Scribble

User Guide

by Kohei Honda and Gary Brown

Installation

This is the installation guide for the Scribble tools. Scribble is a notation for describing interaction based protocols between multiple parties.

1.1. Pre-Requisites

The pre-requisites for the Scribble design time tools are:

1. Java
   The design time tools are Java based, so you will need a suitable JVM (Java Virtual Machine) to run the tools. If you also intend to generate Java APIs, for use at runtime, then you will need a JDK (Java Development Kit).

   Download the version 1.6 (or higher) from the

   > **Note**
   >
   > SUN WEBSITE

   . Once downloaded, follow the instructions to install the JVM or JDK on your system.

2. Eclipse
   The Scribble protocol descriptions can be edited using a standard text editor, and the Scribble tools can be invoked using command line tools (as described in the following chapter).

   However it is also possible to use the Scribble tools from within the Eclipse IDE environment, by installing the Scribble tools as plugins. If you wish to use this approach, then you will need a version of Eclipse (3.5 or higher) which can be downloaded from the Eclipse website.

1.2. Installing the Command Line Version

To install the command line version of the Scribble tools:

1. Download the Scribble tools from .....

2. Unpack the tool distribution in a suitable location

3. Setup environment
   The commands can be executed from the bin folder of the Scribble tools distribution. Alternatively, the `bin` folder can be added to the execution path, to enable the commands to be performed from any folder.

   For example, on Linux running bash, simply edit the `.bash_profile` file within your home directory to add:

```
PATH=$PATH:${path-to-scribble}/bin
```

1.3. Installing the Eclipse Version

Note

This version is not currently available. It will require an update site to be setup and the automated build of the update site artefacts.

Command Line Tools

This section describes how to use the command line tools that are available in the `bin` folder of the Scribble protocol tools distribution.

Information on the Scribble protocol notation (or language) can be found in the *Scribble Protocol Guide*.

2.1. Parsing a Protocol description

The *parse* command takes a single parameter, which is the path to the file containing the protocol description to be parsed.

For example, if the user is in the top level folder of the Scribble tools distribution, without the `bin` folder being added to the system path, then the following command can be executed to parse one of the sample protocol descriptions:

```
bin/parse.sh samples/scribble/Purcahsing.spr
```

If the supplied file path is not valid, then the command will report an error.

This command will read the protocol description, as shown below, and convert it into an internal object model representation.

```
namespace samples.notation;

import samples.notation.Order;
import samples.notation.CreditCheck;
import samples.notation.CreditOk;
import samples.notation.InsufficientCredit;
import samples.notation.Confirmation;
import samples.notation.OutOfStock;

protocol Purchasing {
 participant Buyer, Broker, CreditAgency, Seller;

 Order from Buyer to Broker;

 CreditCheck from Broker to CreditAgency;

 choice from CreditAgency to Broker {
  when CreditOk {
   Order from Broker to Seller;

   choice from Seller to Broker {
    when Confirmation {
     Confirmation from Broker to Buyer;
    }
    when OutOfStock {
     OutOfStock from Broker to Buyer;
    }
   }
  }
  when InsufficientCredit {
   InsufficientCredit from Broker to Buyer;
  }
 }
}
```

If any errors are detected in the syntax of the parsed protocol description, then these will be reported to the command window. For example, if you edit the supplied file, and change the keyword *participant* to append an 'X', then the following error would be produced:

```
ERROR: [line 11] no viable alternative at input 'participantX'
```

2.2. Validating a Protocol description

The *validate* command takes a single parameter, which is the path to the file containing the protocol description to be validated.

For example, if the user is in the top level folder of the Scribble tools distribution, without the `bin` folder being added to the system path, then the following command can be executed to validate one of the sample protocol descriptions:

```
   bin/validate.sh samples/scribble/Purcahsing.spr
```

When this command is performed initially, it will complete without any errors. However if you edit the `samples/scribble/Purchase.spr` file, and change the following line:

```
   CreditCheck from Broker to CreditAgency;
```

For example, change the *Broker* participant to *Broker2*, and then re-run the *validate* command. This will result in the following error messages:

```
ERROR: Unknown participant 'Broker2'
```

```
ERROR: Activity is not connected with preceding activities' participants
```

2.3. Checking Conformance a Protocol description

The *conforms* command takes two parameters, which are both paths to a file containing a protocol description. The first parameter is the protocol description to be checked for conformance against the second parameter's protocol description. So the second parameter is the *reference* protocol description.

For example, if the user is in the top level folder of the Scribble tools distribution, without the `bin` folder being added to the system path, then the following command can be executed to check one of the sample protocol descriptions as being conformant with another reference protocol description:

```
  bin/conforms.sh samples/scribble/OrderProcess.spr samples/scribble/ReferenceOrderProcess.spr
```

If you inspect the two process definitions, you will find one difference. The first protocol definition has the following interaction:

```
 MyOrder from Buyer to Seller;
```

The second, reference protocol description, has the following interaction:

```
 Order from Buyer to Seller;
```

This results in the following conformance error message:

```
ERROR: Type mismatch with referenced description, was expecting 'Order'
```

2.4. Projecting a Protocol description

The *project* command takes two parameters. The first parameter is the protocol description to be projected and the second parameter is the *participant*.

For example, if the user is in the top level folder of the Scribble tools distribution, without the `bin` folder being added to the system path, then the following command can be executed to project one of the sample protocol descriptions:

```
   bin/project.sh samples/scribble/Purchasing.spr Seller
```

This results in the following located Protocol being displayed on the console:

```
namespace samples.notation;
import samples.notation.Order;
import samples.notation.CreditCheck;
import samples.notation.CreditOk;
import samples.notation.InsufficientCredit;
import samples.notation.Confirmation;
import samples.notation.OutOfStock;
protocol Purchasing @ Seller {
 participant Buyer, Broker, CreditAgency;
 Order from Broker;
 choice to Broker {
  when Confirmation {
  }
  when OutOfStock {
  }
 }
}
```

2.5. Simulating a Protocol description against an Event List

The *simulate* command takes two parameters. The first parameter is the located protocol description and the second parameter is the event list to be simulated against the protocol.

For example, if the user is in the top level folder of the Scribble tools distribution, without the `bin` folder being added to the system path, then the following command can be executed to simulate the protocol description:

```
bin/simulate.sh samples/scribble/Purchasing@Buyer.spr samples/scribble/Purchasing@Buyer.events
```

The event file is a *comma separated value (csv)* format, with the first column representing the event type, and the second representing the value relevant for the event type. The event types are listed below:

- sendMessage

The value represents the message type.

- receiveMessage
  The value represents the message type.

- sendChoice
  The value represents the choice label.

- receiveChoice
  The value represents the choice label.

- sendDecision
  The value represents the decision boolean value (e.g. true or false). This can be used in conjunction with an *Optional* or *Repeat* protocol construct.

- receiveDecision
  The value represents the decision boolean value (e.g. true or false). This can be used in conjunction with an *Optional* or *Repeat* protocol construct.

The event file used in the sample command above is:

```
sendMessage,samples.notation.Order
receiveChoice,_Confirmation
receiveMessage,samples.notation.Confirmation
```

and the result of running the command is:

```
INFO: Validated SendMessage samples.notation.Order
INFO: Validated ReceiveChoice _Confirmation
INFO: Validated ReceiveMessage samples.notation.Confirmation
```

Embedding Scribble Tools

This section describes how to embed the Scribble tools into an application.

3.1. OSGi Services

This section shows how to use Scribble tools in an OSGi environment. The components within the Scribble tools are all implemented as OSGi services, and therefore can be easily accessed by other OSGi compliant modules.

3.1.1. Parsing a Protocol description

3.1.2. Validating a Protocol description

--- using the validation manager

3.2. Direct Injection

Direct injection is a mechanism for initialising the relationship between components based on the specification of interest in an interface. When an implementation of a required interface is instantiated, all components that expressed an interest in the interface will be initialised with the implementation.

Direct injection frameworks, such as Spring, can be used to automate this mechanism, by providing the necessary relationships in a configuration file. Alternatively an application can wire these relationships using an explicit mechanism.

The main benefit of this approach is that the 'using' component is unaware of the implementation of the interface - it simply expresses a need for a component that implements the required interface, without caring how it comes in to existence.

When the components are used in an OSGi context, the OSGi bundle activation mechanism is used to initialise the modules, and register them as being available for use by other components.

Note

Show some examples using spring, or another relevant approach, to initialising the tools within an app.