Scribble

Protocol Guide

by Kohei Honda and Gary Brown

Overview

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

"Scribbling is necessary for architects, either physical or computing, since all great ideas of architectural construction come from that unconscious moment, when you do not realise what it is, when there is no concrete shape, only a whisper which is not a whisper, an image which is not an image, somehow it starts to urge you in your mind, in so small a voice but how persistent it is, at that point you start scribbling" - Kohei Honda 2007

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 {
    participant 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 participants (or parties). A single interaction has been defined, sending a message of type 'Hello' from participant 'You' to participant '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:

- Namespace

  The namespace helps to organise protocol definitions into categories based on a hierarchical naming convention.

- 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:

```
namespace <namespace> ";"

{ import <fully qualified type> [ "@" <URL> ] ";" }*

protocol <name> [ "@" <Participant> "{"
    .... ";"
"}"
```

The namespace can define a user specific hierarchy to organise protocol definitions. The separate parts of the hierarchy can be specified separated by '.'.

The import statement enables fully qualified types to be referenced within the protocol definition based on their local name. For example, the type *my.scribble.type.Order* could be imported into a protocol definition, and subsequently referenced based only on the local part of the fully qualified name, i.e. Order.

The types can represent any scribble type that needs to be referenced. This could be message types, exchanged as part of an interaction, or other protocol definitions.

The final part of the protocol definition is the declaration of the protocol unit itself. This defines the name of the protocol and whether it is located at a particular participant.

The following represents a 'global' protocol example:

```
namespace my.scribble.examples;

import my.scribble.type.Order;
import my.scribble.type.Trade @ http://www.examples.org/types/Trade.xsd ;

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

The namespace can be any user defined scope. In this case it is *my.scribble.examples*.

This example shows two variations of the import statement. The first importing a single type based on its fully qualified name. The second importing a particular type, and also identifying a URL where more details about the type can be found.

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

A local protocol variation would be:

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

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

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 participant

The syntax for the interaction is:

```
<MessageSignature> [from <Participant>] [to <Participant>]
```

```
```

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

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

In this sample, two participants are declared, with the interaction indicating that a message of type 'Order' will be sent from participant 'Customer' to participant '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 zero or more types.

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 participants. One of the participants will be the decision maker, initiating the interaction, and the other participant will be the recipient, reacting to the specific message received.

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

```
choice [from <Participant>] [to <Participant>] "{"
 when <MessageSignature> "{"
     ...
    "}"
 when <MessageSignature> "{"
     ...
    "}"
"}"
```

For example,

```
CreditCheck from Seller to CreditAgency;

choice from CreditAgency to Seller {
    when CreditRefused {
    }
    when CreditOk {
    }
}
```

2.5. Repetition

The *repeat* construct represents the 'while' style loop. A decision will be made at one or more nominated participants. If more than one located participant is defined, then all of those participants 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 participants associated with the construct.

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

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

The following example shows a repeat construct, located at the *Buyer* participant. 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;
}
```

2.6. 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 two 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.

The syntax for the *run* construct is:

```
run <ProtocolName>"(" <CP1> = <P1> { "," <CPn> = <Pn> }* ")" "{"
    participant CP1 { "," CPn }* ";"
    ...
"}"
```

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

```
participant Client, Supplier;
....
run PlaceOrder(Buyer = Client, Seller = Supplier);
....
```

```
protocol PlaceOrder {
    participant Buyer, 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. A *run* based external variation for the composing protocol would be:

```
participant Client, Supplier;
....
run PlaceOrder(Buyer = Client, Seller = Supplier);
```

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 {
    participant Buyer, Seller;

    Order from Buyer to Seller;

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

3.2. Credit Check Protocol

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

```
protocol CreditCheck {
    participant Client, CreditAgency;

    CheckCredit from Client to CreditAgency;

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

3.3. Purchasing Goods Conversation

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

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

    Order from Buyer to Seller;

    CheckCredit from Seller to CreditAgency;

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

Unsupported

This chapter describes constructs that are still *work in progress*.

This means that they may be included in the tool support for the notation, but may change in a future version. So any use of these constructs are not guaranteed to be supported in future versions of the tools.

4.1. Parallel

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

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

```
parallel "{"
    ...
{ "}" and "{"
    ... }+
"}"
```

For example,

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

4.2. Global Escape

The concept of a 'global escape' is to support the abrupt termination of a set of activities based on the occurance of a situation. The situation may result from an interaction, or an internal condition within one of the co-operating participants, which subsequently results in the other parties being informed to 'escape' from their normal activities.

```
try "{"
    ...
{ (
"}" catch [ "@" <Participant> {"," <Participant>}* ] <Type> "{"
...
|
```

```
"}" interrupt [ "@" <Participant> {"," <Participant>} ] "{"
    ...
) }*
"}"
```

There can be zero or more catch blocks, and zero or more interupt blocks.

The *catch* block represents a set of activities triggered based on an exception being raised at local participant (see the *raise* activity. If a located participant is specified, then this will be the participant at which the exception will be raised - and the initiator participant for any subsequent activity performed in the block. If more than one participant is specified, then each of these participants will need to be synchronized in their decision to escape from the try block.

The *interrupt* block represents a set of activities triggered by some internal condition. If a located participant is specified, then this will be the participant at which the interrupt will occur. As with the catch block, this will also be the initiator participant for any subsequent activity performed in the block. If more than one participant is specified, then each of these participants will need to be synchronized in their decision to escape from the try block.

The following example shows how internal decisions within the *Buyer* can be used to escape from the repetition of receiving quotes from the *Seller*, and result in either accept or cancel quote messages being sent to the *Seller*.

```
try {
    repeat @ Seller {
        Quote from Seller to Buyer;
    }
} interrupt @ Buyer {
    AcceptQuote from Buyer to Seller
} interrupt @ Buyer {
    CancelQuote from Buyer to Seller
}
```

The following variation repetitively gets a quote from the *Supplier* until the *Buyer* decides to escape from the normal block by raising a *Quit* type that is caught at the *Buyer* and used to send a *CancelQuote* message to the *Seller*.

```
try {
    Enquiry from Buyer to Seller;
    repeat @ Seller {
        Quote from Seller to Buyer;
        choice @ Buyer {
            raise @ Buyer Quit;
        } or {
            ...;
        }
```

```
    }
} catch @ Buyer Quit {
    CancelQuote from Buyer to Seller
}
```

4.3. Raise

The *raise* construct is used in conjunction with the global escape *catch* block, to enable a normal flow to terminate and jump to the appropriate *catch* block.

Note

How do we handle propagation - if no catch block exists for raised type, then it is passed out? Does this apply to sub-protocols?

```
raise "@" <Participant> <Type>;
```

As shown in the previous section on global escape, the *raise* can be used to escape from a normal flow of activities into a *catch* block.

```
try {
    Enquiry from Buyer to Seller;
    repeat @ Seller {
        Quote from Seller to Buyer;
        choice @ Buyer {
            raise @ Buyer Quit;
        }
    }
} catch @ Buyer Quit {
    CancelQuote from Buyer to Seller
}
```

4.4. Unordered

4.4.1. Syntax

```
unordered "{"
    ( <Activity1> ";" )+
```

```
"}"
```

Note

Syntax to be defined. One possibility would be to find a symbol, other than ';' to represent an ordered relationship between the activities. However this is easy to misread, and therefore misunderstand the description. We could use a similar structure as with **parallel**, with each unordered activity being in a separate path - this would enable groups of activities to be unordered - but then this would get confusing with parallel. The syntax above is another possibility, so use the same ';' separate as sequence, but enclose in an 'unordered' region.

4.4.2. Usage

```

```

4.5. Merge

4.5.1. Syntax

```

```

4.5.2. Usage

```

```