Scribble 2.0

Protocol Guide

Overview

Scribble is a simple text based language for describing interactions between multiple parties in the form of a *Protocol*.

The *Protocol* notation is intended to provide an abstract description of the communication behaviour, with addtional optional layers being used to describe assertions and other governance related information.

In the world of programming languages, the measure of the simplicity of a language is based on its representation of a simple 'hello world' example. The Scribble Protocol representation of such an example would be:

```
protocol HelloWorld {
    role You, World;
    Hello from You to World;
}
```

The notation will be explained in more detail in this guide, but basically the protocol is named 'HelloWorld', and describes the communication behaviour between two roles (or parties). A single interaction has been defined, sending a message of type 'Hello' from role 'You' to role 'World'.

In the following chapter we will explore the main constructs of the protocol notation. Then we will explore a range of examples, before concluding with a brief overview of some more advanced constructs that are still under development.

Protocol Constructs

2.1. Protocol Definition

The protocol definition is comprised of a:

- Imports

  A list of import statements that can reference types (messages or other protocols) in other namespaces, optionally providing a URL to the detailed information about the type.

- Protocol unit

  The protocol definition contains a single top level protocol unit, with a name that is scoped to the previously defined namespace.

The syntax for the process definition is:

```
( import [ <TypeSystem> ]
   [ "<DataType>" as ] <name>
   ( "," [ "<DataType>" as ] <name> )*
   [ from "<Location>" ] ";" )*

protocol <name> [ "@" <Role> ]
     [ "(" ( role | <Type> ) <name>
    ( "," role | <Type> ) <name> )*
      ")" ] "{"
    .... ";"
"}"
```

The import statement is used to define a type that will be used within the protocol definition. When referenced in the protocol, the type is known by a local name (or alias). If we want to be able to monitor, or use the protocol definition in any other 'real world' situation, then we need to bind the concrete type information to this alias.

The import statement can optionally define a type system associated with the imported type. For example, this could be 'java' if referring to a Java class or interface, or 'xsd' for an XSD type or element.

Within the type information, we can identify a specific data type, followed by the 'as' keyword and then the name of the type alias.

The import can optionally specify the location of the type information, by specifying the 'from' keyword followed by a string literal with type system specific location information.

In its simpliest form, the import can just define the type name, which will be represented without any type system specific information. The next level can introduce a type specific 'data type' value. Finally the most complete version will include the location of the type information.

Following the import statements is the declaration of the protocol unit itself. This defines the name of the protocol and whether it is located at a particular role.

The following represents a 'global' protocol example:

```
import Customer;

import xsd
    "{http://www.acme.org/Purchasing}Order" as Order,
    "{http://www.acme.org/Purchasing}Quote" as Quote
    from "../schema/MySchema.xsd";

import java
  "Order" as Order,
  "Quote" as Quote
  from "org.scribble";

protocol PurchaseGoods {
    role Buyer, Seller;
    ....
}
```

This example shows three variations of the import statement. The first importing a single type based on a name, without any concrete type information being bound.

The second importing a particular XSD schema, from a schema location, and referring to two specific types within the schema. The first being an XSD type, known by the qualified name *{http://www.acme.org/Purchasing}Order* and locally referred to using the alias *Order*. The second being an XSD element, known by the qualified name *{http://www.acme.org/Purchasing}Quote* and locally referred to using the alias *Quote*.

The third import statement shows the case where two Java classes are bound to local aliases. The Java package is specified within the 'from' clause, and the class name is defined prior to the 'as' keyword in each case.

The global protocol is then defined, named as *PurchaseGoods*. This is a global protocol because it does not specify a particular role at which the definition is located.

A local protocol variation would be:

```
protocol PurchaseGoods @ Buyer {
    role Seller;
    ....
}
```

This local representation of the protocol defines the behaviour from the *Buyer* role's perspective. That is why the roles declared within the protocol unit only include the *Seller*, as this is the role with which the *Buyer* is going to communicate.

The protocol can also be defined with parameters, to allow other protocols to invoke them with specific values. Below is a variation of the previous example, with the roles passed into the protocol instead.

```
protocol PurchaseGoods(role Buyer,
         role Seller) {
    ....
}
```

The way in which another protocol can be invoked will be presented in a subsequent section.

2.2. Interaction

Interactions in Scribble are based on two assumptions:

- Asynchrony, so no wait on sends, and
- Message order preservation for messages sent to the same role

The syntax for the interaction is:

```
<MessageSignature> [from <Role>] [ to <Role> ( "," <Role> )* )]
```

The following example shows a similar type of interaction as shown in the 'hello world' example.

```
role Customer, Supplier;
Order from Customer to Supplier;
```

In this sample, two roles are declared, with the interaction indicating that a message of type 'Order' will be sent from role 'Customer' to role 'Supplier'.

```
placeOrder(Order) from Customer to Supplier;
```

This example demonstrates an alternative way for the exchanged message to be specified. In the first sample a message-style was used. In this sample, an RPC style has been used, specifying the operation name with type parameters. In this case, only a single typed parameter *Order* has been specified, but this could be a comma separated list of one or more types.

Note

When specifying interactions, it is not possible to just define an operation name with no type parameters.

2.3. Sequence

The *sequence* construct is a list of activities, separated by a semi-colon, such that each subsequent activity is only performed after the completion of the preceding activity.

```
"{"
    ( <Activity> ";" )*
"}"
```

where *Activity<i>* represents any protocol based activity or construct.

The following example shows a sequence of interactions.

```
{
    Order from Buyer to Seller;
    Invoice from Seller to Buyer;
    Payment from Buyer to Seller;
    Confirmation from Seller to Buyer;
}
```

2.4. Choice

The *choice* construct represents a set of mutually exclusive paths triggered by different interactions that could occur between two roles. One of the roles will be the decision maker, initiating the interaction, and the other role will be the recipient, reacting to the specific message received.

The syntax for the **choice** construct is:

```
choice [from <Role>] [to <Role> ( "," <Role> )* ] "{"
 <MessageSignature> ":"
     ...

 <MessageSignature> ":"
     ...
"}"
```

For example,

```
CreditCheck from Seller to CreditAgency;

choice from CreditAgency to Seller {
    CreditRefused:

    CreditOk:
}
```

For example,

```
CreditCheck from Seller to CreditAgency;

choice from CreditAgency to Seller {
    CreditRefused:

    CreditOk:
}
```

Another example is,

```
Order from Buyer to Broker;

choice from Broker to Buyer {
    validProduct():
        Order from Broker to Supplier;
        Confirmation from Supplier to Broker;
        OrderDetails from Broker to Buyer;

    invalidProduct(UnknownProduct):
}
```

In this example, the first choice path defines a message signature with only a label (or operation name). The significance of this is that, depending upon the message transport being used, this label may not be carried from the *Broker* to the *Buyer*. The possible scenarios are:

- Label sent as a control message
  If supported by the message transport, the label can be immediately issued as an RPC operation with no message content.
- Label piggy backed on next relevant message
  If the transport is message based, as opposed to RPC based, then the next message carried from the *Broker* to *Buyer* can include some additional metadata to define the label.
- Label is ignored
  If the transport is message based, then it is possible to use the next message between the *Broker* and *Buyer* to determine which path has been taken. In the example above, the first path would be distinguished based on observing the *OrderDetails* message, and the second path the *UnknownProduct* message.

2.5. Parallel

The *par* construct defines a set of paths that represent behaviour that should occur concurrently.

The syntax for the **par** construct is:

```
par "{"
    ...
( "}" and "{"
    ... )+
"}"
```

For example,

```
par {
    CheckStock from Seller to Wholesaler;
    StockAvailability from Wholesaler to Seller;
} and {
    CreditCheck from Seller to CreditAgency;
    CreditReport from CreditAgency to Seller;
}
```

2.6. Unordered

The *unordered* construct defines a set of statements that represent behaviour that should occur in any order.

The syntax for the **unordered** construct is:

```
unordered "{"
    ...
"}"
```

For example,

```
unordered {
    CheckStock from Seller to Wholesaler;
    CreditCheck from Seller to CreditAgency;
}
```

2.7. Repetition

The *repeat* construct represents the 'while' style loop. A decision will be made at one or more nominated roles. If more than one located role is defined, then all of those roles must synchronize in their decision making, using some non-observable mechanism.

The first activity contains within the repetition construct must be initiated at one of the located roles associated with the construct.

The syntax for the **repeat** construct is:

```
repeat "@" <Role> { "," <Role> }* "{"
    ...
"}"
```

The following example shows a repeat construct, located at the *Buyer* role. This means that the Buyer will be responsible for deciding when to iterate, and when to terminate the repetition.

It also means that the initial activity (in this case interaction) defined within the repeat construct must be initiated by the Buyer. In this case, the *Buyer* is sending an *Order* message to the *Seller*.

```
repeat @ Buyer {
    Order from Buyer to Seller;
    Invoice from Seller to Buyer;
}
```

2.8. Recursion

Recursion is supported in the protocol definition by defining a 'rec' keyword with a label prior to a block, that defines the scope of the recursive behaviour, and at some point in the enclosed behaviour, the same label is used to show where the recursion should be performed. The label can only be used within the scope of the recursion block to which the label has been associated.

```
rec <Label> "{"
    ...
    <Label> ";"
"}"
```

The following example shows a recursion construct defined using the label 'Transaction'. Within the associated block, the recursion is triggered by the 'Transaction' clause.

```
rec Transaction {
    ...
 Transaction;
}
```

2.9. Global Escape

The 'global escape' concept provides a means for breaking out of a particular scope based on an interaction. It is similar to the structure of a try/catch mechanism used in traditional programming languages for dealing with exceptions, and therefore the same construct has been used.

The syntax for the **global escape** construct is:

```
try "{"
 ...
( "}" catch "(" <MessageSignature> from <Role> to <Role> ")" "{"
 ... )+
"}"
```

In the following example, the body of the *try* block is enacted, involving an interaction between a *Buyer* and *Seller*, followed by some other activities.

During this scoped set of activities, if the *Seller* returns an *OutOfStock* message, then it will cause the flow of control to move to the first catch block. However if the *Buyer* sends an *OrderExpired* or *OrderCancelled* message, then the flow will move to the second catch block.

```
try {
 Order from Buyer to Seller;
 ...
} catch (OutOfStock from Seller to Buyer) {
 ...
} catch (expire(OrderExpired) from Buyer to Seller
    | OrderCancelled from Buyer to Seller) {
 ...
}
```

2.10. Composition

Protocols can be defined in a modular way, with one protocol being able to compose another using the *run* construct.

The *run* construct composes another protocol in a synchronous manner. This means that the composed protocol will complete before any subsequent activity in the composing protocol can proceed.

There are three ways in which another protocol can be composed. These are:

- Nested
  The nested variation defines the sub-protocol as an inner part of the composing protocol - in a similar way to an inner class in Java.
- External
  The external variation defines the sub-protocol in a separate protocol definition, which is then referenced within the composing protocol.
- Inline
  The inline option defines an anonymous protocol.

The syntax for the nested and external *run* construct is:

```
run <ProtocolName>
   [ "(" <param> ( "," <param> )* ")" ] "{"
    ...
"}"
```

An example of the internal variation, using the *run*, is:

```
role Client, Supplier;
....
```

```
run PlaceOrder(Client, Supplier);
....
protocol PlaceOrder(role Buyer, role Seller) {
    ....
}
```

The external variation is similar to the internal variation above, except that the composed protocol definition (i.e. PlaceOrder in this case), would be stored in a separate definition.

The syntax for the inline *run* construct is:

```
run protocol "{"
    ...
"}"
```

Examples

This chapter presents some examples using the *protocol* notation.

3.1. Buyer Seller Protocol

This example shows how a *Buyer* participant and *Seller* participant may interact in an ordering process.

```
protocol BuyerSeller {
    role Buyer, Seller;

    Order from Buyer to Seller;

    choice from Seller to Buyer; {
        Invoice {
        }
        Rejected {
        }
 }
}
```

3.2. Credit Check Protocol

This example shows how a *Client* role performs a credit check against a *CreditAgency* role.

```
protocol CreditCheck {
    role Client, CreditAgency;

    CheckCredit from Client to CreditAgency;

    choice from CreditAgency to Client {
        CreditOk {
        }
        NoCredit {
        }
    }
}
```

3.3. Purchasing Goods Conversation

This example shows how a protocol can be defined that 'implements' the previous two protocol examples.

```
protocol BuyerSellerCreditCheck {
    role Buyer, Seller, CreditAgency;

    Order from Buyer to Seller;

    CheckCredit from Seller to CreditAgency;

    choice from CreditAgency to Seller {
        CreditOk {
         Invoice from Seller to Buyer;
        }
        NoCredit {
         Rejected from Seller to Buyer;
        }
    }
}
```