The BachT Coordination Language and its Implementation in Scala

J.-M. Jacquet and D. Darquennes
Faculty of Computer Science
University of Namur

March 2016

Abstract

This paper presents the coordination language BachT together with a command-line interpreter written in Scala. It is written as a support for the course INFO M451 "Conception d'applications mobiles".

1 The BachT language

We consider in this paper a simplified version of a dialect of Linda, developed at the University of Namur, and named Bach (see [JL07]). This language is based on four primitives for accessing a tuplespace. The *tell(t)* primitive puts an occurrence of the tuple t on the store. The *ask(t)* primitive checks the presence of the tuple t on the store while the *nask(t)* primitive checks its absence. Finally the *get(t)* primitive removes an occurrence of the tuple t on the store. It is worth noting that the *tell* primitive always succeeds whereas the last three primitives suspend as long as the presence/absence of the tuple t is not met. Moreover, the tuple space is seen as a multiset of tuples.

The simplified version we shall use is based on tokens instead of more structured tuples in the aim of stressing the coordination patterns. As a result, to further stress this point, we shall name *store* the corresponding tuple space, actually composed of tokens. The resulting language is subsequently denoted as BachT. It is formally defined as follows.

Definition 1 *Let* Stoken *be an enumerable set, the elements of which are subsequently called tokens and are typically represented by the letters t and u.*

Definition 2 *Define the set $\mathcal{T}$ of the token-based primitives as the set of primitives T_b generated by the following grammar:*

$$T_b \quad ::= \quad tell(t) \mid ask(t) \mid get(t) \mid nask(t)$$

where t represents a token.

2 Transition system

To study the BachT language, a semantics needs to be defined. To that end, we shall use an operational one in the style of Plotkin ([Plo81]), based on a transition system. The configurations to be considered consist of an agent, summarizing the current state of the agents running on the store, and a multi-set of tokens, denoting the current state of the store. In order to express the termination of the computation of an agent, we extend the set of agents by adding a special terminating symbol E that can be seen as a completely computed agent. For uniformity purpose, we abuse the language by qualifying E as an agent.

Figure 1 specifies the transition rules for the primitives of the BachT language. The first rule (T) expresses that an atomic agent tell(t) can be executed in any store σ, and that its action has the effect of adding the token t to the same store. The second rule (A) establishes that an atomic agent ask(t) can be executed in any store σ containing the token t, however leaving the store σ unaltered after its execution. The third rule (G) works similarly to the previous rule (A), but with the difference of retrieving the token t initially present on the store σ after the execution of the agent get(t). Finally, the fourth rule (N) establishes that an atomic agent nask(t) can be executed in any store σ not containing the token t, leaving the store σ unaltered after its execution.

To grasp the complete language, we shall define it by considering as complex agents the statements obtained by combining them by the non-deterministic choice operator "+" (used among others in CCS), the parallel operator, denoted by the "||" symbol and the sequential operator, denoted by the " ; " symbol. To

$$(\mathbf{T}) \qquad \langle\, tell(t) \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \cup \{t\} \,\rangle$$

$$(\mathbf{A}) \qquad \langle\, ask(t) \mid \sigma \cup \{t\} \,\rangle \longrightarrow \langle\, E \mid \sigma \cup \{t\} \,\rangle$$

$$(\mathbf{G}) \qquad \langle\, get(t) \mid \sigma \cup \{t\} \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle$$

$$(\mathbf{N}) \qquad \frac{t \notin \sigma}{\langle\, nask(t) \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle}$$

Figure 1: Transition rules for token-based primitives (BachT)

$$(\mathbf{S}) \qquad \frac{\langle A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle}{\langle A\ ;\ B \mid \sigma \rangle \longrightarrow \langle A'\ ;\ B \mid \sigma' \rangle}$$

$$(\mathbf{P}) \qquad \frac{\langle A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle}{\begin{array}{c}\langle A \mid\mid B \mid \sigma \rangle \longrightarrow \langle A' \mid\mid B \mid \sigma' \rangle \\ \langle B \mid\mid A \mid \sigma \rangle \longrightarrow \langle B \mid\mid A' \mid \sigma' \rangle\end{array}}$$

$$(\mathbf{C}) \qquad \frac{\langle A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle}{\begin{array}{c}\langle A\ +\ B \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle \\ \langle B\ +\ A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle\end{array}}$$

Figure 2: Transition rules for the operators

meet the intuition of the terminating agent E, we shall always rewrite agents of the form $(E; A)$, $(E \mid\mid A)$ and $(A \mid\mid E)$ as A. This is technically achieved by defining the extended set of agents as $\mathcal{L}_B \cup \{E\}$ and by justifying the simplifications by imposing a bimonoid structure. The formal definition is as follows.

Definition 3 *Define the BachT language $\mathcal{L}_B$ as the extended set of agents A generated by the following grammar:*

$$A \quad ::= \quad E \mid T \mid A\ ;\ A \mid A \mid\mid A \mid A\ +\ A$$

where T represents a token-based primitive, E is the special terminating symbol, and where "$\,;\,$", "$\,\mid\mid\,$" and "$\,+\,$" denote the sequential, parallel and choice compositional operators.

Figure 2 details the usual rules for sequential composition, parallel composition, interpreted in an interleaving fashion, and CCS-like choice.

The first rule (S) describes the sequencial composition of two agents A and B. If the agent A makes a first step to A', transforming the store σ in a store σ', then the global composition moves to A' followed by B, with σ' as resulting store. If A terminates successfully, the composition is written $E\ ;\ B$ which is equal to B. Intuitively, this means that the second agent B will only compute after the full execution of the first agent A. Rule (P) describes the parallel composition of two agents A and B, which is computed in an interleaved way. At every moment one of the two agents may compute, but not both synchronously. After a first step of execution of agent A, both compositions $(A' \mid\mid B)$ and $(B \mid\mid A')$ indicate that A' and B must continue their computation in a parallel way. The succesful execution of the global agent $A \mid\mid B$ is reached when all of its components have finished their own computation. Finally transition rule (C) indicates that a choice between two agents A and B must compute like either A or B , but only one of them and that the alternative is chosen in view of the first step.

3 Observables and operational semantics

We are now in a position to define what we want to observe from the computations. Following previous work of the Namur research team on coordination (see eg [BJ99, BJ03a, BJ03b, LJ04, LJ07, LJBB04]), we shall actually take an operational semantics recording the final state of the computations. This is understood as the final store coupled to a mark indicating whether the considered computation is successful or not. Such marks are respectively denoted as δ^+ (for the successful computations) and δ^- (for failed computations).

Definition 4

1. *Let Stoken be a denumerable set, the elements of which are subsequently called tokens and are typically represented by the letters t and u.*
2. *Define the set of stores Sstore as the set of finite multisets with elements from Stoken.*
3. *Let δ^+ and δ^- be two fresh symbols denoting respectively success and failure. Define the set of histories Shist as the cartesian product $Sstore \times \{\delta^+, \delta^-\}$.*
4. *Define the operational semantics $\mathcal{O} : \mathcal{L}_B \rightarrow \mathcal{P}(Shist)$ as the following function: for any agent $A \in \mathcal{L}_B$*

$$\begin{array}{rcl} \mathcal{O}(A) & = & \{(\sigma, \delta^+) : \langle A|\emptyset\rangle \rightarrow^* \langle E|\sigma\rangle\} \\ & & \cup\{(\sigma, \delta^-) : \langle A|\emptyset\rangle \rightarrow^* \langle B|\sigma\rangle \not\rightarrow, B \neq E\} \end{array}$$

where $\rightarrow^*$ denotes the transitive closure of $\rightarrow$ and where $\not\rightarrow$ denotes the absence of a transition step.

4 A command-line interpreter of BachT

4.1 Introduction

In order to allow the reader to experiment with the BachT language but also in the aim of laying down the foundations for student projects, we have developed a command-line interpreter of BachT based on Scala.

Concretely, our goal is to provide the reader with a simple interface allowing him to ask for the run of an agent expressed in BachT and to see the successive contents of the store, allowing him to thereby trace an execution. As an example, we aim at something as follows :

```
> run "(tell(t);get(u)) || (get(t);tell(u))"

{ t }
{ }
{ u }
{ }
Success
```

Note that such a trace corresponds to the only possible execution. Indeed the computation of the above parallel agent necessarily consists of executing in sequence the $tell(t)$ primitive, yielding the store $\{t\}$, then the $get(t)$ primitive, yielding the empty store, then the $tell(u)$ primitive, yielding the store $\{u\}$ and finally the $get(u)$ primitive, yielding the empty store. In doing so, all the four primitives have been successfully computed, which allows to conclude to a successful computation.

Scala seems a good choice for such a task since it combines object-oriented programming and functional programming. Additionnally, the attention paid by its developpers to avoid programmers to write what can be implicitly deduced by the type system offers a very concise way of implementing languages.

More precisely, as noted in [MO06] and [MZ06], from the object-oriented flavor, Scala stays close to conventional languages such as Java and C#, sharing with them most of the basic operators, data types, and control structures. Thanks to this, Scala can seamlessly inter-operate with code written in those two languages. Similarly to Smalltalk's, Scala considers every value as an object, and every operation as a message send, resulting from the invocation of a method. Scala classes and objects can inherit from Java classes and implement Java interfaces. This facilitates the use of Scala code inside Java framework. From the functional point of view, Scala considers that every function is a value.

Scala supports both styles of abstraction for types and for values: parameterization and abstract members. It has a mechanism of mixin-class composition, which is a form of multiple inheritance. Finally Scala allows decomposition of objects by pattern matching.

With this brief explanation of Scala, we are now ready to present our BachT interpreter. It is composed of three main components : a parser of agents, the implementation of the store and finally a simulator which performs the execution of a BachT agent from the execution of basic primitives. For that latter purpose, it is helpful to represent the structure of a parsed agent in internal structures. This is the purpose of the following abstract data, also depicted in figure 3. Technically, an abstract class, called `Expr`, is first introduced. It is refined in three ways :

```
class Expr
case class bacht_ast_empty_agent() extends Expr
case class bacht_ast_primitive(primitive: String, token: String) extends Expr
case class bacht_ast_agent(op: String, agenti: Expr, agentii: Expr) extends Expr
```

Figure 3: The abstract BachT data.scala file

- as a case class `bacht_ast_empty_agent` to represent the empty agent E,
- as a case class `bacht_ast_primitive` to represent a primitive in the form of a pair composed of primitive type (tell, ask, nask, get) and of a token
- as a case class `bacht_ast_agent` to represent a composed agent formed from an operator applied to two sub-agents `agenti` and `agentii`.

4.2 The parser

As exposed in chapter 33 of [MO10], Scala offers facilities to parse languages. The main ingredients to do so are, on the one hand, a library to define parsers, which subsequently basically allows to define the class BachTParsers as inherited from the class RegexParsers, parsing regular expressions, and the possibility of applying functions to the result of strings having been parsed. This is technically achieved in three ways :

- firstly, by considering parsers as functions that consume a reader and yield a parse result and by sequencing these consumptions through the ~ operator. For instance, in

  ```
  "tell(" ~ token ~ ")"
  ```

  Scala tries to read the string `tell(` then what is defined by the function `token` and finally the string `)`. The value returned by this evaluation is formally an instance of the ~ class, which here can be viewed as a pair, or, in our case, as two embedded pairs, namely a triple.

  It is worth noting that repetition can be specified by the `rep` operator. In that case, the value returned is a list.

- secondly, by allowing, through the ^^ construction, to apply a function to the result of a parser, as in

  ```
  ("[a-z][0-9a-zA-Z_]*").r ^^ {_.toString}
  ```

  There the regular expression, obtained by `[a-z][0-9a-zA-Z_]*`, is passed to the `toString` function, which transforms it to a string.

- thirdly, by examining a value through a case statement, which allows to perform a matching, as illustrated as follows :

  ```
  "tell("~token~")" ^^ {
          case _ ~ vtoken ~ _  => bacht_ast_primitive("tell",vtoken) }
  ```

  There a string composed of the string "tell(" followed by the result of the function `token` – which turns out to return the string as just explained above with the regular expression – followed by the string ")" is given as the corresponding threefold sequence of strings to be matched to the expression `_ ~ vtoken ~ _`. It is worth noting that `vtoken` is actually a variable which is matched with the corresponding token in case the matching is successful. The underscores denotes different anonymous variables which are respectively used to match the strings "tell(" and ")". In case the matching is successful, the value after the arrow `=>` is given as a result of the parsing. In the above example, a new case class is returned for the primitive tell with `vtoken` as token. It is worth noting that for expressivity purpose, Scala permits to avoid to explicitly write the new statement (which would have been written in Java for instance).

Writing the parser forces us to specify the priorities of the operators – which we have not done when presenting the language in definition 3. Our choice is quite classical : we stipulate that the sequential composition binds more than the parallel composition which itself binds more than the non-deterministic choice operator. As a result and given the fact that left-recursion is to be avoided by Scala parsers, we define agents as follows :

- an agent is a choice-like agent
- a choice-like agent is a parallel-like agent possibly followed by the choice operator followed by a choice-like agent
- a parallel-like agent is a sequential-like agent possibly followed by the parallel operator followed by a parallel-like agent
- a sequential-like agent is a simple agent possibly followed by the sequential operator followed by a composition-like agent
- a simple agent is either a primitive or an agent enclosed between parentheses.

The code for the parsing of the agents follows directly from this intuition. It is embodied in the functions `compositionChoice`, `compositionPara`, `compositionSeq`, `simpleAgent` and `parenthesizedAgent`. For them, it is worth noting that parsing is applied recursively which forces variables to be instantiated to the result of the parsing of subexpressions. Take the following function as an example :

```
  def compositionPara : Parser[Expr] =
                  compositionSeq~rep(opPara~compositionPara) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii)  => bacht_ast_agent(op,agi,agii) }
```

The first case of the matching `ag ~ List()` instantiates `ag` to the result of a sequential-like agent which is followed by an empty list, namely which is followed by an empty repetition of the parallel operator followed by a parallel-like agent. In the second case of the matching, `agi` represents the parsing of the sequential-like agent (in a similar way `ag` does for the first case) and `agii` represents the parsing of the repetition of the parallel operator followed by a parallel-like agent. As an example, assume the agent $tell(t); get(u))||(get(t); tell(u))$ is parsed by the function `compositionPara`. Then the values for `agi` and `agii` are respectively

```
agi = bacht_ast_agent( ";", bach_ast_primitive("tell", "t"),
                          bach_ast_primitive("get", "u"))
agii = bacht_ast_agent( ";", bach_ast_primitive("get", "t"),
                          bach_ast_primitive("tell", "u"))
```

Consequently, it is sufficient to build the structure `bach_ast_agent(op,agi,agii)` to get the expected internal form :

```
bach_ast_agent("||",
                bacht_ast_agent( ";", bach_ast_primitive("tell", "t"),
                                      bach_ast_primitive("get", "u")),
                agii = bacht_ast_agent( ";", bach_ast_primitive("get", "t"),
                                      bach_ast_primitive("tell", "u")))
```

The code of the parser is presented in figure 4. Besides the functions described for the agents, it consists of a function `token` for parsing tokens and of the definition of three values to represent the three operators (non-deterministic choice operator, parallel operator and sequential operator). As the reader will easily notice, we have defined a token as a string composed of at least a small letter ranging between a and z, possibly followed by a composition of figures between 0 and 9 and/or small or capital letters.

It is practical to define an object instantiating the `BachTParsers` so as to use it directly in the command-line interpreter. This is achieved in the code of figure 5. Two methods are furthermore provided to parse primitives and compositionnally composed agents.

```
class BachTParsers extends RegexParsers {

  def token      : Parser[String] = ("[a-z][0-9a-zA-Z_]*").r ^^ {_.toString}

  val opChoice   : Parser[String] = "+"
  val opPara     : Parser[String] = "||"
  val opSeq      : Parser[String] = ";"

  def primitive : Parser[Expr]    = "tell("~token~")" ^^ {
        case _ ~ vtoken ~ _  => bacht_ast_primitive("tell",vtoken) }   |
                                "ask("~token~")" ^^ {
        case _ ~ vtoken ~ _  => bacht_ast_primitive("ask",vtoken) }    |
                                "get("~token~")" ^^ {
        case _ ~ vtoken ~ _  => bacht_ast_primitive("get",vtoken) }    |
                                "nask("~token~")" ^^ {
        case _ ~ vtoken ~ _  => bacht_ast_primitive("nask",vtoken) }

  def agent = compositionChoice

  def compositionChoice : Parser[Expr] = compositionPara~rep(opChoice~compositionChoice)
      ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii)  => bacht_ast_agent(op,agi,agii) }

  def compositionPara : Parser[Expr] = compositionSeq~rep(opPara~compositionPara) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii)  => bacht_ast_agent(op,agi,agii) }

  def compositionSeq : Parser[Expr] = simpleAgent~rep(opSeq~compositionSeq) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii)  => bacht_ast_agent(op,agi,agii) }

  def simpleAgent : Parser[Expr] = primitive | parenthesizedAgent

  def parenthesizedAgent : Parser[Expr] = "("~>agent<~")"

}
```

Figure 4: Parser: the class `BachTParsers`

```
object BachTSimulParser extends BachTParsers {

  def parse_primitive(prim: String) = parseAll(primitive,prim) match {
        case Success(result, _) => result
        case failure : NoSuccess => scala.sys.error(failure.msg)
  }

  def parse_agent(ag: String) = parseAll(agent,ag) match {
        case Success(result, _) => result
        case failure : NoSuccess => scala.sys.error(failure.msg)
  }

}
```

Figure 5: Parser : the object `BachTSimulParser`

```
import scala.collection.mutable.Map

class BachTStore {

   var theStore = Map[String,Int]()

   def tell(token:String):Boolean = {
      if (theStore.contains(token))
        { theStore(token) = theStore(token) + 1 }
      else
        { theStore = theStore ++ Map(token -> 1) }
      true
   }


   def ask(token:String):Boolean = {
      if (theStore.contains(token))
             if (theStore(token) >= 1) { true }
             else { false }
      else false
   }


   def get(token:String):Boolean = {
      if (theStore.contains(token))
             if (theStore(token) >= 1)
               { theStore(token) = theStore(token) - 1
                 true
               }
             else { false }
      else false
   }


   def nask(token:String):Boolean = {
      if (theStore.contains(token))
             if (theStore(token) >= 1) { false }
             else { true }
      else true
   }


   ...
```

Figure 6: The `BachTStore class`

4.3 The store

The store is implemented as a mutable map in Scala. Initially empty, it is enriched for each told token by an association of this token to a number representing the number of its occurrences on the store. More precisely, the execution of a tell primitive, say `tell(t)` consists in checking whether `t` is already in the map. If it is then the number of occurrences associated with it is simply incremented by one. Otherwise a new association `(t,1)` is added to the map. Dually, the execution of `get(t)` consists in checking whether `t` is in the map and, in this case, of decrementing by one the number of occurrences. In case one of these two conditions is not met then the get primitive cannot be executed. Note that with this simple strategy, a token may appear in the map but with 0 as a number of occurrences associated with it. Hence the implementation has not only to test whether the token appears in the map but also to test whether the associated number of occurrences is more than or equal to one. The ask primitive has a similar behaviour without removing an instance. Finally the nask primitive as an opposite behaviour, succeeding in case the ask primitive fails and failing in case it succeeds.

The code for the primitives is presented in figure 6. Two auxiliary functions are presented in figure 7. The first one, `print_store` takes care of the printing of the contents of the store. The second one, `clear_store` aims at resetting the store to the empty map.

Finally, figure 8 defines the object `bb` of type `BachStore` with a function `reset` as a synonym for the `clear_store` function. Both constructs will be handy for using the command-line simulator.

```
class BachTStore {

  ...

   def print_store {
      println("{")
      for ((t,d) <- theStore)
         println ( t + "(" + theStore(t) + ")" )
      println("}")
   }

   def clear_store {
      theStore = Map[String,Int]()
   }

}
```

Figure 7: The `BachTStore class` continued

```
object bb extends BachStore {

   def reset { clear_store }

}
```

Figure 8: The `bb` object

4.4 The simulator

The simulator consists in repeatedly executing a transition step, as defined by the operational semantics of section 2. In our implementation, this boils down to the definition of function `run_one`, which assumes an agent in a parsed form given and which returns a pair composed of a boolean and an agent in parsed form. The boolean aims at specifying whether a transition step has taken place. In this case, the associated agent consists of the agent obtained by the transition step. Otherwise, failure is reported with the given agent as associated agent.

The function assumes a store. It is given as a parameter of the `BachTSimul` in which `run_one` is defined.

The function is defined inductively on the structure of its argument, say `agent`. If it is a primitive, then the `run_one` function simply consists in executing the primitive on the store. This is technically achieved by the `exec_primitive` function, which actually calls the associated primitive function on the store.

If `agent` is a sequentially composed agent ag_i ; ag_{ii}, then the transition step proceeds by trying to execute the first subagent ag_i. Assume this succeeds and delivers ag' as resulting agent. Then the agent returned is ag' ; ag_{ii} in case ag' is not empty or more simply ag_{ii} in case ag' is empty. Of course, the whole computation fails in case ag_i cannot perform a transition step, namely in case `run_one` applied to ag_i fails.

The code for these two first cases is presented in figure 9.

The cases of the composed agent by a parallel or choice operator are more subtle. Indeed for both cases one should not always favour the first or second subagent. To avoid that behaviour, we randomly assign 0 or 1 to the `branch_choice` variable and depending upon this value we start by evaluating the first or second subagent. In case of failure, we then evaluate the other one and if both fails we report a failure. In case of success for the parallel composition we determine the resulting agent in a similar way to what we did for the sequentially composed agent. For a composition by the choice operator the tried alternative is simply selected. The code for these two cases is reported in figures 10 and 11.

With the one step transition function coded, the simulator mainly consists of a loop which is executed while the current agent is non empty and while failure does not occur. This is materialized in the `bacht_exec_all` function detailed in figure 12. As for the previous component, an object is created to ease the deployment of the command-line interpreter. It defines an `apply` function which essentially consists of executing the function `bacht_exec_all` on the parsed agent. Two other functions `eval` and `run` are used as synonyms for it. This code together with the skeleton of the `BachTSimul` class is presented in figure 13.

4.5 Using the command-line interperter

The complete code is listed on webcampus. It is organized in four files, one for each of the three classes identified above together with one for the definition of the case classes. For the ease of use, they have

```
def run_one(agent: Expr):(Boolean,Expr) = {

   agent match {

      case bacht_ast_primitive(prim,token) =>
         {  if (exec_primitive(prim,token)) { (true,bacht_ast_empty_agent()) }
            else { (false,agent) }
         }

      case bacht_ast_agent(";",ag_i,ag_ii) =>
         {  run_one(ag_i) match
               { case (false,_) => (false,agent)
                 case (true,bacht_ast_empty_agent()) => (true,ag_ii)
                 case (true,ag_cont) => (true,bacht_ast_agent(";",ag_cont,ag_ii))
               }
         }

      ...

   }
}


def exec_primitive(prim:String,token:String):Boolean = {
        prim match
         { case "tell" => bb.tell(token)
           case "ask"  => bb.ask(token)
           case "get"  => bb.get(token)
           case "nask" => bb.nask(token)
         }
   }

}
```

Figure 9: BachT-simulator: primitive and sequential composition

been concatenated to form a single file, called `bacht-cli.scala`, following our aim to write a command-line interpreter for BachT.

Scala offers a very practical mechanism to write methods in a postfix form. As a result, we shall subsequently write `ag run "tell(t)"` instead of `ag.run("tell(t)")`. Thanks to this facility, the command-line interpreter can be used as illustrated in figure 14 to run the agent of subsection 4.1. There, after having launched the scala interpreter, we load the file `bacht-cli.scala` and then evaluate the agent $(tell(t); get(u))||(get(t); tell(u))$, after which we empty the store by evaluating `bb reset`. As the reader will notice, this corresponds to what we aimed at in subsection 4.1. Note that, for the ease of reading, we decorate tuples with their number of occurrences instead of listing these occurrences in sequence.

As another example, we ask to the interpreter to evaluate the following expression:

$$(ask(t)\ ;\ tell(u))\ +\ ((nask(s); ask(t))\ ||\ (tell(t)\ ;\ get(t))$$

In this example, the ask(t) in the first part of the choice cannot be executed, as the store is empty of any token. Only the second part of the choice can be executed. This can provide two different results. Indeed, in the parallel composition, the left part as well as the right part can start: nask(s) can be successful as well as tell(t). If the computation starts with the left part, nask(s), then ask(t) must wait for tell(t) to deposit a token t on the store. Before the execution of the get(t), the procedure verifies if there is a pending primitive for t, which is the case with ask(t). The ask(t) being executed, the get(t) can be invoked to retrieve t. This produces the first trace of execution in Figure 15.

If the computation starts with the right part of the parallel composition, a token is first placed on the store. Before executing get(t), the primitive nask(s) checks for the absence of s, then ask(t) checks for the presence of t, and finally, get(t) retrieves the token t from the store. This produces the second trace of execution in Figure 16.

As a third example more related to the common life, let us consider the following situation of a holydaymaker that hesitates between two destinations for his next holidays: to the Canary Islands or to some Mountains in France. His final choice is dictated by the local weather conditions, namely by the confirmation of a sunny sky in the islands, or by a high fresh snowfall in the mountains. Assume the two conditions are respectively represented by a token s and a token f on the store consulted by the holidaymakers. Moreover let the two possibilities for his choices be represented by tokens c and m. With these tokens, the questioning of the store state about the weather conditions is done with the primitives ask(s) and ask(f). Then the final decision is represented by the following process: $ask(s)\ ;\ tell(c)\ +\ ask(f)\ ;\ tell(m)$.

```
val bach_random_choice = new Random()

def run_one(agent: Expr):(Boolean,Expr) = {

   agent match {

      ...

      case bacht_ast_agent("||",ag_i,ag_ii) =>
         {  var branch_choice = bach_random_choice.nextInt(2)
            if (branch_choice == 0)
              { run_one( ag_i ) match
                   { case (false,_) =>
                       { run_one( ag_ii ) match
                           { case (false,_)
                               => (false,agent)
                             case (true,bacht_ast_empty_agent())
                               => (true,ag_i)
                             case (true,ag_cont)
                               => (true,bacht_ast_agent("||",ag_i,ag_cont))
                           }
                       }
                     case (true,bacht_ast_empty_agent())
                               => (true,ag_ii)
                     case (true,ag_cont)
                               => (true,bacht_ast_agent("||",ag_cont,ag_ii))
                   }
              }
            else
              { run_one( ag_ii ) match
                   { case (false,_) =>
                       { run_one( ag_i ) match
                           { case (false,_)
                               => (false,agent)
                             case (true,bacht_ast_empty_agent())
                               => (true,ag_ii)
                             case (true,ag_cont)
                               => (true,bacht_ast_agent("||",ag_cont,ag_ii))
                           }
                       }
                     case (true,bacht_ast_empty_agent())
                               => (true,ag_i)
                     case (true,ag_cont)
                               => (true,bacht_ast_agent("||",ag_i,ag_cont))
                   }
              }

         }
```

Figure 10: BachT-simulator: parallel composition

```
val bach_random_choice = new Random()

def run_one(agent: Expr):(Boolean,Expr) = {

   agent match {

      ...

      case bacht_ast_agent("+",ag_i,ag_ii) =>
         {  var branch_choice = bach_random_choice.nextInt(2)
            if (branch_choice == 0)
              { run_one( ag_i ) match
                   { case (false,_) =>
                        { run_one( ag_ii ) match
                             { case (false,_) => (false,agent)
                               case (true,bacht_ast_empty_agent())
                                 => (true,bacht_ast_empty_agent())
                               case (true,ag_cont)
                                 => (true,ag_cont)
                             }
                        }
                     case (true,bacht_ast_empty_agent())
                       => (true,bacht_ast_empty_agent())
                     case (true,ag_cont)
                       => (true,ag_cont)
                   }
              }
            else
              { run_one( ag_ii ) match
                   { case (false,_) =>
                        { run_one( ag_i ) match
                             { case (false,_)
                                 => (false,agent)
                               case (true,bacht_ast_empty_agent())
                                 => (true,bacht_ast_empty_agent())
                               case (true,ag_cont)
                                 => (true,ag_cont)
                             }
                        }
                     case (true,bacht_ast_empty_agent())
                       => (true,bacht_ast_empty_agent())
                     case (true,ag_cont)
                       => (true,ag_cont)
                   }
              }
         }
```

Figure 11: BachT-simulator: non-deterministic choice

```
def bacht_exec_all(agent: Expr):Boolean = {

    var failure = false
    var c_agent = agent
    while ( c_agent != bacht_ast_empty_agent() && !failure ) {
       failure = run_one(c_agent) match
            { case (false,_)          => true
              case (true,new_agent)  =>
                 { c_agent = new_agent
                   false
                 }
            }
        bb.print_store
        println("\n")
    }

    if (c_agent == bacht_ast_empty_agent()) {
       println("Success\n")
       true }
    else {
       println("Failure\n")
       false }

}
```

Figure 12: BachT-simulator: main loop

```
import scala.util.Random
import language.postfixOps

class BachTSimul(var bb: BachTStore) {

   val bacht_random_choice = new Random()

   def run_one(agent: Expr):(Boolean,Expr) = { ... }

   def bacht_exec_all(agent: Expr):Boolean = { ... }

   def exec_primitive(prim:String,token:String):Boolean = { ... }

}


object ag extends BachTSimul(bb) {

  def apply(agent: String) {
     val agent_parsed = BachTSimulParser.parse_agent(agent)
     ag.bacht_exec_all(agent_parsed)
  }

  def eval(agent:String) { apply(agent) }
  def run(agent:String) { apply(agent) }

}
```

Figure 13: BachT-simulator: the `BachTSimul` class and the object `ag`

```
dda$scala
Welcome to Scala version 2.11.7 (OpenJDK 64-Bit Server VM, Java 1.6.0_24).
Type in expressions to have them evaluated.
Type :help for more information.

scala> :load bacht-cli.scala
Loading bacht-cli.scala ...
...

scala>ag run "(tell(t);get(u)) || (get(t);tell(u))"
{ t(1) }
{ }
{ u(1) }
{ }
Success

scala>bb reset

scala>
```

Figure 14: Running the BachT command line interperter

```
dda$scala
Welcome to Scala version 2.11.7 (OpenJDK 64-Bit Server VM, Java 1.6.0_24).
Type in expressions to have them evaluated.
Type :help for more information.

scala> :load bacht-cli.scala
Loading bacht-cli.scala ...
...

scala>ag run "(ask(t);tell(u)) + ((nask(s);ask(t)) || (tell(t);get(t)))"
{ }
{ t(1) }
{ t(1) }
{ }
Success

scala>bb reset

scala>
```

Figure 15: Running the BachT command line interperter

```
dda$scala
Welcome to Scala version 2.11.7 (OpenJDK 64-Bit Server VM, Java 1.6.0_24).
Type in expressions to have them evaluated.
Type :help for more information.

scala> :load bacht-cli.scala
Loading bacht-cli.scala ...
...

scala>ag run "(ask(t);tell(u)) + (nask(s);ask(t) || (tell(t);get(t))"
{ t(1) }
{ t(1) }
{ t(1) }
{ }
Success

scala>bb reset

scala>
```

Figure 16: Running the BachT command line interperter

References

[BJ99] A. Brogi and J.-M. Jacquet. On the Expressiveness of Coordination Models. In C. Ciancarini and A. Wolf, editors, *Proceedings of the Third International Conference on Coordination Languages and Models*, volume 1594 of *Lecture Notes in Computer Science*, pages 134–149. Springer-Verlag, Apr 1999.

[BJ03a] A. Brogi and J.-M. Jacquet, editors. *Foclasa 2002, Foundations of Coordination Languages and Software Architectures (Satellite Workshop of CONCUR 2002)*, volume 68, 2003.

[BJ03b] A. Brogi and J.-M. Jacquet. On the Expressiveness of Coordination via Shared Dataspaces. *Science of Computer Programming*, 46(1–2):71 − 98, 2003.

[JL07] J.-M. Jacquet and I. Linden. Coordinating Context-aware Applications in Mobile Ad-hoc Networks. In T. Braun, D. Konstantas, S. Mascolo, and M. Wulff, editors, *Proceedings of the first ERCIM workshop on eMobility*, pages 107–118. The University of Bern, 2007.

[LJ04] I. Linden and J.-M. Jacquet. On the Expressiveness of Absolute-Time Coordination Languages. In R. De Nicola, G.L. Ferrari, and G. Meredith, editors, *Proc. 6th International Conference on Coordination Models and Languages*, volume 2949 of *Lecture Notes in Computer Science*, pages 232–247. Springer, 2004.

[LJ07] I. Linden and J.-M. Jacquet. On the Expressiveness of Timed Coordination via Shared Dataspaces. *Electronical Notes in Theoretical Computer Science*, 180(2):71–89, 2007.

[LJBB04] I. Linden, J.-M. Jacquet, K. De Bosschere, and A. Brogi. On the Expressiveness of Relative-Timed Coordination Models. *Electronical Notes in Theoretical Computer Science*, 97:125–153, 2004.

[MO06] et al M. Odersky. An Overview of the Scala Programming Language - Technical Report - Second Edition. Technical report, Ecole Polytechnique Federale de Lausanne (EPFL), Lausanne, Switzerland, 2006.

[MO10] B. Venners M. Odersky, L. Spoon. *Programming in Scala.* Artima, 2010.

[MZ06] M. Odersky M. Zenger. Independently Extensible Solutions to the Expression Problem - Technical Report. Technical report, Ecole Polytechnique Federale de Lausanne (EPFL), Lausanne, Switzerland, 2006.

[Plo81] G. Plotkin. A Structured Approach to Operational Semantics. (Computer Science Department, Aarhus University, DAIMI FN-19), 1981.