possible to create an instance of that PE and use it in a workflow:

```
TutorialQuery query = new TutorialQuery;
```

Thus a PE constructor describes how existing PEs can be used to produce a desired composite PE which implements a given abstract PE. In this case, `query` is a PE instance, an instance of `TutorialQuery`. Meanwhile `TutorialQuery` is a (rather trivial) composite PE, of type `PE<SQLToTupleList>` — in other words, `TutorialQuery` is an implementable version of the abstract PE `SQLToTupleList`. Thus `SQLToTupleList` exists principally to describe the kind of internal connection exhibited by PEs like `TutorialQuery`; the role of abstract PEs is to provide vessels into which to insert compositions of other PEs like `SQLQuery`. Composite PEs will be deconstructed on execution by the ADMIRE gateway into their constituent PEs, which will in turn be repeatedly deconstructed until only 'primitive' PEs remain for which there exists concrete implementations. Instances of these implementations may then be distributed and executed on many different resources, but to the user, there is only one top-level PE instance to concern themselves with.

Having defined a new type of abstract PE, a constructor for making implementable versions of that abstract PE, and then an example of such an implementable PE, it would be helpful if those entities could be preserved for future workflows. That is the role of the `register` directive:

```
register SQLToTupleList, lockSQLDataSource, TutorialQuery;
```

The command to `register` directs the ADMIRE gateway to record the given entities within its local repository, registering their existance within that repository in its local registry. It is possible to register not only derivative PEs, but as demonstrated, abstract PEs and PE constructors (though not PE instances, which exist only for the lifetime of a given workflow).

However it is not generally a good idea to register new entities without placing them into some kind of intelligent hierarchy, so as to allow them to be easily located by other users, and to avoid overwriting other entities which happen to share the same name. This is done by placing entities into *packages*, which can themselves hold further sub-packages. We have already encountered packages before when importing PEs such as `SQLQuery`:

```
use dispel.db.SQLQuery;
```

The above directive states that PE `SQLQuery` resides in package `db`, which itself resides in `dispel` (as it happens, `dispel.db` is the main database processing package of Dispel).

When registering new entities, we want to put them within a particular package along with similar entities. To do this, we wrap entire Dispel scripts within `package` directives:

```
package tutorial.example {
    ...
}
```

Any invocations of `register` within a given `package` environment will automatically be registered within the package named (in the above example, `tutorial.example`). Thus, a full Dispel script introducing new workflow components to the ADMIRE framework will take on an appearance not unlike that of the Dispel script below:

```
package tutorial.example {
    // Import existing PE from the registry.
    use dispel.db.SQLQuery;

    // Define new PE type.
    Type SQLToTupleList is PE( <Connection expression> => <Connection data> );

    // Define new PE constructor.
    PE<SQLToTupleList> lockSQLDataSource(String dataSource) {
        SQLQuery query = new SQLQuery;
        |-repeat enough of dataSource-| => query.resource;
        return PE( <Connection expression = query.expression> =>
                   <Connection data = query.data> );
    }

    // Create new PEs.
    PE<SQLToTupleList> TutorialQuery = lockSQLDataSource("uk.org.UoE.dbA");
    PE<SQLToTupleList> MirrorQuery   = lockSQLDataSource("uk.org.UoE.dbB");

    // Register new entities.
    register TutorialQuery, MirrorQuery;
}
```

If we then wanted to make use of `TutorialQuery` in another script, we need only declare the following:

```
use tutorial.example.TutorialQuery;
```

In this manner can users simplify the construction of complex workflows, either by simplifying the use of certain standard PEs (by wrapping common configurations of complex PEs into simpler PEs with fewer inputs) or by encapsulating complex recurring tasks within a simple interface (by wrapping whole workflows into a single PE definition).

```
// Import PEs from the local registry.
use tutorial.example.TutorialQuery;

// Create instances of PEs.
TutorialQuery query   = new TutorialQuery;
Results       results = new Results;

// Construct workflow (no data source needed).
|-"SELECT * FROM littleblackbook WHERE id <= 10"-| => query.expression;
|-"10 entries from the little black book"-| => results.name;
query.data => results.input;

// Submit workflow.
submit results;
```

CHAPTER

THREE

CONSTRUCTING MORE SOPHISTICATED WORKFLOWS

So far we have only considered very simple workflows and very simple composite PEs which are merely wrappers for only slightly more complex existing PEs. If we want to construct more complex workflows however, we need to be able to scale component composition to arbitrary degrees, and be able to exercise more control over the selection of components based on circumstances at execution time. In order to do that, we need to be able to refer to certain variable factors at execution time. The declaration and assignment of values to variables falls into the domain of the first of Dispel's three type systems — the *language* type system. Using language types, we can direct the iterative construction of workflows and impose conditions on certain elements, as well as configure functions such as the constructor functions used to build custom PEs.

3.1 Dispel Language Types

The Dispel language type system validates the variables, constants and functions used within Dispel scripts. A variable is simply a vessel for some value. Every variable has a language type, and its existence must be declared before use:

```
Integer number;
```

In this case, a variable `number` of language type `Integer` is declared. A variable name must begin with an alphabetic character and contain only alphanumeric characters or underscores (_). By convention, variable names use camel-case. Variables must be assigned an initial value before they can be used; afterwards, variables can be assigned new values as often as desired:

```
Integer numberOfSources;

numberOfSources = 4;
numberOfSources = -1;
```

Typically, variables are assigned an initial value upon declaration, as so:

```
Integer number = 4;
```

Variables can only be assigned to literals or expressions of the correct language type. Variables of language type `Integer` can only be assigned to integer values or expressions which evaluate to integer values — if another variable is provided, then the former variable is assigned the value of the latter variable at the point of evaluation:

```
Integer number = 0;

number = 3 + (-4);
number = factorial(7);
```

```
Integer square = number * number;
```

Dispel recognises five basic language types; `Boolean`, `Integer`, `Real`, `String` and `Stream`. Each of these language types has its own valid literal type:

- `Boolean` variables can only be assigned to one of two values; `true` or `false`:

  ```
  Boolean statement1 = false, statement2 = true;
  ```

- `Integer` variables can be assigned any positive or negative integer value, as described above.
- `Real` variables can be assigned any decimal value:

  ```
  Real pi = 3.14, negative = -43.265;
  ```

- `String` variables can be assigned to any character string; character strings must be enclosed within double quotes:

  ```
  String text = "";
  ...
  text = "Hello World!";
  ```

  Special characters (such as tabs, carriage returns and double quote itself) are represented within strings by special escape characters preceded by a backslash (`\`). Longer strings can be split into segments, appended using the + operator:

  ```
  text = "This is the first line...\n" +
         "This is the second line with text in \"quotes\".";
  ```

- `Stream` variables can only be assigned to stream literals, as described in *Describing workflow input*. Stream literals are enclosed in stream delimiters (`|-` and `-|`) and can contain either a comma-separated list of values (which can be literals, expressions or variables of any of the above language types *except* `Stream` itself) or a stream expression such as `repeat`:

  ```
  Integer three = 3;
  Stream empty = |--|, list = |-1, "2", three-|;
  Stream repeating = |-repeat 11 of "Eleven"-|;
  ```

  As with strings, stream literal fragments can be appended together using the + operator:

  ```
  Stream fragment = |-2-|;
  Stream concat = |-1-| + fragment + |-3-|;
  ```

  Streams represent ordered sequences, with the left-most elements preceding elements to the right; concatenations of streams add to the end of the resulting stream.

Dispel also recognises an additional language type `Connection`, representing a connection interface. `Connection` is a 'null' type however, its variables never holding any value — instead connection interface 'variables' are simply handles for establishing connections between interfaces as well as streams and interface. As such, variables of type connection are simply declared and need never be assigned values, except to other connection interfaces:

```
Connection input;
Stream data = |-1, 2, 3, 4-|;
data => input;
Connection alias = input;
```

Arrays of variables can be created by first defining the type of the array's constituent elements and the size of the array, and then assigning values to each individual element of the array in turn as if it was a new variable of the relevant type:

```
Boolean[] array = new Boolean[3];
       array[0] = true;
       array[1] = false;
       array[2] = true;
```

Arbitrarily multi-dimensional arrays can be created by creating arrays of arrays:

```
Integer[][] matrix = new Integer[3][2];
      matrix[0][0] = 0;
      matrix[0][1] = 1;
      matrix[1][0] = 34;
      matrix[1][1] = -3;
      matrix[2][0] = -245;
      matrix[2][1] = 1111;
```

The length of an array `array` can be retrieved by referencing the `length` property of `array`:

```
Integer size = array.length;            // = 3
Integer outerSize = matrix.length;      // = 3
Integer innerSize = matrix[0].length;   // = 2
```

The length of an array is always an integer.

PEs are considered to be language types as well. As such, PE instances must be declared with a type (such as `SQLQuery`) and assigned a value — this will always be a new instantiation of the given PE type, made using the `new` directive:

```
SQLQuery query = new SQLQuery;
```

As already described in *Defining new types of processing element*, new types of PEs can be created using a `Type` declaration. There are in fact two ways to create new PE types. The first uses an internal connection signature:

```
Type SQLToTupleList is PE( <Connection expression> => <Connection data> );
```

This must then be followed by an invocation of a suitable PE constructor in order to provide an implementation of the abstract type, as shown in *Deriving new workflow elements*:

```
PE<SQLToTupleList> TutorialQuery = lockSQLDataSource("uk.org.UoE.dbA");
```

Another approach is to modify an existing PE type:

```
Type LockedSQLQuery is SQLQuery with initiator resource;
```

In this case, `SQLQuery` is modified such that its input interface `resource` is an `initiator`. Connection modifiers like `initiator` are used to specify restrictions on the use of connection interfaces, and can be used to refine how a given workflow element is implemented upon workflow submission — for now however, we shall defer further details.

3.2 Iterative workflow construction

With the ability to define variables comes the ability to define iterators. Assume that we want to create an array of parallel `SQLQuery` PE instances where each instance queries a different data source, but otherwise all instances perform the same query operation, sending the results further along the workflow. To start with, we need to initialise the array of `SQLQuery` instances and provide a connection interface from which to extract query expressions:

```
Connection input;
SQLQuery[] queries = new SQLQuery[numberOfSources];
```

We would also need an array of connection interfaces from which to acquire data source information and another array of connection interfaces to which to send the result (remember that we cannot simply connect multiple outputs to a single input, but will need to use a suitable PE to combine the outputs later):

```
Connection[] sources = new Connection[numberOfSources];
Connection[] outputs = new Connection[numberOfSources];
```

The problem then lies with how to properly instantiate the constituent elements of array `queries` and connect each PE instance to the correct inputs and output for arbitrary values of `numberOfSources`. We need an iterator.

Iterators are used to repeatedly execute a block of statements whilst a given condition holds, and as such can be used to succinctly describe repetitive workflow patterns. Dispel supports two standard iteration constructs; `while` and `for`.

The `while` construct is the simplest type of iterator. At each cycle of the iterator, a condition is evaluated, and the statement block within the loop is then executed only if that condition evaluates as `true`; otherwise, execution proceeds beyond the loop. For example:

```
Integer i = 0;
while (i < numberOfSources) {
    queries[i] = new SQLQuery;
    input => queries[i].expression;
    sources[i] => queries[i].source;
    queries[i].data => outputs[i];
    i++;
}
```

Naturally if the loop is to terminate, the body of the iterator must do something which will eventually cause the evaluation of the condition to fail — in the above example, the statement `i++` increments variable `i`, ensuring that eventually, `i` will equal `numberOfSources`.

Since many iterators rely on a single control variable which is updated regularly during each cycle of the loop however, there exists a variant of the `while` construct known as a `for` loop. Each `for` loop consists of an initialisation part (where the control variable is initialised), a conditional part (which determines when the loop should terminate), and an update part (where the control variable is updated). For example:

```
for (Integer i = 0; i < numberOfSources; i++) {
    queries[i] = new SQLQuery;
    input => queries[i].expression;
    sources[i] => queries[i].source;
    queries[i].data => outputs[i];
}
```

In the above example, a new instance of `SQLQuery` is created within array `queries` and connected to surrounding interfaces a number of times equal to `numberOfSources`. First control variable `i` is initialised, which is incremented at the end of every iteration (as directed by the statement `i++`) as long as the condition `i < numberOfSources` holds.

An interator is used in constructor `makeCorroboratedSQLQuery` below to implement abstract PE `MulticastQuery`:

```
package tutorial.example {
    use dispel.db.SQLQuery;
    use dispel.list.ListIntersect;

    // A PE type which queries multiple sources.
    Type MulticastQuery is PE( <Connection expression; Connection[] sources> =>
                               <Connection data> );

    // Use parallel SQLQuery instances to corroborate results.
    PE<MulticastQuery> makeCorroboratedSQLQuery(Integer size) {
        // Define aliases for workflow inputs in advance.
        Connection   expr;
        Connection[] srcs = new Connection[size];
        // Create instances of internal PEs.
        SQLQuery[]    queries   = new SQLQuery[size];
        ListIntersect intersect = new ListIntersect with inputs.length = size;

        // Connect SQLQuery instances in parallel.
        for (Integer i = 0; i < size; i++) {
            queries[i] = new SQLQuery;
            expr => queries[i].expression;
            srcs[i] => queries[i].resource;
            queries[i].data => intersect.inputs[i];
        }

        // Return intersection of query results.
        return PE( <Connection expression = expr; Connection[] sources = srcs> =>
                   <Connection data = intersect.output> );
    }

    register MulticastQuery, makeCorroboratedSQLQuery;
}
```

This particular implementation of `MulticastQuery` queries multiple data sources at once, and returns the intersection of results — in other words, it only returns results which can be corroborated by mutiple sources. It does this using the `for` loop described above and an instance of PE `ListIntersect` to combine the outputs of the constituent `SQLQuery` instances.

Note the instantiation of `ListIntersect` PE instance `intersect` in Lines 18–19 specifies the required size of the array of inputs which it should merge:

```
ListIntersect intersect = new ListIntersect
     with inputs.length = numberOfSources;
```

The use of `with` in this fashion can be used to specify configurable aspects of a PE instance's state and operation. We shall see other uses of `with` later in this tutorial. The above constructor can then be used to create a new composite PE as described here:

```
package tutorial.example {
    // Construct a 12-database corroborated query PE.
    PE<MulticastQuery> Corroborate = makeCorroboratedSQLQuery(12);

    // Create instances of PEs.
    Corroborate query   = new Corroborate;
    Results     results = new Results;

    // Construct workflow with 12 data sources.
    |- "Hello world!" -| => query.expression;
    for (Integer i = 0; i < 12; i++) |-"uk.org.UoE.db" + i-| => query.sources[i];
    |- "Results" -| => results.name;
    query.data => results.input;

    submit;
}
```

Note that for composite PEs, the size of input and output connection arrays is set at the point of PE construction.

3.3 Conditional workflow construction

In the previous section, we provided a constructor for abstract PE `MulticastQuery` which returned the intersection of results. Another reasonable implementation of `MulticastQuery` would be one which simply returned all results provided by all data sources. In question then is whether or not to remove duplicate results: if we simply want to see what results exist, we may prefer to remove duplicates; if we want to analyse how often certain results arise, we may prefer not to.

Let us assume that we have an iterator which provides an array of `SQLQuery` PE instances as described in the previous section. Assume also that we connect the outputs of these PE instances to an instance `merge` of PE `Combiner`, a PE which reads a list from each of its inputs and combines them into one output list in some fashion. What we want to do is modify the behaviour of `merge` based on whether or not we want to preserve duplicate results. We can do this by writing our PE constructor in such a way that it chooses between two different implementations of `Combiner` the value of a boolean variable passed to the constructor at execution-time:

```
Boolean unique;
```

If `unique` evaluates as `true`, then we want `merge` to be an instance of `ListUnion`, a PE which returns the union of the lists of inputs given to it; otherwise, we want `merge` to be an instance of `ListMerge`, which will combine all results without any concern for duplicates at all.

When statement blocks must be executed dependent upon certain conditions, we use the `if`/`else` or `switch`/`case` constructs. A basic `if` conditional executes a statement block only if a given expression evaluates as `true`:

```
if (unique) {
    merge = new ListUnion with inputs.length = size;
}
```

Alternatively, an `if`/`else` conditional will execute one statement block if the condition evaluates as `true`, and another if it evaluates as `false`:

```
if (removeDuplicates) {
    merge = new ListUnion with inputs.length = size;
} else {
    merge = new ListMerge with inputs.length = size;
}
```

An `if`/`else` conditional is used in constructor `makeMassSQLQuery` below. This constructor, which is also based on `MulticastQuery` like the previous script, works similarly to `makeCorroboratedSQLQuery`, except that it returns all results provided by all data sources:

```
package tutorial.example {
    use dispel.db.SQLQuery;
    use dispel.list.Combiner;
    use dispel.list.ListUnion;
    use dispel.list.ListMerge;
    // Import abstract type defined earlier.
    use tutorial.example.MulticastQuery;

    // Use parallel SQLQuery instances to collect results.
    PE<MulticastQuery> makeMassSQLQuery(Integer size, Boolean unique) {
        // Prepare components for connection.
        Connection   expr;
        Connection[] srcs    = new Connection[size];
        SQLQuery[]   queries = new SQLQuery[size];
        Combiner     merge;
        // Select combiner based on uniqueness condition.
        if (unique) merge = new ListUnion with inputs.length = size;
        else        merge = new ListMerge with inputs.length = size;

        // Connect SQLQuery instances in parallel.
        for (Integer i = 0; i < size; i++) {
            queries[i] = new SQLQuery;
            expr => queries[i].expression;
            srcs[i] => queries[i].resource;
            queries[i].data => merge.inputs[i];
        }

        // Return merger of query results.
        return PE( <Connection expression = expr; Connection[] sources = srcs> =>
                   <Connection data = merge.output> );
    }

    register makeMassSQLQuery;
}
```

In addition, this function takes `unique` as a parameter, a boolean variable which determines which implementation of `Combiner` is used within the composite PE in order to prune out duplicate results. The script below can be used to demonstrate `makeMassSQLQuery`'s use:

```
package tutorial.example {
    // Construct two different 12-database aggregated query PEs.
    PE<MulticastQuery> QueryList = makeMassSQLQuery(12, false);
    PE<MulticastQuery> QuerySet = makeMassSQLQuery(12, true);

    // Create instance sof PEs.
    QueryList query1 = new QueryList;
    Results results1 = new Results;
    QuerySet  query2 = new QuerySet;
    Results results2 = new Results;

    // Construct a workflow which permits duplicates.
    |- "Hello world!" -| => query1.expression;
    for (Integer i = 0; i < 12; i++) |-"uk.org.UoE.db" + i-| => query1.sources[i];
    |- "Results" -| => results1.name;
    query1.data => results1.input;
    // Construct a workflow which does not.
    |- "Goodbye world!" -| => query2.expression;
    for (Integer i = 0; i < 12; i++) |-"uk.org.UoE.db" + i-| => query2.sources[i];
    |- "Results" -| => results2.name;
    query2.data => results2.input;

    submit;
}
```

It is possible to nest multiple `if`/`else` conditionals:

```
if (dayOfTheWeek = "Monday") {
    colour = "gray";
} else if (dayOfTheWeek = "Tuesday") {
    colour = "yellow";
} else {
    colour = "green";
}
```

If the nesting of `if`/`else` conditionals becomes tedious, or when there are numerous choices for a given condition, the `switch`/`case` construct may be more useful:

```
switch (dayOfTheWeek) {
    case "Monday"    : colour = "gray";   break;
    case "Tuesday"   : colour = "yellow"; break;
    case "Wednesday" : colour = "red";    break;
    case "Thursday"  :
    case "Friday"    : colour = "blue";   break;
    default          : colour = "green";
}
```

We use the `break` keyword to exit from the `switch` construct — otherwise execution 'falls through' and executes all cases until the next `break` statement or until the end of the `switch` construct is reached (so in the above example case `"Thursday"` would execute `colour = "blue"`). The `default` keyword is used to mark the special case where none of the specified cases are satisfied.

The `break` keyword can be used to exit any statement block enclosed by braces (\{ and \}). Thus `break` can be used to exit an iterator. When iterators are nested, the `break` keyword will only break the inner-most iterator, leaving the outer iterators to execute as normal:

```
for (Integer i = 0; i < 100; i++) {
    for (Integer j = 0; j < 100; j++) {
        if (j == 50) { break; }
        ... // Statement block A.
    }
    ... // Statement block B.
}
```

In the above example, the statement block A will be executed five thousand times whilst statement block B will be executed only one hundred times. Similarly, the `continue` can be used within an iterator to jump to the next iteration without breaking out of the iterator entirely:

```
for (Integer k = 0; k < 100; k++) {
    ... // Statement block A.
    if (j < 50) { continue; }
    ... // Statement block B.
}
```

In the above example, statement block A will be executed one hundred times, whereas statement block B will only be executed fifty times.

CHAPTER

FOUR

MANIPULATING THE FLOW OF DATA

Thus far, we have constructed workflows with little concern as to the nature of the data being streamed between PE instances. However different PEs expect different inputs and produce output in accordance with their own specifications. Data may be consumed by inputs at different rates or require that data be consumed across all inputs synchronously, perhaps requiring data to be buffered whilst waiting for other parts of the workflow to catch up. Some PEs serve to push data through a workflow, others serve to pull data along. Many PEs are principally driven by one input, consuming data from other inputs only when needed — once the data-stream into that particular input is exhausted, any continuing input from other sources may be irrelevant. All of these factors play influence on data-intensive workflows.

In Dispel, many of these factors can be concealed from the casual user, the enactment platform hidden behind the gateway ensuring that data is adequately streamed and buffered across the length of an executing workflow, and ensuring that the data is of the correct form. The standard PEs provided by Dispelare designed to behave in the most intuitive fashion possible, so that most users will instinctively use them correctly.

Nonetheless, control over these factors can be exercised within Dispel. Expert users demand the ability to construct more sophisticated workflows, which require more careful attention to the flow of data; casual users rely on expert users to resolve data flow issues and conceal the detail behind the veil of composite PEs with simple interfaces. In this section we introduce the second of Dispel's type systems, the *structural* type system, which conerns itself with the logical structure of data streamed through connections. We also introduce connection modifiers, which can be used to describe how a PE instance consumes and produces data. Finally, we look at how to move between the language and structural type systems using stream literals.

4.1 Dispel structural types

Structural types are purely concerned with the data flowing through connections between and within PE instances. In that respect they differ from language types, which are principly used to guide the construction of workflows by providing variables which control iteration, selection and the behaviour of functions. Superficially, structural types are very similar to language types — we have structural types like `Integer` and `String` — but we also have arbitrarily complex structures (involving lists and tuples of structural types) and partial descriptions (wherein we permit lists of undefined types or only define some of the elements in a tuple).

Conceptually, we consider a data-stream as being a sequence of data elements, each holding a single 'unit' of data. However what that 'unit' constitutes can vary under different circumstances — in one context, we might expect each unit to be a single integer value, in another we might expect a list of tuples, each tuple containing multiple elements. Generally, the interfaces of PEs are annotated with information about the structure of data units streamed through them:

```
Type ConvertIntegerToReal is PE( <Connection:Integer input> =>
                                  <Connection:Real output> );
```

It is possible to omit structural information when defining new types of PE as we have in the past, in which case the structural type expected by each connection is considered to be `Any`, meaning the data can be in any form. We can get away with this for simple workflows, but for more complex workflows we need to be more careful. Consider a scenario in which the following new PE type is registered:

```
Type Normalise is PE( <Connection input> => <Connection output> );
```

What is the expected input and output of `Normalise`? A user might be able to infer something from the package in which `Normalise` is registered, but essentially the input and output structural types can only be determined by experimentation or searching for additional documentation. Annotating new PE types with structural type information makes them more self-documenting and also permits the gateway through which workflows are enacted to perform type verification and validation of a submitted workflow prior to execution.

More complex structural types can be constructed in a number of ways. *Lists* can be of any length, but each element must have the same abstract structure. A list is defined by enclosing the structure type of the elements within the list within square brackets as so:

```
Type IntListToRealList is PE( <Connection:[Integer] input> =>
                              <Connection:[Real] output> );
```

Note that connection interface type annotation only describes a single 'unit' of data carried within a data stream; it is already given that a description of the content of a stream over a set period of time is an ordered list of values. Thus if a stream outputs a simple sequence of real numbers one after another, then the structure type of the stream is `Real` *not* `[Real]`. On the other hand, if each element of the stream is itself a list of numbers (perhaps each of different length), then the structural type of the stream will be `[Real]` after all.

Conventional arrays also exist as structural types. Unlike lists, arrays can be multi-dimensional, but must be of fixed size. At the same time, we do not concern ourselves within Dispel with what that fixed size actually is. The reasoning behind this is that whilst it is important to know that the output of one PE is an array, or that another PE requires an array of certain dimensionality as input in order to validate a given workflow, we do not need to know the size of the array because we are always merely piping data from one processing element to another. An array is represented just as for language type arrays:

```
Type MatrixToVectorList is PE( <Connection:Real[][] input> =>
                               <Connection:[Real[]] output> );
```

In this case, `MatrixToVectorList` takes as input a two-dimensional matrix of real numbers and outputs a list of real arrays (demonstrating the combination of list and array structural types).

The other important structural type is that of a tuple. A tuple is an unordered collection of elements of different types. A tuple is enclosed in angle brackets, within which must be found a sequence of typed keys to which values can be assigned:

```
Type GridLocToAngleLoc is
    PE( <Connection:<Integer x, y; String name> gridLoc> =>
        <Connection:<Real angle, magnitude; String label> angleLoc> );
```

Variable names must correspond to the keys used by processing elements themselves to identify the parts of the tuple; as demonstrated above, multiple keys of the same type can be defined together (so one can write `Integer x, y;` instead of `Integer x; Integer y;`).

Complex structural types can be given aliases in the same manner as PE types using an `Stype` declaration:

```
Stype GridLoc  is <Integer x, y; String name>;
Stype AngleLoc is <Real angle, magnitude; String label>;

Type GridLocToAngleLoc is PE( <Connection:GridLoc gridLoc> =>
                              <Connection:AngleLoc angleLoc> );
```

What if we do not know (or care) about some or all of the structure of elements passing through a data stream however? In that case we have at our disposal the `Any` generic type and the `rest` identifier. An element of structural type `Any` can take any shape or form, and can be used anywhere, including within arrays, lists and tuples. Meanwhile, the `rest` identifier is used within tuples to encapsulate all tuple elements not referenced prior. To illustrate:

```
Stype GridLoc3D  is <Integer x, y, z; String name>;
Stype AbsGridLoc is <Integer x, y, z; Any name>;
Stype GridLocXZ  is <Integer x, z; rest>;
```

In the above example, each statement matches those prior to it (though not those after). Note that `rest` must be the last referenced element in a tuple, but can subsume any of the elements within that tuple (for example, it can happily subsume element `y` without subsuming element `z`).

Streams also have a structural type, determined by its content. This type will always be the least common sub-type of all the data elements described within the stream:

```
Stream first  = |-1, 2, 3, 4-|;
Stream second = |-"one", "two", "three", "four"-|;
Stream third  = |-1, "two", 3, 4.0-|;
Stream fourth = |-<key = 11; value = "eleven">,
                  <key = 12; value = "twelve"; note = "2 * 6">-|;
```

In the above example, stream `first` has structural type `Integer`, whilst `second` has type `String`. Stream `third` has structural type `Any`, whilst `fourth` has type `<Integer key; String value; rest>`.

Earlier in *Defining new types of processing element* we defined a PE type `SQLToTupleList`. We can now annotate that type with the correct structural types:

```
Type SQLToTupleList is PE( <Connection:String expression> =>
                           <Connection:[<rest>] data> );
```

The above declaration states that for each string of text representing an SQL query, a PE of type `SQLToTupleList` produces a list of tuples describing a response to that query (though we do not concern ourselves with the content of those tuples). As a consequence, when we define PE `TutorialQuery` using `lockSQLDataSource` and then register it, the gateway knows that `TutorialQuery` accepts only inputs of type `String` and outputs a sequence of tuple lists. This means that when validating a submitted Dispel workflow, the gateway can verify that any instance of `TutorialQuery` is being fed structurally correct input and output.

It is also possible to define (or re-define) the structural type of specific instances of PEs:

```
TutorialQuery query = new TutorialQuery
    with data as data:[<Integer key; String value>];
```

This can be useful if the user knows that the data being streamed into or out of a PE instance is limited to a particular subset of the input or output normally permitted, and that the PE instance is to be connected to other PE instances which *only* permit data limited to that particular subset:

```
Type DictionaryKeySort is PE( <Connection:[<Integer key; String value>] unsorted> =>
                              <Connection:[<Integer key; String value>] sorted> );

DictionaryKeySort sort = new DictionaryKeySort;
query.data => sort.unsorted;
```

Ordinarily, this would be considered unsafe by the enactment gateway, being that superficially `TutorialQuery` instances can produce data which cannot be consumed by instances of `DictionaryKeySort`. Of course this approach only works if the data being produced really is limited to the specified structural type – otherwise, the gateway will raise an error or filter elements in accordance with its configuration.

What happens if the data produced by one PE instance is incompatible with the expected input of another PE instance to which it has been connected? For example, if the output of one PE is of structural type `Integer` whilst the expected input of another PE is of type `Real`:

```
Type Thermometer is PE( <> => <Connection:Integer measurement> );
Type SteamEngine is PE( <Connection:Real temperature> =>
                        <Connection:Real acceleration> );

Thermometer thermo = new Thermometer;
SteamEngine engine = new SteamEngine;
thermomeasurement => enginetemperature;
```

Implemented as is, `engine` cannot consume the data given. In this case, we need a *converter*. A converter is a PE, generally with one input and one output, which transforms data from one form into another without actually changing its semantic content — for example, PE `ConvertIntegerToReal` defined at the beginning of this section. In Dispel, we can explicitly introduce a converter into a workflow as just another PE, or we can rely on the gateway to which we submit our workflow to insert any necessary converters for us:

```
Thermometer thermo = new Thermometer;
SteamEngine engine = new SteamEngine;
ConvertIntegerToReal convert = new ConvertIntegerToReal;
thermo.measurement => convert.input;
convert.output => engine.temperature;
```

The ability of a gateway to perform automatic type conversions is useful, but not unlimited. In particular, the ability of a gateway to to decompose complex structural types and perform conversions depends on there being a common abstract structure apparent in both the original and target structural type descriptions — one would not expect every possible output of `TutorialQuery` to be convertable into structural type `[<Integer key; String value>]` without user knowledge of the possible results which can be obtained from a given set of queries. In general, it is best not to rely too heavily on the intelligence of any given gateway without good cause, particularly within Dispel scripts which could be submitted to different gateways of possibly different type conversion capability.

4.2 Type coercions

To recap, the language type system is concerned with control flow within Dispel scripts, whilst the structural type system is concerned with data flow in workflows. It is vitally important that there is no confusion between the two type systems — even in circumstances where language types and structural types appear identical (as for `Integer` for example), there may be implementational differences between the types used by the Dispelparser for validating and executing scripts, and the types used within the actual implementation of the workflow described by a Dispel script.

If we want to step between the realm of language types and structural types, then we need the means to convert data between type systems. To this end, we have at our disposal both *implicit* and *explicit* type coercion. Implicit type coercion occurs between primitive language and structural types, and occurs within stream literals. For example, when we write:

```
Integer x = 3, y = 64, z = -4;

|-x, y, z-| => counterinput;
```

What we are doing is making a conversion of variables `x`, `y` and `z` from the `Integer` language type to the `Integer` structural type. This type of coercion also occurs whenever variables defined within the Dispel script are wrapped within tuples, arrays or lists and inserted into stream literals. Note that stream literals are the only way to directly inject information from a Dispel script into the data stream described by a PE workflow.

The implicit type coercion described above is adequate for quickly constructing simple data streams to inject into a workflow, but is less suitable for more complex or dynamically constructed streams. For example, how do we feed an array of strings into a stream as a sequence of elements? Assume that we have the stated the following:

```
TutorialQuery query = new TutorialQuery;
String[] queries = new String[3];
  queries[0] = "SELECT name FROM littleblackbook";
  queries[1] = "SELECT address FROM littleblackbook";
  queries[2] = "SELECT number FROM littleblackbook";
```

How can we inject queries into query? We can insert it bodily into a data stream, but then we have a single element containing an array of strings, which we now know is the incorrect structure for data going into an instance of TutorialQuery, being of type SQLToTupleList:

```
|-queries-|:String[] => query.expression; // Incorrect structure.
```

We can inject it manually element by element, but this requires that we know the size of the array, and is particularly tedious for large arrays:

```
|-queries[0], queries[1], queries[2]-|:String => query.expression;
```

A more promising approach is to use a loop to build the stream based on the length of the array at the time of computation. We do not want to have to define the loop anew every time however, so we use a *stream function*. Stream functions are simply functions which return elements of type Stream, which can then be connected directly to a connection interface or concatenated with other streams. For example, to feed an array element by element into a stream, we only need a function like stringArrayToStream:

```
Stream stringArrayToStream(String[] array) {
    Stream stream = |--|;
    for (Integer i = 0; i < arraylength; i++) {
      stream += array[i];
    }
    return stream;
}
```

Note that we are still using implicit type coercion to add elements of array to stream. Having registered this function (or in this case, imported it from package dispel.stream), we can then invoke it when injecting the array with the desired content into our workflow:

```
stringArrayToStream(queries) => queryexpression;
```

Stream functions are thus the preferred means by which to move between the language and structural type systems for constructed types.

4.3 Connection modifiers

Connection modifiers are a class of modifier attached to connection interfaces which describe how that interface produces / consumes data and how it does that in relation to other interfaces within the same PE. For example, the initiator modifier, when applied to an input interface, asserts that the modified interface consumes data before all other input interfaces within a given PE; the other interfaces will only begin to consume data once the initiator terminates:

```
Type LockedSQLQuery is
    PE( <Connection:String expression; Connection:String initiator source> =>
        <Connection:[<rest>] data> );
```

In this case, an instance of LockedSQLQuery handles queries in a similar fashion to an instance of SQLQuery; however unlike SQLQuery, this PE only reads the data source input once, prior to any queries, and returns results only from that data source. As alluded to earlier, there is a simpler way to define PEs which are basically minor modifications on existing ones:

```
Type LockedSQLQuery is SQLQuery with initiator source;
```

If we are only concerned with a one-off use of a modified PE, we can apply connection modifiers upon instantiation:

```
SQLQuery query = new SQLQuery with initiator source;
```

Connection modifiers are both descriptive and perscriptive; they inform users about the behaviour of certain PEs by their type descriptions, and they permit users to make ad-hoc modifications of existing PEs. Consider the two PEs `Combiner` and `SynchronisedCombiner`:

```
Type Combiner is PE( <Connection[] inputs> => <Connection output> );
Type SynchronisedCombiner is Combiner with roundrobin inputs;
```

The basic behaviour of `Combiner` is simply to unite its input streams into one stream — no commitment is made as to how data from each input is ordered in the output stream. `SynchronisedCombiner` however does make such a commitment — the effect of the `roundrobin` modifier, when applied to an array of input connection interfaces, is to assert that each interface in the array will consume a single data element (as defined by the highest-level abstract structure of the data given) in order, further consumption restricted into a full cycle has been performed. Thus, we know that the output of an instance of `SynchronisedCombiner` will consist of all first data elements of each input in order, followed by all second elements in order, and so on until all inputs terminate.

The ability of users to apply *ad-hoc* modifications to existing PEs is limited to the ability of the enactment platform to implement the modified PE; in most cases however, this is a simple matter, since most modifications in Dispel can be implemented over existing code via interim interfaces. The full list of available connection modifiers can be found in the Dispel reference manual.

4.4 Dispel domain types

The final type system used in Dispel is the domain type system. As with structural types, domain type information is appended onto connection interface declarations — in this case using double-colons (`::`). Unlike structural types however, domain types describe not the structure of streamed data, but what a given data flow *is* from the perspective of a domain expert. For example:

```
Type SQLToTupleList is PE( <Connection:String::"db:SQLQuery" expression> =>
                           <Connection:[<rest>]::"db:TupleRowSet" data> );
```

From this we can infer that the text strings consumed by interface `expression` represent queries in the SQL language. We can also infer that the lists of tuples produced by interface `data` are sets of entries (rows) drawn from a database. Domain types permit ontological information to be embedded into Dispel workflows, which can be used for mapping equivalences between ontological terms, or simply to assist the user in understanding how to use a given PE type correctly.

Domain type descriptors are enclosed in quotation marks and prefixed with a *namespace* identifier. The namespace identifier serves as an alias for the ontology from which a domain type is drawn. Namespace identifiers must be introduced within a Dispelscript using a `namespace` declaration:

```
namespace db "http://www.admire-project.eu/ontology/db#";
```

In this case, the namespace `db` is defined, drawing upon an ontology for databases (`db` is in fact one of the core Dispel namespaces used internally for PE types, along with `dispel`, which need not be explicitly defined in scripts).

Like with structural types, domain type mismatches can be dealt with using converters, albeit in this case ones that often maintain the same structure of data flowing into and out of the converter, but rescale the data based on the percieved and required domain types:

```
Type CelsiusToKelvin is PE( <Connection:Real::"measure:Celsius" celsius> =>
                            <Connection:Real::"measure:Kelvin"> );
```

For the `CelsiusToKelvin` converter, both input and output are of structural type `Real`, but all celsius values going in are offset to match the equivalent kelvin value. Again, implicit type conversions performed automatically by the gateway to which a Dispel script is submitted may be available, but should not be relied upon too heavily.

CHAPTER

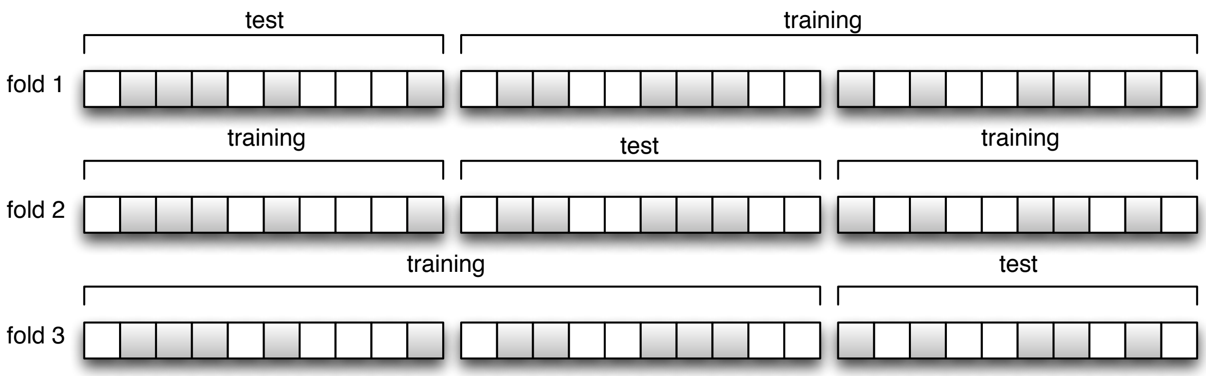
FIVE

CASE STUDIES

In this section we examine two different scenarios within which Dispel can be used, demonstrating many of the language's features.

5.1 *k*-fold cross validation

In statistical data mining and machine learning, *k-fold cross validation* is an abstract pattern used to estimate the classification accuracy of a learning algorithm. It determines that accuracy by repeatedly dividing a data sample into disjoint training and test sets, each time training and testing a classification model constructed using the given algorithm before collating and averaging the results. By training and testing a classifier using the same data sample divided differently each time, it is hoped that a learning algorithm can be evaluated without being influenced by biases that may occur in any particular partitioning of training and test data.



3-fold cross validation on a data sample of 30 elements.

The basic structure of a *k*-fold validation workflow pattern is very simple. First, a random permutation of the sample data set is generated, which is then partitioned into *k* disjoint subsets. A classifier is trained using the given learning algorithm and then tested *k* times. In each training and testing iteration, referred to as a *fold*, all sample data excluding that of one subset, different each time, is used for training; the excluded subset is then used to test the resulting classifier. This entire process can be repeated with different random permutations of the sample data. The figure above illustrates how a data sample might be divided for a *3*-fold cross validation process.

To specify a *k*-fold cross validation workflow pattern which can be reused for different learning algorithms and values of *k*, it is best to specify a PE function which can take all necessary parameters and return a bespoke cross validator PE on demand. Such a PE can then be instantiated and fed a suitable data corpus, producing a measure of the average accuracy of classification:

```
package dispel.datamine {
    use dispel.core.DataPartition;
    use dispel.list.ListMerge;

    // Produces a k-fold cross validation workflow pattern.
    PE<DataValidator> makeCrossValidator(Integer k, PE<TrainClassifier> Train,
                                         PE<ApplyClassifier> Classify, PE<ModelEvaluate> Evaluate) {
        Connection input;
        // Data must be partitioned and re-combined for each fold.
        PE<DataPartitioner> FoldData = makeDataFold(k);
        FoldData  fold  = new FoldData;
        ListMerge union = new ListMerge with inputs.length = k;

        // For each fold, train a classifier then evaluate it.
        input => fold.data;
        for (Integer i = 0; i < k; i++) {
            Train    train    = new Train;
            Classify classify = new Classify;
            Evaluate evaluate = new Evaluate;

            fold.training[i] => train.data;
            train.classifier => classify.classifier;
            fold.test[i] => classify.data;
            classify.result => evaluate.predicted;
            fold.test[i] => evaluate.expected;
            evaluate.score => union.inputs[i];
        }

        // Return cross validation pattern.
        return PE( <Connection data = input> => <Connection results = union.output> );
    }

    register makeCrossValidator;
}
```

The above Dispel request defines a PE function `makeCrossValidator`. This function describes an implementation of `Validator`, an abstract PE which given a suitable dataset, should produce a list of results from which (for example) an average score and standard deviation can be derived:

```
Type Validator is PE( <Connection:[<rest>]::["kdd:Observation"] data> =>
                      <Connection:[Real]::["kdd:Score"] results> );
```

The function `makeCrossValidator` requires four parameters:

- `Integer k` specifies the number of subsets into which the sample data should be split and the number of training iterations required — in other words, `k` is the *k* in *k*-fold.
- `PE<TrainClassifier> Trainer` is a PE type, instances of which can be used to train classifiers — it must encapsulate the learning algorithm to be tested and be compatible with the `TrainClassifier` PE.
- `PE<DataClassifier> Classifier` is a PE type, instances of which take test data and a classifier model and produce a prediction. Any classifier must be a implementable version of the `DataClassifier` PE.
- `PE<ModelEvaluator> Evaluator` is a PE type, instances of which take observation data and accompanying predictions, and assigns a score based on the accuracy of those predications. Must be an implementation version of the `ModelEvaluator` PE.

In this case there are three instances where a PE type is passed into a PE function in order that it be able to create an arbitrary number of instances of that PE within its internal workflow. Each must be compatible with (have internal connection signatures subsumed by) a given (possibly abstract) PE.

```
Type TrainClassifier is PE( <Connection:[<rest>]::["kdd:Observation"] data> =>
                            <Connection:Any::"kdd:Classifier" classifier> );
```

`TrainClassifier` consumes a body of training data in the form of a list of tuples and produces a classification model. Any PE implementation of `TrainClassifier` must encapsulate a learning algorithm and must know how to interpret the data provided — this includes knowing which feature a classifier is to be trained to predict. Thus any such PE would probably be a bespoke construction generated by a function immediately prior to the creation of the cross validator.

```
Type ApplyClassifier is PE( <Connection:[<rest>]::["kdd:Observation"] data;
                             Connection:Any::"kdd:Classifier" classifier> =>
                            <Connection:[<rest>]::["kdd:Prediction"] result> );
```

`Classifier` consumes a body of data in the form of a list of tuples and a classification model, producing a list of tuples describing classification results.

```
Type ModelEvaluator is PE( <Connection:[<rest>]::["kdd:Observation"] expected;
                            Connection:[<rest>]::["kdd:Prediction"] predicted> =>
                           <Connection:[Real]::["kdd:Score"] score> );
```

`ModelEvaluator` consumes a body of observations (test data) alongside an accompanying body of predictions (classifications), producing a score between zero and one rating the accuracy of classification.

The workflow described by function `makeCrossValidator` begins by splitting its input using a `FoldData` PE, created using sub-function `makeDataFold`. A *k*-fold cross validator must partition its input data into *k* subsets, and construct training and test data sets from those subsets; `makeDataFold` is defined below:

```
package dispel.datamine {
    use dispel.core.RandomListSplit;
    use dispel.list.ListMerge;

    // Produces a PE capable of splitting data for k-fold cross validation.
    PE<DataPartitioner> makeDataFold(Integer k) {
        Connection   input;
        Connection[] trainingData = new Connection[k];
        Connection[] testData     = new Connection[k];
        // Create instance of PEs for randomly splitting and recombining data.
        RandomListSplit sample = new RandomListSplit with results.length = k;
        ListMerge[]     union  = new ListMerge[k];

        // After partitioning data, form training and test sets.
        input => sample.input;
        for (Integer i = 0; i < k; i++) {
            union[i] = new TupleUnionAll with inputs.length = k - 1;
            for (Integer j = 0; j < i; j++) sample.outputs[j] => union[i].inputs[j];
            sample.outputs[i] => testData[i];
            for (Integer j = i + 1; j < k; j++) sample.outputs[j] => union[i].inputs[j - 1];
            union[i].output => trainingData[i];
        }

        // Return data folding pattern.
        return PE( <Connection data = input> =>
                   <Connection[] training = trainingData; Connection[] test = testData> );
    }

    register makeDataFold;
}
```

This function describes an implementation of the abstract PE `DataPartitioner` which given a suitable dataset, should produce an array of training data sets and an array of test data sets:

```
Type DataPartitioner is PE( <Connection:[<rest>]::["kdd:Observation"] data> =>
                            <Connection[]:[<rest>]::["kdd:Observation"] training;
                             Connection[]:[<rest>]::["kdd:Observation"] test> );
```

The function `makeDataFold` requires just one parameter `count`, specifying the number of folds of the input data to create. The function itself uses two existing PE types: `RandomListSplit`, which randomly splits its input into a number of equal subsets (or as close to equal as can be managed); and `TupleUnionAll`, which combines its inputs (each carrying lists of tuples) into a single tuple list. In the workflow described by the function, an instance of `RandomListSplit` is used to partition all incoming data, and each partition is placed into all but one training set (different for each partition) using an instance of `TupleUnionAll`; each partition is also taken as its own test dataset. All training datasets and test datasets are then sent out of the workflow.

Using `makeDataFold`, the function `makeCrossValidator` can construct a PE which will prepare training and test data for cross validation. For each 'fold' of the cross validation workflow pattern, one training set is used to train a classifier via an instance of the provided `Trainer` PE. This classifier is then passed on to an instance of the provided `Classifier` PE, which uses it to make predictions on the test dataset corresponding to the training set (that is, the single partition of the original input data *not* used for training). Finally, the generated predictions are sent along with that same test data to an instance of the provided `Evaluator` PE, which assigns a score to the classifier based on the accuracy of its predictions.

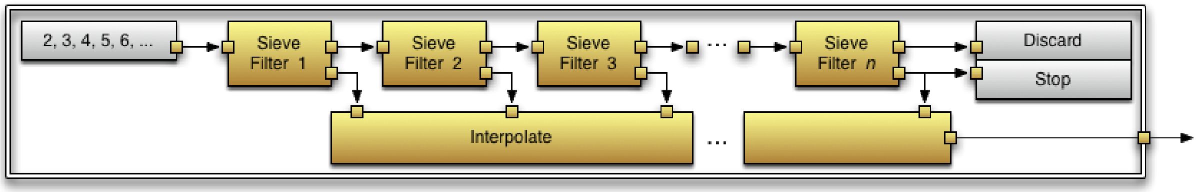
The scores for every fold of the workflow pattern are then combined using an instance of `ListBuilder`, an existing PE which constructs an (unordered) list from its inputs.

```
package dispel.datamine {
    // Import test PEs.
    use dispel.tutorial.DataProducer;
    use dispel.tutorial.TrainingAlgorithmA;
    use dispel.tutorial.BasicClassifier;
    use dispel.tutorial.MeanEvaluator;

    // Create a cross validator PE.
    PE<Validator> CrossValidator
        = makeCrossValidator(12, TrainingAlgorithmA, BasicClassifier, MeanEvaluator);
    // Make instances of PEs for workflows.
    DataProducer   producer  = new DataProducer;
    CrossValidator validator = new CrossValidator;
    Results        results   = new Results;

    // Connect workflow.
    |- "uk.org.UoE.data.corpus11" -| => producer.source;
    producer.data => validator.data;
    validator.results => results.input;
    |- "Classifier Scores" -| => results.name;

    submit;
}
```

The above Dispel request demonstates the *k*-fold cross validation in use. PEs compatible with `TrainClassifier`, `DataClassifier` and `ModelEvaluator` are provided which are then used to implement a new PE `CrossValidator`. This PE can then simply be connected to a suitable data source (in this case, an instance of `DataProducer`), and a place to put its results (in this case, simply an instance of `Results` — however one can imagine a PE which takes input from several cross validators, each testing a different learning algorithm, which then maps the average result for each algorithm in a graph with standard deviations noted).

5.2 The Sieve of Eratosthenes

The *Sieve of Eratosthenes* is a simple algorithm for finding prime numbers. The algorithm works by counting natural numbers and filtering out numbers which are composite (non-prime). We start with the integer 2 and discard every integer greater than 2 that is divisible by 2. Then, we take the smallest of all the remaining integers, which is definitely a prime, and discard every integer greater than that prime (in this case 3). We continue this process with the next integer and so on, until the desired number of primes have been discovered.



The internal composition of the Sieve of Eratosthenes.

The Sieve of Eratosthenes, whilst ultimately a toy application, serves as a useful device to demonstrate such Dispel concepts as connection modifiers, stream comprehensions and cascading termination. The Sieve can be implemented by a pipeline pattern described by a PE constructor. Using such a constructor, it is possible to implement the pipeline for arbitrary numbers of primes. This pipeline pattern will take the form shown above.

The principal component of a Sieve of Eratosthenes is the filtering element used to determine whether or not a given integer is divisible by the last encountered prime. We define an abstract filter as so:

```
Type AbstractFilter is
    PE( Stype Element is Any;
        <Connection:Element input> =>
        <Connection:Element filtered; Connection:Element unfiltered> );
```

A filter is expected to take a stream of inputs and split it into two streams (the 'filtered' stream and the remainder). A filter is not supposed to be transformative, so the output streams should be of the same type as the input stream. Note that an implementation of `AbstractFilter` is not actually expected to discard either filtered or unfiltered elements, since it cannot be predicted which elements will be of most interest in a particular use-case; if a workflow designer has no use of a given output, that output can be redirected to `discard`.

Our filtering element must filter the first integer encountered on its input stream as a prime (assuming the correct construction of the sieve as a whole), discard all successive integers divisible by that prime, and pass onto the next filter all remaining input. We can use a `ProgrammableIntegerFilter` to do the heavy lifting:

```
Type ProgrammableIntegerFilter is
      PE( <Connection:Integer terminator input;
           Connection:String initiator expression;
           Connection[]:Integer lockstep parameters> =>
          <Connection:Integer filtered; Connection:Integer unfiltered> );
```

We need only specify the filtering behaviour via input `expression` (filter all *x* where *x* is divisible by some set integer *y*) and bind any free variables within the filter specification via `parameters` (in this case *y*, provided by the first integer to enter our filter). To split off the first input, we use `HeadFilter`:

```
Type HeadFilter is AbstractFilter with filtered as head, unfiltered as tail,
  @description = "Diverts the head of a stream.";
```

We can then create a constructor `makeSieveFilter` as defined below; because the constructor has no variable parameters, we immediately construct `SieveFilter` and export it:

```
package tutorial.example {
    use dispel.filter.AbstractFilter;
    use dispel.filter.HeadFilter;
    use dispel.filter.ProgrammableIntegerFilter;

    // Define sieve element constructor.
    PE<AbstractFilter> makeSieveFilter() {
        // Create reference to input connection.
        Connection:Integer input;
        // Instantiate internal components.
        HeadFilter split = new HeadFilter;
        ProgrammableIntegerFilter divide = new ProgrammableIntegerFilter
            with parameters.length = 1;

        // Construct internal workflow.
        |-"x if (x % $0) == 0"-| => divide.expression;
        input => split.input;
        split.head => divide.parameter[0];
        split.tail => divide.input;
        divide.unfiltered => discard;

        // Output first integer received and all indivisible integers.
        return PE( <Connection input = input> =>
                   <Connection filtered = split.head;
                    Connection unfiltered = divide.filtered> );
    }

    // Create the sieve element PE.
    PE<AbstractFilter> SieveFilter = makeSieveFilter();

    register SieveFilter;
}
```

We can now define a constructor for the Sieve. The Sieve consists of an array of filters, which sequentially redirect primes to an `Interpolate` PE:

```
Type Interpolate is PE( Stype Element is Any;
                        <Connection[]:Element inputs> =>
                        <Connection:Element output> );
```

In order to ensure that the primes are output in order of discovery, we modify the interpolator's inputs to be `roundrobin`. We connect each filter's unfiltered output (being the sequence of numbers not divisible by the first prime encountered) to the next filter, except for the last, which discards all such values (having found the last prime of interest). We use a stream comprehension to generate all integers in sequence from 2 onwards indefinitely, and connect that to the first filter:

```
package tutorial.example {
    use dispel.core.Interpolate;
    use tutorial.example.SieveFilter;
    use dispel.math.PrimeGenerator;

    // Define sieve constructor.
    PE<PrimeGenerator> makeSieveOfEratosthenes(Integer count) {
        // Instantiate internal components.
        SieveFilter filter      = new SieveFilter[count];
        Interpolate interpolate = new Interpolate
            with roundrobin inputs, input.length = count;
        // Initialise sieve elements.
        for (Integer i = 0; i < count - 1; i++)
            filter[i] = new SieveFilter with terminator output;
        filter[count - 1] = new SieveFilter with terminator prime;

        // Construct internal workflow.
        |-x for x in 2..-| => filter[0].input;
        for (Integer i = 0; i < count - 1; i++) {
            filter[i].unfiltered => filter[i + 1].input;
            filter[i].filtered   => interpolate.inputs[i];
        }
        filter[count - 1].unfiltered => discard;
        filter[count - 1].filtered   => interpolate.input[count - 1];
        filter[count - 1].filtered   => stop;

        // Return all primes generated.
        return PE( <> => <Connection primes = interpolate.output> );
    }

    register makeSieveOfEratosthenes;
}
```

That is enough to implement the Sieve; however we also want the Sieve to shutdown once all required primes have been found rather than pour integers into the ether indefinitely. So we specify each filter's unfiltered output steam as `terminator`, except for the last (which is discarded) — we instead create a connection from that filter's prime output to `stop` and declare that as being `terminator`. The effect of this is to create a backwards termination cascade once the last prime is generated, which will cascade back through all filters and end the infinite integer stream as well as close the interpolator — thus ensuring the efficient shutdown of the entire Sieve.

The Sieve of Eratosthenes for one hundred prime numbers can now be executed as shown below:

```
package tutorial.example {
    // Import abstract PE type and constructor.
    use dispel.math.PrimeGenerator;

    // Construct the sieve.
    PE <PrimeGenerator> SoE100 = makeSieveOfEratosthenes(100);
    SoE100  sieve100 = new SoE100;
    Results results  = new Results;

    // Construct the top-level workflow.
    |-"100 prime numbers"-| => results.name;
    sieve100.primes         => results.input;

    // Submit the workflow.
    submit;
}
```