Scribble Protocol Language Guide

by Scribble Team (Version 0.3.0 Draft 1)

Chapter 1. Overview

This chapter provides an overview of Scribble.

1.1. What is Scribble?

Scribble is a language to describe application-level protocols among communicating systems. A protocol represents an agreement on how participating systems interact with each other. Without a protocol, it is hard to do a meaningful interaction: participants simply cannot communicate effectively, since they do not know when to expect the other parties to send their data, or whether the other party is ready to receive a datum it is sending. In fact it is not clear what kinds of data is to be used for each interaction. It is too costly to carry out communications based on guess works and with inevitable communication mismatch (synchronisation bugs). Simply, it is not feasible as an engineering practice.

Scribble presents a stratified description language:

- The bottom layer is the type layer, in which we describe the bare skeleton of conversations structures as types for interactions (known in the literature as session type).
- The assertion layer allows elaboration of a type-layer description using assertions.
- Finally the third layer, protocol document layer, allows description of multiple protocols and constraints over them.

Each layer offers distinct behavioural assurance for validated programs.

1.2. How can it be used?

The development and validation of programs against protocol descriptions could proceed as follows:

- A programmer specifies a set of protocols to be used in her application.
- She can verify that those protocols are valid, free from livelocks and deadlocks.
- She develops her application referring to those protocols, potentially using communication constructs available in the chosen programming language.
- She validates her programs against protocols using a protocol checker, which detects lack of conformance.
- At the execution time, a local monitor can validate messages with respect to given protocols, optionally blocking invalid messages from being delivered.

Chapter 2. Defining a Protocol

2.1. A Basic Protocol

The following example shows the basic structure of a Scribble protocol:

```
module scribble.example.Basic;

type <xsd> "{http://scribble.org/examples}Greetings" from "HelloWorld.xsd"
 as Greetings;

global protocol HelloWorld (role Me, role World) {
        hello(Greetings) from Me to World;
        hello(Greetings) from World to Me;
}
```

The first line always defines the `module`, which is a *.* separated name representing the location of the Scribble protocol file within a hierarchy.

The last part of the module name, `Basic` in this case, represents the filename with a *.scr* extension. The preceding parts represent a folder/directory hierarchy. So the protocol above would be contained in a file *scribble/example/Basic.scr*.

Note

Need to describe root folder, or search path.

The purpose of a protocol is to exchange typed messages between participants. The third line shows an example of how the protocol declares these types.

The part between < > identifies the nature of the type, in this case it is an XSD (XML Schema Definition). The next part, which is in double quotes, identifies the type in a format appropriate for the schema. The `from` value (also in double quotes) represents the location of the schema definition. Finally the `as` value is an alias which the protocol will use to reference this type.

The final part of this simple protocol is the `global protocol` definition. The name of this global protocol is `HelloWorld` and it is associated with two roles, `Me` and `World`. This means that the protocol will describe the interactions that occur between these roles from a global (or endpoint neutral) perspective. Later in this guide we will discuss the topic of projection and the local protocols that describe the behaviour for each of the roles involved in the global protocol.

In the body of the global protocol there is two message transfer (or interaction) statements. This statements identify the message signature (operation name followed by the types(s) within the braces), and the `from` and `to` roles. So in this example the role `Me` is saying *hello* to `World`, and then `World` responds by saying *hello* back.

The role names used in the `from` and `to` clauses must have previously been defined in the protocol header.

Tip

Although described as two separate statements, this simple pattern can be used to represent a request/response interaction between the two parties (roles). If this is the intention, then when defining the protocol it is recommended that the operation name be the same on both statements.

2.2. Adding Decision Points

The previous section introduced a simple protocol which involved a sequence of two interactions (or message transfers). Most protocols involve some decision points which result in the protocol having a different behaviour.

The following example illustrates how a decision point can be defined. We'll change the protocol to represent a more interesting purchasing example:

```
module scribble.example.Purchasing;

type <xsd> "{http://scribble.org/examples}QuoteRequest" from
 "Purchasing.xsd" as QuoteRequest;
type <xsd> "{http://scribble.org/examples}Quote" from "Purchasing.xsd" as
 Quote;
type <xsd> "{http://scribble.org/examples}Order" from "Purchasing.xsd" as
 Order;
type <xsd> "{http://scribble.org/examples}OrderAck" from "Purchasing.xsd" as
 OrderAck;
type <xsd> "{http://scribble.org/examples}OutOfStock" from "Purchasing.xsd"
 as OutOfStock;

global protocol BuyGoods (role Buyer, role Seller) {
        quote(QuoteRequest) from Buyer to Seller;

        choice at Seller {
                quote(Quote) from Seller to Buyer;
                buy(Order) from Buyer to Seller;
                buy(OrderAck) from Seller to Buyer;
        } or {
                quote(OutOfStock) from Seller to Buyer;
        }
}
```

As in the original example, we have a module definition followed by the type declarations. In this example we have five types defined.

The `global protocol` is similar to the first example, however instead of simply returning a single response, this example introduces the `choice` construct.

A `choice` represents a decision point within the protocol. Decisions must always be made at a nominated role, with that decision then being communicated to the other roles based on the behaviour defined in the selected path.

This means that each path within the choice *must* begin within a message transfer initiated by the role that made the decision. Therefore, the following choice statement would be invalid:

```
        choice at Seller {
                quote(Quote) from Seller to Buyer;
        } or {
                // Invalid, as initial interaction must come from role
'Seller'
                cancel(CancelRequest) from Buyer to Seller;
        }
```

To ensure that the other (non-decision making) roles correctly infer which path should be taken, the message types of the initial message transfers from the decision making role must be distinct. So, the following example would be invalid:

```
        choice at Seller {
                quote(Quote) from Seller to Buyer;
                buy(Order) from Buyer to Seller;
                buy(OrderAck) from Seller to Buyer;
        } or {
                // Invalid, as type 'Quote' was used in the other choice
path
                quote(Quote) from Seller to Buyer;
                reject(Order) from Buyer to Seller;
        }
```

Note

Protocols where the decision is conveyed within the message contents is not currently handled by this version of Scribble.

2.3. Concurrency

This section will discuss how protocols can be defined where behaviour can be performed concurrently (i.e. in parallel). We will extend the previous example by adding some new roles, and a parallel construct to check concurrently whether the buyer’s credit is acceptible and if there is enough stock to satisfy the quote.

```
module scribble.example.Purchasing;
```

```
type <xsd> "{http://scribble.org/examples}QuoteRequest" from
 "Purchasing.xsd" as QuoteRequest;
type <xsd> "{http://scribble.org/examples}Quote" from "Purchasing.xsd" as
 Quote;
type <xsd> "{http://scribble.org/examples}CreditReport" from
 "Purchasing.xsd" as CreditReport;
type <xsd> "{http://scribble.org/examples}StockAvailability" from
 "Purchasing.xsd" as StockAvailability;
type <xsd> "{http://scribble.org/examples}Order" from "Purchasing.xsd" as
 Order;
type <xsd> "{http://scribble.org/examples}OrderAck" from "Purchasing.xsd" as
 OrderAck;
type <xsd> "{http://scribble.org/examples}InsufficientCredit" from
 "Purchasing.xsd" as InsufficientCredit;
type <xsd> "{http://scribble.org/examples}OutOfStock" from "Purchasing.xsd"
 as OutOfStock;

global protocol BuyGoods (role Buyer, role Seller, role CreditAgency, role
 Store) {
        quote(QuoteRequest) from Buyer to Seller;

        par {
                creditCheck(Quote) from Seller to CreditAgency;
                creditCheck(CreditReport) from CreditAgency to Seller;
        } and {
                stockCheck(Quote) from Seller to Store;
                stockCheck(StockAvailability) from Store to Seller;
        }

        choice at Seller {
                quote(Quote) from Seller to Buyer;
                buy(Order) from Buyer to Seller;
                buy(OrderAck) from Seller to Buyer;
        } or {
                quote(InsufficientCredit) from Seller to Buyer;
        } or {
                quote(OutOfStock) from Seller to Buyer;
        }
}
```

Each of the paths defined within the par construct are expected to occur concurrently. The protocol will not progress past the par construct until each of the paths within the par have completed.

Note

To avoid potential conflicts between message exchanges in the concurrent paths, it is currently necessary to use distinct operator names between role pairs in different paths. So (for example), in the above example, the second path could not use *creditCheck* as an operator.

2.4. Recursion

The recursion construct can be used to define protocols involving repetition.

```
module scribble.example.Purchasing;

type <xsd> "{http://scribble.org/examples}QuoteRequest" from
 "Purchasing.xsd" as QuoteRequest;
type <xsd> "{http://scribble.org/examples}Quote" from "Purchasing.xsd" as
 Quote;
type <xsd> "{http://scribble.org/examples}Order" from "Purchasing.xsd" as
 Order;
type <xsd> "{http://scribble.org/examples}OrderAck" from "Purchasing.xsd" as
 OrderAck;
type <xsd> "{http://scribble.org/examples}OutOfStock" from "Purchasing.xsd"
 as OutOfStock;

global protocol BuyGoods (role Buyer, role Seller) {

        quote(QuoteRequest) from Buyer to Seller;

        rec SubmitQuote {
                choice at Seller {
                        quote(Quote) from Seller to Buyer;
                        buy(Order) from Buyer to Seller;
                        buy(OrderAck) from Seller to Buyer;
                } or {
                        quote(OutOfStock) from Seller to Buyer;
                }

                choice at Buyer {
                        quote(QuoteRequest) from Buyer to Seller;

                        continue SubmitQuote;
                } or {
                        quit() from Buyer to Seller;
                }
        }
}
```

The `rec` block defines the scope of the recursion, with a label to identify the block (i.e. SubmitQuote in this example).

The `continue` statement defines where the recursion should actually occur. In this example, after the initial quote request has been handled, the `Buyer` then has the option to either submit a further quote, or quit the session. If they submit another quote request, then it will continue back to the start of the recursion block where it will await the response from the `Seller`.

The recursion block will only complete when the `Buyer` sends a `quit` message to the `Seller`.

Chapter 3. Advanced Constructs

3.1. Interruptible

The *interruptible* construct is used to describe a communication where one party, that is pending a message from another party, decides to interrupt the conversation (i.e. abort, timeout, cancel). This situation needs special consideration, as the interrupt message may cross the response from the other party, leading to non-deterministic behaviour in both roles.

```
module scribble.example.Interruptible;

type <xsd> "{http://scribble.org/examples}Order" from "Examples.xsd" as
 Order;
type <xsd> "{http://scribble.org/examples}Receipt" from "Examples.xsd" as
 Receipt;

global protocol GInterruptibleTest(role Buyer,role Seller) {
        interruptible MyLabel: {
                buy(Order) from Buyer to Seller;
                buy(Receipt) from Seller to Buyer;
        } with {
                cancel(Order) by Buyer;
        }
}
```

The *interruptible* construct enables the normal flow of behaviour to be described within the main block, with the possible interrupt messages to be defined in the **with** clause.

3.2. Multicast

The multicast pattern enables a message to be sent from one role to more than one other role.

```
module scribble.example.Multicast;

type <xsd> "{http://scribble.org/examples}Notification" from "Examples.xsd"
 as Notification;

global protocol Notify (role Client, role ServerA, role ServerB) {
        send(Notification) from Client to ServerA, ServerB;
}
```

In this example, the message is sent from the *Client* role to roles *ServerA* and *ServerB* at the same time.

3.3. Sub-Protocols

As protocols become more complicated, as with any type of program, we need a way to decompose the description into more manageable components. The other significant advantage of this decomposition is to create reuseable components that can be used in building many other higher level protocols (especially in combination with the geneics support described in the next section).

Sub-protocols can either be:

- invoked within the same module, if the *calling* and *called* protocols are defined in the same module file
- defined in separate modules with,
  - the **do** statement in the *calling* protocol providing the fully qualified module and protocol name
  - the *calling* prototocol's module importing the *called* protocol's module

Each of these approaches will be described in the following sub-sections.

3.3.1. Invoking a protocol in the same module

The following example shows how to call a protocol that has been defined within the same module.

```
module scribble.example.SubProtocols;

type <xsd> "{http://scribble.org/examples}Notification" from "Examples.xsd"
 as Notification;

global protocol Main (role Client, role Server) {
        do Sub(Client as R1, Server as R2);
}

global protocol Sub (role R1, role R2) {
        send(Notification) from R1 to R2;
}
```

The global protocol *Main* is calling the global protocol *Sub*. In this example, the roles defined within the *Main* protocol are mapped to the ones declared in the *Sub* protocol using the **as** keyword. Roles can also be associated based on their position within the list, e.g.

```
        do Sub(Client, Server);
```

3.3.2. Expliciting referencing a protocol in another module

The following example shows how to call a protocol that has been defined within another module based on a fully qualified name. The first module is the *main* module with the *calling* protocol,

```
module scribble.example.MainModule;

global protocol Main (role Client, role Server) {
        do scribble.example.SubModule.Sub(Client as R1, Server as R2);
}
```

and the second module is the *sub* module with the *called* protocol,

```
module scribble.example.SubModule;

type <xsd> "{http://scribble.org/examples}Notification" from "Examples.xsd"
 as Notification;

global protocol Sub (role R1, role R2) {
        send(Notification) from R1 to R2;
}
```

3.3.3. Importing a protocol from another module

The following example shows how to call a protocol that has been imported from another module. The first module is the *main* module with the *calling* protocol,

```
module scribble.example.MainModule;

import scribble.examples.SubModule;

global protocol Main (role Client, role Server) {
        do Sub(Client as R1, Server as R2);
}
```

and the second module is the *sub* module with the *called* protocol,

```
module scribble.example.SubModule;

type <xsd> "{http://scribble.org/examples}Notification" from "Examples.xsd"
 as Notification;

global protocol Sub (role R1, role R2) {
        send(Notification) from R1 to R2;
}
```

In the main module, the fully *sub* module is imported, allowing any of the types or protocols defined in that module to be referenced based on the local names (e.g. *Sub* for the protocol name).

It is also possible to just import a particular protocol or type from another module, using the following variation of the import statement:

```
from scribble.examples.SubModule import Sub;
```

Finally, if there is a name conflict with the protocol or type being imported, then it can be locally renamed using an alias. For example,

```
from scribble.examples.SubModule import Sub as OtherProtocol;

...

        do OtherProtocol(Client as R1, Server as R2);
...
```

3.4. Generics and parameters

Note

TO BE DEFINED

Chapter 4. Projection

- local protocol variations
- projects usually automatically generated using tooling, but may want to create to define endpoint behaviour and validate against a global description, so this section shows the differences from a global description.