The B2Scala Tool

1. Programming interface

To incorporate **Bach** into Scala, two primary challenges need to be addressed: firstly, the declaration of **data**, and secondly, the declaration of **agents**.

1.1. Data

In relation to data, the trait *si_Term* is created to encompass si-terms. Specific si-terms are subsequently defined as case classes derived from this trait. For example, to operate on *f(1,2)* within one of the primitives *(tell, ask, …)*, the following declaration needs to be established:

```
case class f( x: Int, y: Int ) extends SI_Term
```

Example: The public keys and nonces utilized in the Needham-Schroeder protocol are represented by the following tokens.

```
val pka = Token ( pka )
val pkb = Token ( pkb )
val na  = Token ( na  )
val nb  = Token ( nb  )
```

Encrypted messages are coded by the following si-terms:

```
case class encrypt2( n: SI_Term, k: SI_Term ) extends SI_Term
case class encrypt3( n: SI_Term, x: SI_Term, k: SI_Term ) extends SI_Term
```

1.2. Agents

The fundamental concept for programming agents involves utilizing constructs in the form of:

```
val P = Agent { (tell( f( 1, 2 ) ) + tell( g( 3 ) ) ) || (tell( a ) + tell( b ) ) }
```

To integrate a Bach agent within Scala definitions, the key component is the **Agent** object. This object is defined with an apply method, as shown below:

```
object Agent { def apply(agent: BSC_Agent) = CalledAgent(() => agent) }
```

This object consists of a function that maps a **BSC_Agent** into the Scala structure **CalledAgent**. The latter takes a thunk, a function with no arguments that returns an agent. This lazy evaluation approach is essential for handling recursively defined agents.

The **BSC_Agent** type is a trait equipped with methods for parsing Bach composed agents. Technically, it is defined as follows:

```
trait BSC_Agent {  this: BSC_Agent => def * ( other: => BSC_Agent )  =  ConcatenationAgent( () => this, other _)
                                      def ||( other: => BSC_Agent )  =  ParallelAgent( ()      => this, other _)
                                      def + ( other: => BSC_Agent )  =  ChoiceAgent( ()        => this, other _) }
```

- The **;** symbol is reserved in Scala, sequential composition is expressed using the ***** symbol.
- The composition symbols *****, **||**, and **+** utilize Scala’s postfix operations.

With these definitions, a construct like **tell(t) + tell(u)** is interpreted as the method call **+** to **tell(t)** with **tell(u)** as the argument.

A generalized choice is ponded by the following construct:

```
GSum( L , x  =>  ag( x ) )
```

There **L** is a list, offering choice elements and *x => ag(x)* is a function to which the choice is applied. Consequently, *GSum(List(a,b,c) , x => tell(x) || ask(x))* is equivalent to *(tell(a) || ask(a)) + (tell(b) || ask(b)) + (tell(c) || ask(c))*.

2. Implementation of the Domain Specific Language

The construction of the domain-specific language relies on the same components utilized in the Scan and Anemone workbenches **Scan [1]** ,**Anemone [2]**. These components address two primary considerations: the implementation of the store and the interpretation of agents.

2.1. The store

The store is implemented as a mutable map in Scala, initially devoid of entries. It undergoes augmentation with each structured piece of information relayed, associating it with a numerical value representing the frequency of occurrences within the store. The implementation of basic operations stems directly from this concept.

Example: When executing a **tell** primitive, denoted as **tell(t)**, the system verifies whether **t** already exists in the map. If it does, the associated occurrence count is incremented by one. Alternatively, if **t** is not present, a new association **(t,1)** is added to the map. Conversely, the execution of the **get(t)** primitive involves checking whether **t** exists in the map. If so, the associated occurrence count is decremented by one. If either of these conditions is not met, the **get** primitive cannot be executed.

2.2. The interpretation of agents

Agents are interpreted through the iterative execution of transition steps, primarily defined by the function **run_one**. This function, given an agent in internal form, yields a boolean value and an agent in internal form. The boolean indicates whether a transition step occurred. If true, the associated agent is the result of the transition; otherwise, the failure is reported with the original agent.

- The function is defined inductively based on the structure of the argument **ag**.
- For a primitive **ag**, **run_one** executes the primitive on the store.

In the case of a sequentially composed agent **ag_i ; ag_{ii}**, the transition step attempts to execute the step of the first subagent **ag_i**. If successful, and if **ag'** denotes the result of the first step of **ag** then the whole resulting agent is **ag' ; ag_{ii}** if **ag'** is not empty, or simply **ag_{ii}** if **ag'** is empty. Failure occurs if **ag_i** cannot transition.

Handling agents composed of parallel or choice operators requires a randomized approach. A boolean variable, randomly set to **0** or **1**, determines whether to evaluate the first or second subagent first. In case of failure, the other subagent is evaluated, and if both fail, a failure is reported. For successful parallel composition, the resulting agent is determined similarly to the sequentially composed agent. For a choice operator composition, the tried alternative is selected.

The computation of a procedure call follows a similar approach as expected.

Footnotes

[1] J-M. Jacquet, M. Barkallah, Scan: A Simple Coordination Workbench, in: H. Riis Nielson, E. Tuosto (Eds.), Proceedings of the 21st InternationalConference on Coordination Models and Languages, Vol. 11533 of LectureNotes in Computer Science, Springer, 2019, pp. 75–91.

[2] J-M. Jacquet, M. Barkallah, Anemone: A workbench for the Multi-Bach coordination language, 2021, in: Science of Computer Programming, 202, p.102579.