Tutorial on the B2Scala Tool: modelling a restaurant application

Doha Ouardi$^{1[0009-0007-0213-0748]}$,
Manel Barkallah$^{1[0000-0003-2608-5658]}$, and
Jean-Marie Jacquet$^{1[0000-0001-9531-0519]}$

Nadi Research Institute, Faculty of Computer Science, University of Namur
Rue Grandgagnage 21, 5000 Namur, Belgium
{doha.ouardi,manel.barkallah,jean-marie.jacquet}@unamur.be

Abstract. This article aims at introducing users to the B2Scala tool through the modelling of a restaurant application. It is written as a complement to the article "The B2Scala Tool: integrating Bach in Scala with Security in Mind".

Keywords: Coordination · Bach · Scala.

1 Altar, a tablet-based restaurant

To illustrate the use of B2Scala, we shall use a tablet-based restaurant application. It is subsequently named Altar, as an acronym for *A wonderfuL TablEt-based Restaurant*.

In the aim of being modern and providing a nice experience, the main idea of the Altar restaurant is to use the latest technology to order dishes, have them served and pay at the end of the meal. More precisely, as depicted in Figure 1, each table is equipped with an interactive electronic tablet. Clients are assumed to have created an account before entering the restaurant[1]. After having logged on the tablet, they can consult what is on the menu, place orders (possibly multiple times), track the status of the orders, and pay at the end of the meal. The cooks receive the order in the kitchen and prepare the dishes. Once done, they inform the waiters which serve the dishes at the ordering table. To simplify the modelling, we shall assume that a few Japanese dishes and drinks can be served: nigiri, maki, temaki and beer.

As we shall see, the Bach coordination language and consequently the B2Scala tool provide a solid framework for coordinating the actions of all the agents, including clients, cooks and waiters. It enables transparent communication and ensures that all parts of the restaurant work together efficiently.

As B2Scala relies on Bach, we shall first model the application in Bach and then will show how to derive a B2Scala program from it.

[1] A part that we shall not model.

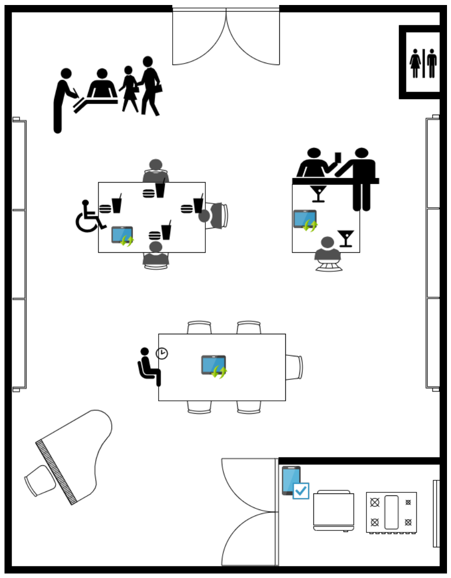

Fig. 1. The Altar Restaurant

2 Bach in a snapshot

To describe the application in Bach, one needs to explain how information placed on the shared space is represented and how it is handled by concurrent processes running around the shared space. This respectively leads to the two following subsections.

2.1 Data

Tokens are used as the fundamental elements to represent information. They consist of discrete units of information with no structure. In contrast, so-called structured information terms, or *si-terms* for short, take the form $f(t_1, ..., t_n)$ where f is a functor and the t_i's are si-terms, possibly being tokens.

Example 1. In the Altar example, `manel` is a token representing a client (in fact one of the authors of this paper) and `order(manel,beer)` is a si-term expressing an order placed by the client to get beer.

2.2 Statements

Primitives. Inspired by Linda, Bach relies on four key primitives tailored for shared space interaction. They are depicted in Figure 2. They consist of

– the *tell(t)* primitive, which places an occurrence of t on the shared space

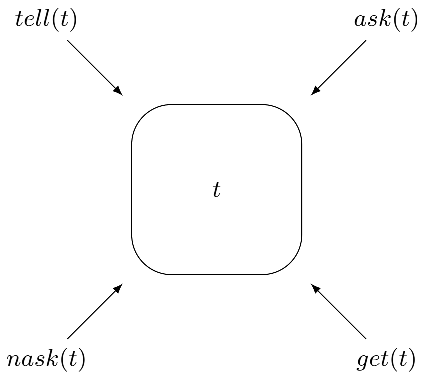

Fig. 2. primitives in Bach

– the *ask(t)* primitive, which checks for the presence of t on the shared space
– the *nask(t)* primitive, which dually checks for its absence
– the *get(t)* primitive, which deletes an occurrence of t on the shared space.

It is important to note that the *tell* primitive always succeeds, while the last three primitives suspend as long as the presence/absence of t is not satisfied. Moreover, the shared space is seen as a multiset of tokens, which naturally leaves room for multiple occurrences.

Example 2. In the Altar example, `tell(order(manel,table_3,beer))` may be used to indicate that client `manel` orders beer at table 3 while `ask(status(manel,served)` may be used to check that the order from `manel` has been served.

Agents. Primitives may be combined to construct agents by using traditional composition operators from concurrency theory:

– sequential composition noted ";",
– parallel composition noted " || ",
– and non-deterministic choice noted "+".

Following concurrency theory, procedures may also be defined to describe agents. Their definition take the following form

$$proc\_name(arg_1, \cdots, arg_n) = agent$$

where *proc_name* is the name of a procedure, the arg_i's are arguments consisting of si-terms and *agent* is an agent, as just described. Following this introduction, procedure calls can be used as primitives are in the composition of agents.

Example 3. Coming back to the Altar restaurant, the behaviour of a client identified by *id* can be depicted as follows:

```
Client(id) = LogIn(id); EnjoyAltar(id); LogOut(id)
```

This definition involves *Client* as a procedure name, *id* as an argument and calls sequentially procedures *LogIn*, *EnjoyAltar* and *LogOut*.

3 Modelling Altar in Bach

Let us now see how the Altar restaurant can be modelled. Three kinds of agents need to be tackled: the clients, the cooks and the waiters. Assuming that we have three clients (manel, alice and emma), one cook (john) and one waiter (tom), the behavior of the restaurant can be modelled by the following parallel composition:

```
Client(manel) || Client(alice) || Client(emma)
          || Cook(john) || Waiter(tom)
```

All these five agents will interact via the shared space. As a result, it is quite easy to introduce a new client, cook or waiter: one has just to introduce him in the parallel composition.

Let us now code these three kinds of agents.

3.1 Modelling a client

Clients – and more generally all the three agents – are modelled in a generic way by introducing an identifier as an argument. We have already described the behavior of a client as doing a log in, enjoying his time at the Alter restaurant and performing a log out:

```
Client(id) = LogIn(id); EnjoyAltar(id); LogOut(id)
```

Doing a log in or a log out is of no particular interest: this is respectively modelled through the telling and getting of the presence of the client, as follows

```
LogIn(id) = tell(logged(id))
LogOut(id) = get(logged(id))
```

Describing the time spent by a client in the Altar restaurant is more interesting. It involves sitting at a table, placing orders, checking for the status of the orders and finally paying. Assuming four tables, numbered t1 to t4, this can be coded as follows.

```
EnjoyAltar(id) =
      SitAtTable(id,t1); EnjoyAtTable(id,t1)
   + SitAtTable(id,t2); EnjoyAtTable(id,t2)
   + SitAtTable(id,t3); EnjoyAtTable(id,t3)
   + SitAtTable(id,t4); EnjoyAtTable(id,t4)
EnjoyAtTable(id,t) =
      HandleOrders(id,t); PayOrders(id,t)
HandleOrders(id,t) =
      MakeOrder(id,t); HandleOrders(id,t)
   + CheckOrder(id,t,prepared); HandleOrders(id,t)
   + CheckOrder(id,t,served); HandleOrders(id,t)
   + CheckOrder(id,t,in_progress); HandleOrders(id,t)
```

Note that these procedures would be in a more compact form if a generalized choice was offered. For instance, `EnjoyAltar` could then be recoded as follows

```
EnjoyAltar(id) = sum t in {t1,t2,t3,t4}
                          SitAtTable(id,t); EnjoyAtTable(id,t)
```

This will in fact be provided by B2Scala but we keep the longer version in this section for the ease of understanding.

Making an order is simulated by placing an order request on the shared space, namely of telling a si-term of the form *order*(id, t, d) for some dish d. This will be taken by the cook who will indicate that the dish is ready by telling a *order_prepared* si-term. Similarly a waiter will indicate that the order has been served by taking a *order_prepared* si-term and telling a *order_served* si-term. Hence checking that the status of an order amount to checking the presence of one of the above si-terms. This results in the following code:

```
MakeOrder(id,t) =
      tell(order(id,t,nigiri))
   + tell(order(id,t,maki))
   + tell(order(id,t,temaki))
   + tell(order(id,t,beer))
CheckOrder(id,t,d,in_progress) =
      ask(order(id,t,d))
CheckOrder(id,t,d,prepared) =
      ask(order_prepared(id,t,d))
CheckOrder(id,t,d,served) =
      ask(order_served(id,t,d))
```

Finally counting the bill is left outside this simple exercise and is just coded by telling the si-term *orders_paid*(id, t):

```
PayOrders(id,t) = tell(orders_paid(id,t))
```

3.2 Modelling a cook

The behavior of a cook consists in repeatedly taking orders and of preparing them. Concretely, this amounts to getting a si-term *order*(id, t, d) – from some user identified by id, sitting at table t and asking for the dish d – and afterwards of telling a *order_prepared*(id, t, d) si-term. Roughly speaking, this can be coded as

```
Cook(id_cook) = get(order(id,t,d)); tell(order_prepared(id,t,d))
```

However, although *id_cook* is an argument, which does not require quantification, the id, t, and t arguments should be quantified on all possible values. This can be achieved by using non-deterministic choices, as we shall do subsequently.

The first quantification on the *id* of users is operated through the *CookForCustomer* procedure, as follows

```
Cook(id_cook) =
      CookForCustomer(id_cook,manel); Cook(id_cook)
   + CookForCustomer(id_cook,alice); Cook(id_cook)
   + CookForCustomer(id_cook,emma); Cook(id_cook)
```

Similarly, the *CookForCustomer* procedure can be coded by the *CookForCustomerAtTable* procedure by making explicit the tables.

```
CookForCustomer(id_cook,id_client) =
      CookForCustomerAtTable(id_cook,id_client,t1)
   + CookForCustomerAtTable(id_cook,id_client,t2)
   + CookForCustomerAtTable(id_cook,id_client,t3)
   + CookForCustomerAtTable(id_cook,id_client,t4)
```

Finally the *CookForCustomerAtTable* procedure can be coded by making explicit the dishes, as follows.

```
CookForCustomerAtTable(id_cook,id_client,id_table) =
      get(order(id_client,id_table,nigiri));
      tell(order_prepared(id_client,id_table,nigiri))
   + get(order(id_client,id_table,maki));
      tell(order_prepared(id_client,id_table,maki))
   + get(order(id_client,id_table,temaki));
      tell(order_prepared(id_client,id_table,temaki))
   + get(order(id_client,id_table,beer));
      tell(order_prepared(id_client,id_table,beer))
```

3.3 Modelling a waiter

A waiter has a similar behavior than a cook. It waits for an *order_prepared* si-term and then puts and *order_served* si-term to indicate that the dish has been served. The following code results by analagy to what we wrote for a cook.

```
Waiter(id_waiter) =
      WaiterForCustomer(id_waiter,manel); Waiter(id_waiter)
   + WaiterForCustomer(id_waiter,alice); Waiter(id_waiter)
   + WaiterForCustomer(id_waiter,emma); Waiter(id_waiter)

WaiterForCustomer(id_waiter,id_client) =
      WaiterForCustomerAtTable(id_waiter,id_client,t1)
   + WaiterForCustomerAtTable(id_waiter,id_client,t2)
   + WaiterForCustomerAtTable(id_waiter,id_client,t3)
   + WaiterForCustomerAtTable(id_waiter,id_client,t4)

WaiterForCustomerAtTable(id_waiter,id_client,id_table) =
      get(order_prepared(id_client,id_table,nigiri));
      tell(order_served(id_client,id_table,nigiri))
   + get(order_prepared(id_client,id_table,maki));
```

```
    tell(order__served(id_client,id_table,maki))
  + get(order_prepared(id_client,id_table,temaki));
    tell(order__served(id_client,id_table,temaki))
  + get(order_prepared(id_client,id_table,beer));
    tell(order__served(id_client,id_table,beer))
```

4 The Altar restaurant in B2Scala

Let us now derive the B2Scala code from the Bach modelling.

4.1 The top goal

The top goal

```
Client(manel) || Client(alice) || Client(emma)
      || Cook(john) || Waiter(tom)
```

is rewritten first by declaring the tokens *manel*, *alice*, ..., then by naming the clients and putting them in parallel to form the *Restaurant* agent. This is achieved by the following code.

```
val manel = Token("Manel")
val alice = Token("Alice")
val emma  = Token("Emma")
val john  = Token("John")
val tom   = Token("Tom")

val CManel = Agent { Client(manel) }
val CAlice = Agent { Client(alice) }
val CEmma  = Agent { Client(emma) }
val CJohn  = Agent { Cook(john) }
val WTom   = Agent { Waiter(tom) }

val Restaurant = Agent { CManel || CAlice || CEmma || CJohn || WTom }
```

4.2 Modelling a client

Clients are modelled by mimicking the Bach description. For instance, the following Bach code

```
Client(id) = LogIn(id); EnjoyAltar(id); LogOut(id)
```

is rewritten in B2Scala as

```
def Client(id:SI_Term) = Agent {
   LogIn(id) * EnjoyAltar(id) * LogOut(id) }
```

Note that the ";" symbol has a particular meaning in Scala and consequently has been rewritten as "*" when it denotes the sequential composition.

Similarly, the recursive procedure *HandleOrders* coded in Bach as

```
HandleOrders(id,t) =
      MakeOrder(id,t); HandleOrders(id,t)
   + CheckOrder(id,t,prepared); HandleOrders(id,t)
   + CheckOrder(id,t,served); HandleOrders(id,t)
   + CheckOrder(id,t,in_progress); HandleOrders(id,t)
```

is rewritten in B2Scala as follows

```
def HandleOrders(id:SI_Term,t:SI_Term): BSC_Agent = {
      Agent {
                  ( MakeOrder(id,t) * HandleOrders(id,t) ) +
                  ( CheckOrder(id,t,prepared) * HandleOrders(id,t) ) +
                  ( CheckOrder(id,t,served) * HandleOrders(id,t) ) +
                  ( CheckOrder(id,t,in_progress) * HandleOrders(id,t) )
      }
}
```

or, using the generalized sum construct, as

```
def HandleOrders(id:SI_Term,t:SI_Term): BSC_Agent = {
      Agent {
                  ( MakeOrder(id,t) * HandleOrders(id,t) ) +
                  ( GSum( List(prepared, served, in_progress), I =>
                      CheckOrder(id,t,I) * HandleOrders(id,t) ) )
      }
}
```

Note that the si-terms `order(id,t,nigiri)`, ..., `order_served(id,t,d)` need to be defined. This is achieved in a similar way to tokens, as follows:

```
case class order(id:SI_Term,table:SI_Term,dish:SI_Term)
        extends SI_Term
case class order_prepared(id:SI_Term,table:SI_Term,dish:SI_Term)
        extends SI_Term
case class order_served(id:SI_Term,table:SI_Term,dish:SI_Term)
        extends SI_Term
```

4.3 Modelling cooks and waiters

Modelling a cook or a waiter is done in a similar way. For instance, the following Bach declarations

```
Cook(id_cook) =
      CookForCustomer(id_cook,manel); Cook(id_cook)
   + CookForCustomer(id_cook,alice); Cook(id_cook)
   + CookForCustomer(id_cook,emma); Cook(id_cook)

Waiter(id_waiter) =
      WaiterForCustomer(id_waiter,manel); Waiter(id_waiter)
   + WaiterForCustomer(id_waiter,alice); Waiter(id_waiter)
   + WaiterForCustomer(id_waiter,emma); Waiter(id_waiter)
```

are coded as follows

```
def Cook(id:SI_Term): BSC_Agent = Agent {
     GSum( List(manel,alice,emma), C =>
               CookForCustomer(id,C) * Cook(id) )
}

def Waiter(id_waiter:SI_Term): BSC_Agent = Agent {
     GSum( List(manel,alice,emma), C =>
               WaiterForCustomer(id_waiter,C) * Waiter(id_waiter) )
}
```

The whole code is available in the joined file `altar.scala` as well as in appendix A.

4.4 Constraining computations

A feature that is not present in the Bach modelling but which is of practical use is to constraint computations to those which meet some properties. An example of such constraints is to restrict computations to those that start by the log in of Manel, Alice and Emma (in this order), that follow by constraining Manel to avoid special dishes at some tables and finally which conclude by the three clients having logged out.

Log in constraints are to be declared as follows

```
val start_login_manel = bf(logged(manel))
val start_login_alice = bf(logged(alice))
val start_login_emma = bf(logged(emma))
```

Restrictions on what Manel can eat at which table is specified as follows:

```
val special_dish_for_manel =
    not(bf(order(manel,t1,nigiri))) and
    not(bf(order(manel,t2,maki))) and
    not(bf(order(manel,t3,temaki))) and
    not(bf(order(manel,t4,beer)))
```

Moreover ending the session may be formalized as follows:

```
val end_session =
    not(bf(logged(manel))) and
    not(bf(logged(alice))) and
    not(bf(logged(emma)))
```

Finally the global constraint is provided by the following F formula, which requires the logins in the appropriate order and which is continued by formula *Fcont* which constraints computations to Manel's dish restrictions until all the three clients have logged out.

```
val F: BHM_Formula = bHM {
    start_login_manel * start_login_alice *
    start_login_emma * Fcont
```

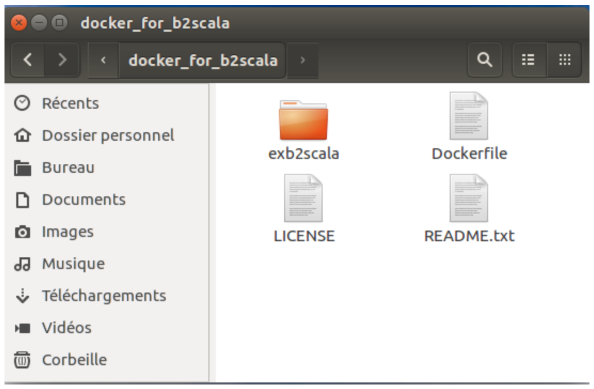

Fig. 3. B2Scala after decompression

```
}

val Fcont: BHM_Formula = bHM {
    (special_dish_for_manel * Fcont) + end_session
}
```

5 Deploying the code in B2Scala

Let us now deploy the code. To that end, we shall use the tool packaged with the docker version; other forms of distribution are to be treated similarly.

Once downloaded and decompressed, the software appears as depicted in Figure 3. Let us go to the `exb2scala` folder and inside it in the `src/main/scala/my_program` folder. It contains a file named `needham_schroeder.scala`, which itself contains the program modelling the Needham-Schroeder protocol. Let us open it with our preferred text editor. The first lines should appear as follows:

```
package exbscala.my_program

import bscala.bsc_data._
import bscala.bsc_agent._
import bscala.bsc_runner._
import bscala.bsc_settings._
import bscala.bsc_formula._

object BSC_modelling extends App {
```

The last line declares the object `BSC_modelling` that is executed when the B2Scala tool is executed. To ease the manipulation, we shall rename it as `BSC_modelling_NDS`, thus getting something like

```
package exbscala.my_program

import bscala.bsc_data._
import bscala.bsc_agent._
import bscala.bsc_runner._
import bscala.bsc_settings._
import bscala.bsc_formula._

// object BSC_modelling extends App {
object BSC_modelling_NDS extends App {
```

Another way of proceeding could be to comment the contents of the whole file.

Let us now add in this directory the file `altar.scala`. It contains the code we just detailed for the Altar restaurant. As one may observe, the first lines are similar

```
package exbscala.my_program

import bscala.bsc_data._
import bscala.bsc_agent._
import bscala.bsc_runner._
import bscala.bsc_settings._
import bscala.bsc_formula._

object BSC_modelling extends App {
```

As the object `BSC_modelling` is declared as we want, we just need to run the docker container. To that end, let us go back to the directory containing the `exb2scala` folder. It should contain a file named `Dockerfile`. It remains to execute the docker by launching successively the two commands:

```
sudo docker build -t 'b2scala_app:1.0' .
sudo docker run b2scala_app:1.0
```

Note that other programs may be introduced in a similar way by commenting previous objects declared as `BSC_modelling` and by declaring in a new file a new object `BSC_modelling` for the application under consideration.

6 Conclusion

As a complementary resource to the article “The B2Scala Tool: integrating Bach in Scala with Security in Mind” written by the same authors, this paper has shown how to code and deploy a new application by using the B2Scala tool. To that end, the application has first been modelled in the coordination language Bach. The resulting code has then be used to derive a B2Scala program.

The authors thank the University of Namur for its support. They also thank the Walloon Region for partial support through the Ariac project (convention 210235) and the CyberExcellence project (convention 2110186).

A Final B2Scala code for the Altar application

The whole code for Altar application is presented in this appendix. It is structured in eight subsections. The header lines are first introduced. They are followed by the description of data, clients, cooks, waiters and of the restaurant. The logical formulae are then proposed. Finally a call is made to an instance of the `BSC_Runner` class for executing the program.

A.1 Header lines

The hearder lines are those introduced above. They aim at importing the appropriate B2Scala classes and at defining the main object of the computation, referred to as `BSC_modelling`.

```
package exbscala.my_program

import bscala.bsc_data._
import bscala.bsc_agent._
import bscala.bsc_runner._
import bscala.bsc_settings._
import bscala.bsc_formula._


object BSC_modelling extends App {
```

A.2 Data

Data need then to be declared. We proceed subsequently by declaring tokens and then structured si-terms. All the data are subclasses of the `SI_Term` declared in the B2Scala implementation.

```
val manel = Token("Manel")
val alice = Token("Alice")
val emma  = Token("Emma")
val john  = Token("John")
val tom   = Token("Tom")

val t1 = Token("table 1")
val t2 = Token("table 2")
val t3 = Token("table 3")
val t4 = Token("table 4")

val nigiri = Token("Nigiri")
val maki = Token("Maki")
val temaki = Token("Temaki")
val beer = Token("Beer")

case class order(id:SI_Term, table:SI_Term, dish:SI_Term)
```

```
        extends SI_Term
case class order_prepared(id:SI_Term, table:SI_Term, dish:SI_Term)
        extends SI_Term
case class order_served(id:SI_Term, table:SI_Term, dish:SI_Term)
        extends SI_Term
case class last_order(id:SI_Term, table:SI_Term, dish:SI_Term)
        extends SI_Term
case class logged(id:SI_Term) extends SI_Term
case class client_sitting_at_table(id:SI_Term, t:SI_Term)
        extends SI_Term

case class t_order(id:SI_Term) extends SI_Term
case class t_check(id:SI_Term) extends SI_Term
```

A.3 The clients

The code for the clients follows that discussed in Section 4.2. We have slightly simplified the application by allowing a client to order only two dishes and to check twice the status of his orders. Moreover several communications are introduced by the `Comm` agents to print messages for actions kept outside the modelling of the application.

```
def Client(id:SI_Term) = Agent {
    LogIn(id) * EnjoyAltar(id) * LogOut(id) }

def LogIn(id:SI_Term) = Agent {
    tell(logged(id)) *
    tell(t_order(id)) * tell(t_order(id)) *
    tell(t_check(id)) * tell(t_check(id)) }

def LogOut(id:SI_Term) = Agent {
    get(logged(id)) }

def EnjoyAltar(id:SI_Term) = Agent {
    GSum( List(t1,t2,t3,t4), T =>
      ( SitAtTable(id,T) * EnjoyAtTable(id,T) ) ) }

def SitAtTable(id:SI_Term, t:SI_Term) = Comm {
    "Client " + id.bsc_toString + " is sitting at " + t.bsc_toString }

def EnjoyAtTable(id:SI_Term, t:SI_Term) = Agent {
    MakeFirstOrder(id,t) * HandleOrders(id,t) * PayOrders(id,t) }

def PayOrders(id:SI_Term, t:SI_Term) = Comm {
    id.bsc_toString + " at " + t.bsc_toString + " just paid " }

def HandleOrders(id:SI_Term, t:SI_Term): BSC_Agent = {
     Agent {
          ( get(t_order(id)) * MakeOrder(id,t) * HandleOrders(id,t) )
+
          ( get(t_check(id)) * CheckOrder(id,t) * HandleOrders(id,t) ) +
          ( nask(t_order(id)) )
      }
}

def MakeFirstOrder(id:SI_Term,t:SI_Term) = Agent {
    get(t_order(id)) * MakeOrder(id,t) }

def MakeOrder(id:SI_Term,t:SI_Term) = Agent {
     GSum( List(nigiri,maki,temaki,beer), D =>
          ( tell(order(id,t,D)) * tell(last_order(id,t,D)) ) ) }

def CheckOrder(id:SI_Term,t:SI_Term) = Agent {
     GSum( List(nigiri,maki,temaki,beer), D =>
            CheckOrderDish(id,t,D) ) }

 def CheckOrderDish(id:SI_Term,t:SI_Term,d:SI_Term) = Agent {
     val in_prepa = d.bsc_toString + " for " + id.bsc_toString +
            " in preparation "
     val prepared = d.bsc_toString + " for " + id.bsc_toString +
            " prepared but not served "
     val served   = d.bsc_toString + " for " + id.bsc_toString +
            " served "

     ( ask(order(id,t,d)) * Comm{ in_prepa } ) +
     ( ask(last_order(id,t,d)) * (
         ( ask(order_prepared(id,t,d)) * Comm{ prepared } ) +
         ( ask(order_served(id,t,d)) * Comm{ served } ) ) )
}
```

A.4 The cook

Cooks are coded following Section 4.3.

```
def Cook(id:SI_Term): BSC_Agent = Agent {
    GSum( List(manel,alice,emma), C =>
        ( CookForCustomer(id,C) * Cook(id) ) ) }

def CookForCustomer(id_cook:SI_Term,id_client:SI_Term) = Agent{
    GSum( List(t1,t2,t3,t4), T =>
        CookForCustomerAtTable(id_cook,id_client,T) ) }

def CookForCustomerAtTable(id_cook:SI_Term,
               id_client:SI_Term,id_table:SI_Term) = Agent {
    GSum( List(nigiri,maki,temaki,beer), O =>
        ( get(order(id_client,id_table,O)) *
          tell(order_prepared(id_client,id_table,O)) ) ) }
```

A.5 The waiter

Waiters are coded similarly by following Section 4.3.

```
def Waiter(id_waiter:SI_Term): BSC_Agent = Agent {
    GSum( List(manel,alice,emma), U =>
       ( WaiterForCustomer(id_waiter,U) * Waiter(id_waiter) ) ) }

def WaiterForCustomer(id_waiter:SI_Term, id_client:SI_Term) = Agent {
    GSum( List(t1,t2,t3,t4), T =>
       WaiterForCustomerAtTable(id_waiter,id_client,T) ) }

def WaiterForCustomerAtTable(id_waiter:SI_Term,
               id_client:SI_Term,id_table:SI_Term) = Agent {
    GSum( List(nigiri,maki,temaki,beer), O =>
        ( get(order_prepared(id_client,id_table,O)) *
          tell(order_served(id_client,id_table,O)) ) ) }
```

A.6 The restaurant

The restaurant is coded as exposed in Section 4.1 by defining all the actors according to their role and putting them in parallel.

```
val CManel = Agent { Client(manel) }
val CAlice = Agent { Client(alice) }
val CEmma  = Agent { Client(emma) }
val CJohn  = Agent { Cook(john) }
val WTom   = Agent { Waiter(tom) }

val Restaurant = Agent { CManel || CAlice || CEmma || CJohn || WTom }
```

A.7 Logic formulae

Computations are constrained by the logic formulae exposed in Section 4.4.

```
val special_dish_for_manel =
  not(bf(order(manel,t1,nigiri))) and
  not(bf(order(manel,t2,maki))) and
  not(bf(order(manel,t3,temaki))) and
  not(bf(order(manel,t4,beer)))

val start_login_manel = bf(logged(manel))
val start_login_alice = bf(logged(alice))
val start_login_emma = bf(logged(emma))

val end_session = not(bf(logged(manel))) and
                  not(bf(logged(alice))) and
                  not(bf(logged(emma)))
```

```
val F: BHM_Formula = bHM {
  start_login_manel * start_login_alice * start_login_emma * Fcont
}

val Fcont: BHM_Formula = bHM {
  (special_dish_for_manel * Fcont) + end_session
}
```

A.8 Running the program

The program is run by taking an instance of the `BSC_runner` class and calling it with the `Restaurant` agent and the `F` formula. Note the last curly bracket which concludes the definition of the `BSC_modelling` object started in Subsection A.1.

```
    val bsc_executor = new BSC_Runner
    bsc_executor.execute(Restaurant,F)

}
```