The B2Scala Tool: integrating Bach in Scala with Security in Mind

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. Although many research efforts have been spent on the theory and implementation of data-based coordination languages, not much effort has been devoted to constructing programming environments to analyze and reason on programs written in these languages. This paper proposes an incarnation in Scala of a Linda-like language, called Bach, developed by the authors. It consists of a Domain Specific Language, internal to Scala, that allows to experiment programs developed in Bach while benefiting from the Scala eco-system, in particular from its type system as well as program fragments developed in Scala. Moreover, we introduce a Hennessy-Milner like logic that allows to restrict the executions of programs to those meeting these logic formulae. Our work is illustrated on the Needham-Schroeder security protocol, for which we manage to rediscover the man-in-the-middle attack first put in evidence by G. Lowe.

Keywords: Coordination · Bach · Scala · Hennessy-Milner Logic · security protocols.

1 Introduction

In the aim of building interactive distributed systems, a clear separation between the interactional and the computational aspects of software components has been advocated by Gelernter and Carriero in [7]. Their claim has been supported by the design of a model, Linda [2], originally presented as a set of inter-agent communication primitives which may be added to almost any programming language. Besides process creation, this set includes primitives for adding, deleting, and testing the presence/absence of data in a shared dataspace.

A number of other models, now referred to as coordination models, have been proposed afterwards. As reported in [4], many of them have been implemented, in some cases as stand alone languages, like Tucson [5], but mostly as API's of conventional languages, accessing tuple spaces through dedicated functions or methods, as in pSpaces [14]. The first approach has the advantage of offering support for a complete algebra-like incarnation of Linda but to the expense of

having to re-implement classical programming constructs that are proposed in conventional languages (like variables, loops, lists, ...). The second approach benefits from the converse characteristics: it is easy to access to classical programming constructs but the abstract control flow that is offered at a process algebraic level, like non-deterministic choice and parallel composition, is to be coded in an ad hoc manner. We turn in this paper to a third approach which enjoys the benefits of the two approaches. It consists in using a domain specific language incarnated inside an existing language. More specifically this paper proposes to embody the Bach coordination language, developed by the authors, inside Scala. In doing so we will take profit of the Scala eco-system while benefiting from all the abstractions offered by the Bach coordination language. A key feature is that we will interpret control flow structures, which we put in good use to restrict computations to those verifying logic formulae. The analysis of the Needham-Schroeder security protocol [17] is used to illustrate our work. Among others, we shall use our coding to highlight the man-in-the-middle attack to it, first put in evidence by G. Lowe [15].

This work is a continuation of previous work on the Scan and Anemone workbenches [9, 10]. It differs by the fact that both Scan and Anemone interpret directly Bach programs. Moreover the PLTL logic they use is different from the logic proposed in this paper.

Our work is also closely linked to the work on Caos [20], which provides a generic tool to implement structured operational semantics and generates intuitive and interactive websites. In particular, one can easily generate a webpage which allows the user to introduce programs in an input text box and to analyze/animate its execution through a collection of widgets. As in our work, this tool is implemented in Scala. However it offers a generic framework which has to be instantiated by defining in Scala the semantics of the language under consideration. In contrast, we take an opposite approach which already offers an implementation of the Bach constructs and in which programmers need to code Bach-like programs in a Scala manner.

The rest of the paper is structured as follows. Section 2 presents the Needham-Schroeder use-case as well as the Bach and Scala languages. Section 3 describes the B2Scala tool, both from the point of view of its usage by programmers and from the implementation point of view. An Hennessy-Milner like logic is proposed in Section 4 together with its effect on reducing executions. Section 5 illustrates how B2Scala coupled to constraint executions can be used to analyze the Needham-Schroeder protocol. Finally Section 6 draws our conclusions and compares our work with related work.

2 Background

2.1 Use-case : the Needham-Schroeder Protocol

The Needham-Schroeder protocol, developed by Roger Needham and Michael Schroeder in 1978 [17], is a pioneering cryptographic solution aimed at ensuring secure authentication and key distribution within network environments. Its

primary objective is to establish a shared session key between two parties, typically referred to as the principal entities, facilitating encrypted communication to safeguard data confidentiality and integrity. The protocol unfolds in a series of steps: initialization, where a client (A) requests access to another client (B) from a trusted server (S), followed by the server's response, which involves authentication, session key generation, and ticket encryption. Subsequently, communication with party B ensues, facilitated by the transmission of the encrypted ticket, along with nonces to ensure freshness. Parties exchange messages encrypted with the session key and incorporate nonces to prevent replay attacks. Mutual authentication is achieved through encrypted messages exchanged between A and B, leveraging the established session key and nonces. Despite its early contributions, the original protocol exhibited vulnerabilities, notably the reflection attack. In response, refined versions have emerged, such as the Needham-Schroeder-Lowe [16] and Otway-Rees protocols [13].

The description of the Needham-Schroeder public key protocol is often slimmed down to the three following actions:

$$\begin{array}{l} Alice \longrightarrow Bob : message(na : a)_{pkb} \\ Bob \longrightarrow Alice : message(na : nb)_{pka} \\ Alice \longrightarrow Bob : message(nb)_{pkb} \end{array}$$

where each transition of the form $X \to Y : m$ represents message m being sent from X to Y. Moreover, the notation m_k represents message m being encrypted with key public key k.

This version assumes that the public keys of Alice and Bob (resp. pka and pkb) are already known to each other. The full version also involves communication between the parties and a trusted server to obtain the public keys.

In this model, Alice initiates the protocol by sending to Bob her nonce na together with her identity a, the whole message being encrypted with Bob's public key pkb. Bob responds by sending to Alice her nonce na together with his nonce nb, the whole message being encrypted this time with Alice's public key pka. Finally Alice sends to Bob his nonce nb, as a proof that a session has been safely made between them. The message is this time encrypted with Bob's public key.

It is worth stressing that, although public keys are know publicly (as the noun suggests), it is only the owners of the corresponding private keys that can decrypt encrypted messages. For instance, the first message sent to Bob can only be decrypted by him.

It is also worth noting that, although sending messages appears as an atomic action in the above description, this is in fact not the case. Messages are transmitted through some medium, let say to simplify the network, and thus are subject to be read or picked up by opponents. This will be illustrated in Section 5 where a more detailed model will be examined.

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

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

Fig. 1. Transition rules for the primitives

2.2 The Bach Coordination Language

Definition of data. Following Linda, the Bach language [6, 11] uses four primitives for manipulating pieces of information : *tell* to put a piece of information on a shared space, *ask* to check its presence, *nask* to check its absence and *get* to check its presence and remove one occurrence. In its simplest version, named BachT, pieces of information consist of atomic tokens and the shared space, called the store, amounts to a multiset of tokens. Although in principle sufficient to code many applications, this is however too elementary in practice to code them easily. To that end, more structured pieces of information are introduced as expressions of the form $f(a_1, \cdots, a_n)$ where f is a functor and a_1, ..., a_n are structured pieces of information. Note that, as usual, tokens are viewed as structured pieces of information having no arguments and are written without parentheses.

Example 1. The nounces used by Alice and Bob in the Needham-Schroeder protocol are coded by the tokens `na` and `nb`, respectively. Similarly, their public keys are coded by the tokens `pka` and `pkb`. A message encrypted by Alice with Bob's public key and providing Alice's nounce with her identity is encoded as the following structured piece of information `encrypt(na, alice, pkb)`.

The set of structured pieces of information is subsequently denoted by $\mathcal{I}$. For short, si-term is used later to denote a structured piece of information.

Agents. The primitives in Bach consist of the `tell`, `ask`, `nask` and `get` primitives already introduced, which take as arguments elements of $\mathcal{I}$. Their execution is formalized by the transition steps of Figure 1. Configurations are taken there as pairs of instructions, for the moment reduced to simple primitives, coupled to the contents of the share space. Following the constraint-like setting in which we have rephrased Linda primitives, the shared space is renamed as *store* and is formally defined as a multiset of si-terms. As a result, rule (T) states that the execution of the $tell(t)$ primitive amounts to enriching the store by an occurrence of t. The E symbol is used in this rule as well as in other rules to denote a terminated computation. Similarly, rules (A) and (G) respectively state that the $ask(t)$ and $get(t)$ primitives check whether t is present on the store with the latter removing one occurrence. Dually, as expressed in rule (N), the primitive $nask(t)$ tests whether t is absent from the store.

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

$$\textbf{(P)}\quad \frac{\langle A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}{\begin{array}{c}\langle A \mid\mid B \mid \sigma\rangle \longrightarrow \langle A' \mid\mid B \mid \sigma'\rangle \\ \langle B \mid\mid A \mid \sigma\rangle \longrightarrow \langle B \mid\mid A' \mid \sigma'\rangle\end{array}} \qquad \textbf{(Pc)}\quad \frac{P(\overline{x}) = A, \langle A[\overline{x}/\overline{u} \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}{\langle P(\overline{u}) \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}$$

Fig. 2. Transition rules for the operators

Primitives can be composed to form more complex agents by using traditional composition operators from concurrency theory: sequential composition, parallel composition and non-deterministic choice. They are respectively denoted by the ; , || and + symbols.

Procedures are defined through the `proc` keyword by associating an agent with a procedure name. As in classical concurrency theory, we assume that the defining agents are guarded, in the sense that any call to a procedure is preceded by the execution of a primitive or can be rewritten in such a form.

Example 2. As an example, the behavior of Alice and Bob can be coded as follows:

```
proc Alice = tell(encrypt(na,a,pkb)); get(encrypt(na,nb,pka));
             tell(encrypt(nb,pkb)).

     Bob = get(encrypt(na,a,pkb); tell(encrypt(na,nb,pka));
           get(encrypt(nb,pkb)).
```

Note that Alice and Bob only tell messages encrypted with the public key of the other and only get messages encrypted with their public key, which simulates their sole use of their private key.

The operational semantics of complex agents is respectively defined through the transition rules of Figure 2. They are quite classical. Rules (S), (P) and (C) provide the usual semantics for sequential, parallel and choice compositions. Rule (Pc) makes procedure call $P(\overline{u})$ behave as the agent A defining procedure P with the formal arguments $\overline{x}$ replaced by the actual ones $\overline{u}$.

In these rules, it is worth noting that we assume agents of the form $(E; A)$, $(E \mid\mid A)$ and $(A \mid\mid E)$ to be rewritten as A.

2.3 The Scala Programming Language

Scala is a statically typed language known for its concise syntax and seamless fusion of object-oriented and functional programming. It stands as a potent tool for a variety of applications [18]. Its static typing ensures code safety, while features like type inference and expressive constructs contribute to readability. Scala's support for functional programming, immutability, and pattern matching renders it apt for scalable applications.

In Scala, variables can be declared as immutable or mutable, as illustrated by the following code snippet.

```
val immutableVariable: Int = 42
var mutableVariable: String = "Hello, Scala!"
```

Methods are introduced with the *def* keyword, can be generic (with type parameters specified in square brackets), can be written in curried form (with multiple parameter lists) and have a return type which is specified at the end of the signature. Here is a simple example with the `add` method.

```
def add(x: Int, y: Int): Int = x + y
```

Methods are typically included in the definition of objects, classes and traits, which act as interfaces in Java. Of particular interest for the implementation of B2Scala is the definition of *case classes* which are classes that automatically define setter, getter, hash and equal methods.

Two main additional features of Scala are worth stressing.

Functions and objects. Functions may be coded by defining objects with an apply function. For instance, if we define

```
object tell {
    def apply(siterm: SI_Term) = TellAgent(siterm)
}

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

then the evaluation of

```
val P = Agent { tell(f(1,2)) }
```

consists first in evaluating *tell* on the si-term $f(1,2)$, which results in the structure $TellAgent(f(1,2))$, and then in evaluating the function *Agent* on this value, which results in the structure $CalledAgent(TellAgent(f(1,2)))$. It is that result which is assigned to P.

Strictness and lazyness. Scala is a strict language that eagerly evaluates expressions. However there are cases in which it is desirable to postpone the evaluation of expressions, for instance to handle recursive definitions of agents. To that end, Scala proposes two basic mechanisms: call-by-name of arguments of functions and so-called thunks. To understand these two concepts, let us consider the following function:

```
def doubleFirst(x: Int, y: Int) = x + x
```

It returns the double of its first argument, regardless of the value of the second one (which is not used). Suppose we want to evaluate $doubleFirst(3+4, 10+20)$. Using an eager strategy, Scala evaluates the two arguments and then computes the double of the first one. As a result $10 + 20$ is computed although the result is not used.

Let us slightly modify the definition of the $doubleFirst$ function, as follows:

```
def doubleFirstLazy(x: Int, y: => Int) = x + x
```

The first argument is passed using the call-by-value strategy. As for the *doubleFirst* function, it is evaluated whenever the function is called. In contrast, the second argument is passed using the call-by-name strategy. Accordingly, it is evaluated when needed and thus in our example not evaluated at all. Such a strategy is particularly useful to code *if-then-else* expressions for which one part only is evaluated according to the evaluation of the condition:

```
def myIf[A](cond: Boolean, onTrue: => A, onFalse: => A): A = {
  if (cond) onTrue else onFalse
}
```

For instance, using the following definition of $myDiv$, the evaluation of $myDiv(0)$ leads only to evaluate 1.

```
def myDiv(x: Float): Float = {
          (myIf[Float](x != 0, 1/x, 1))(x) }
```

However one step further needs to be made to handle recursive expressions that we want to evaluate step by step. In that case, so-called thunks are used. They amount to consider functions requiring no arguments, as in the following definition

```
def myIf[A](cond: Boolean, onTrue: () => A,
                           onFalse: () => A): A = {
      if (cond) onTrue() else onFalse()
}
```

Note that the arguments $onTrue$ and $onFalse$ are now functions taking no arguments and leading to expressions rather than simply expressions. Note also that according to the syntax of Scala $onTrue()$ can also be rewritten as $onTrue\ apply$.

To conclude this point, it is possible to delay the evaluation of val-declared expression by using the *lazy* keyword, such as in

```
lazy val recursiveExpression =  ... recursiveExpression ...
```

3 The B2Scala Tool

3.1 Programming interface

To embody Bach in Scala, two main issues must be tackled: on the one hand, how is data declared, and, on the other hand, how are agents declared.

Data. As regards data, the trait SI_Term is defined to capture si-terms. Concrete si-terms are then defined as case classes of this trait. For instance in order to manipulate $f(1, 2)$ in one of the primitives (tell, ask, ...) the following declaration has to be made:

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

Similarly, tokens can be declared as in

```
case class a() extends SI_Term
```

However that leads to duplicate parentheses everywhere as in $tell(a())$. To avoid that a *Token* class has been defined as a case class of *SI_Term*. It takes as argument a string so that token a can be declared as

```
val a = Token(``a'')
```

Accordingly, a may now be used without parentheses, as in $tell(a)$.

Example 3. As examples, the public keys and nonces used in the Needham-Schroeder protocol are declared as 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
```

Note that Scala does not allow to use the same name for different case classes. We have thus renamed them according to the number of arguments.

Agents. The main idea for programming agents is to employ constructs of the form

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

which encapsulate a Bach agent inside Scala definitions. The *Agent* object is the main ingredient to do so. It is defined as an object with an apply method as follows

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

It thus consists of a function mapping a *BSC_Agent* into the Scala structure *CalledAgent* taking a thunk, which consists of a function taking no argument and returning an agent. As we saw above, this is needed to treat in a lazy way recursively defined agent.

The *BS_Agent* type is in fact a trait equipped with the methods needed to parse 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 _)
}
```

As ; is a reserved symbol in Scala, sequential composition is rewritten with the $*$ symbol.

The definition of the composition symbol $*$, || and + employs Scala facility to postfix operations. Using the above definitions, a construct of the form $tell(t) + tell(u)$ is interpreted as the call of method + to $tell(t)$ with argument $tell(u)$.

It is worth observing that the composition operators take agent arguments with call-by-name and deliver structures using thunks, namely functions without arguments to agents.

It will be useful later to generalize choices such that they offer more than two alternatives according to an index ranging over a set, such as in $\sum_{x \in L} ag(x)$ where $ag(x)$ is an agent parameterized by x. This is obtained in B2Scala by the following construct

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

where `L` is a list.

3.2 Implementation of the Domain Specific Language

The implementation of the domain specific language is based on the same ingredients as those employed in the Scan and Anemone workbenches [9, 10]. They address two main concerns: how is the store implemented and how are agents interpreted.

The store. The store is implemented as a mutable map in Scala. Initially empty, it is enriched for each told structured piece of information by an association of it to a number representing the number of its occurrences on the store. The implementation of the primitives follows directly from this intuition. For instance, 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, in decrementing by one the number of occurrences. In case one of these two conditions is not met then the get primitive cannot be executed.

Interpretation of agents. Agents are interpreted by repeatedly executing transition steps. This boils down to the definition of function `run_one`, which assumes given an agent in an internal form and which returns a pair composed of a boolean and an agent in internal 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 is defined inductively on the structure of its argument, say `ag`. If `ag` is a primitive, then the `run_one` function simply consists in executing the primitive on the store. If `ag` 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 case of an agent composed by a parallel or choice operator is more subtle. Indeed for both cases one should not always favor the first or second subagent. To avoid that behavior, we use a boolean variable, randomly assigned to 0 or 1, 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 computation of a procedure call is performed similarly as one may expect.

4 Constrained executions

The fact that Bach agents are interpreted in the B2Scala tool opens the door to select computations of interest. This is obtained by stating logic formulae to be met. The logic we use is inspired by the Hennessy-Milner logic [8] and the μ-calculus [12]. In view of the coordination context, it rests on basic formulae that assert the presence of si-terms, and, by negation, their absence. For instance `bf(i_running(Alice,Bob))` states that Alice and Bob have initiated a session. Such formulae may be combined with the classical and, or and negation operators. Let us call them bf-formulae and denote them typically with the f, f_1, f_2 symbols. Given the contents of the store, say σ, we shall use $\sigma \models f$ to denote the fact that the bf-formula f holds on the store σ.

Similarly to Hennessy-Milner logic, bHM-formulae are used to specify sequences of properties that have to hold on the sequences of stores produced by computations. They are also defined to offer choices of paths. They are typically denoted as h, h_1, h_2 and are inductively defined by the following grammar:

$$bHM ::= f \mid P \mid h_1 \ + \ h_2 \mid h_1 \ ; \ h_2$$

There f denotes a bf-formula, h_1 and h_2 bHM-formulae and P a variable to be defined by an equation of the form $P = h$. Similarly to agents, we assume that h is guarded in the sense that a bf-formula is requested before variable P is called recursively.

Example 4. As an example, the attack on the Needham-Schroeder protocol may be discovered by finding a computation that obeys to the bHM-formula X defined by

$$X = (\, not(i\_running(Alice, Bob)) \ ; \ X \,) + r\_commit(Alice, Bob)$$

$$(\mathbf{BF}) \qquad \frac{\sigma \models f}{\sigma \vdash f\ [\epsilon]}$$

$$(\mathbf{PF}) \qquad \frac{P = h, \quad \sigma \vdash h\ [h']}{\sigma \vdash P\ [h']}$$

$$(\mathbf{CF}) \qquad \frac{\sigma \vdash h_1\ [h_3]}{\begin{array}{l}\sigma \vdash (h_1 + h_2)\ [h_3] \\ \sigma \vdash (h_2 + h_1)\ [h_3]\end{array}}$$

$$(\mathbf{SF}) \qquad \frac{\sigma \vdash h_1\ [h_3]}{\sigma \vdash (h_1\ ;\ h_2)\ [(h_3\ ;\ h_2)]}$$

Fig. 3. Transition rules for the $\vdash$ relation

$$(\mathbf{ET}) \qquad \frac{\langle A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle, \quad \sigma' \vdash h\ [h']}{\langle A@h \mid \sigma\rangle \hookrightarrow \langle A'@h' \mid \sigma'\rangle}$$

Fig. 4. Extended transition rule

A computation is said to be constrained by a bHM-formula h if the sequence of stores it induces obeys to h. This is defined by means of the auxiliary $\vdash$ relation, itself defined by the rules of Figure 3. Intuitively, the notation $\sigma \vdash h\ [h']$ states that a first bf-formula of h is satisfied on the store σ and that the remaining properties of h' need to be satisfied. Accordingly rule (BF) asserts that if the bf-formula f is satisfied by the store σ then it is also the first formula to be satisfied and nothing remains to be established. The symbol ϵ is used there to denote an empty sequence of bf-formulae. Rule (PF) states that if formula P is defined as h and if a first bf-formula of h is satisfied by σ yielding h' to be satisfied next then so does P with h' to be satisfied subsequently. Finally rules (CF) and (SF) specify the choice and sequential composition of bHM-formulae as one may expect.

Given the $\vdash$ relation, we can define constrained computations by extending the $\rightarrow$ transition relation as the $\hookrightarrow$ relation specified by rule (ET) of Figure 4. Informally this rule states that if, on the one hand, agent A can do a transition from the store σ yielding a new agent A' and a new store σ' and if, on the other hand, a first formula of h is met by σ' yielding h' as a remaining bHM-formula to be established, then agent A can make a constrained transition from store σ and bHM-formula h to agent A' to be computed on store σ' and with respect to bHM-formula h'.

It is worth noting that the encoding in B2Scala is quite easy. On the one hand, basic formulae are defined similarly to Bach primitives through the `bf` function and are combined as primitives are. On the other hand, bHM formulae are defined by the `bHM` function and recursive definitions are handled in the same way as recursive agents.

The interpretation of agents is then made with respect to a bHM-formula. Basically, a step is allowed by `run_one` function if one step can be made according to the bHM-formula, as specified by the $\hookrightarrow$ transition relation. This results in a new agent to be solved together with the continuation of the bHM-formula to be satisfied.

5 The Needham-Schroeder protocol in B2Scala

As an application of the B2Scala tool, let us now code the Needham-Schroeder protocol and exhibit a computation that reflects G. Lowe's attack. The interested reader will find the code, the tool and a video of its usage under the web pages of the authors at the addresses mentioned in [19].

Allowing for an attack requires to introduce an intruder. It is subsequently named Mallory. This being said, the first point to address is to declare nonces and public keys for all the participants of the protocol, namely Alice, Bob and Mallory. This is achieved by the following token declarations:

```
val na = Token("Alice_nonce")
val nb = Token("Bob_nonce")
val nm = Token("Mallory_nonce")

val pka = Token("Alice_public_key")
val pkb = Token("Bob_public_key")
val pkm = Token("Mallory_public_key")
```

It will also be useful later to refer to the three participants, which can be achieved by means of the following token declarations:

```
val alice   = Token("Alice_as_agent")
val bob     = Token("Bob_as_agent")
val mallory = Token("Mallory_as_intruder")
```

To better view who takes which message produced by whom, encrypted messages introduced in section 3, are slightly extended as si-terms of the form $message(Sender, Receiver, Encryted\_Message)$. Moreover, to highlight which message is used in the protocol, we shall subsequently rename encrypted messages as $encrypt_n$, with n the number in the sequence of messages. This leads us to the following declarations:

```
case class encrypt_i(vNonce: SI_Term, vAg: SI_Term,
                     vKey: SI_Term) extends SI_Term
case class encrypt_ii(vNonce: SI_Term, wNonce: SI_Term,
                     vKey: SI_Term) extends SI_Term
case class encrypt_iii(vNonce: SI_Term,
```

```
val Alice = Agent {
  GSum( List(bob,mallory), Y => {
      tell(a_running(Y)) *
      tell( message(alice, Y, encrypt_i(na, alice, public_key(Y))) ) *
      GSum( List(a,nb,nm), WNonce => {
          get( message(Y, alice, encrypt_ii(na,WNonce,pka)) ) *
          tell( message(alice,Y,encrypt_iii(WNonce,public_key(Y))) ) *
          tell( a_commit(Y) )
      })
  })
}
```

Fig. 5. Coding of Alice in B2Scala

```
                          vKey: SI_Term) extends SI_Term
case class message(agS: SI_Term, agR: SI_Term,
                          encM: SI_Term) extends SI_Term
```

Finally, si-terms are introduced to indicate with whom Alice and Bob start and close their sessions. They are declared as follows:

```
case class a_running(vAg: SI_Term) extends SI_Term
case class b_running(vAg: SI_Term) extends SI_Term
case class a_commit(vAg: SI_Term) extends SI_Term
case class b_commit(vAg: SI_Term) extends SI_Term
```

We are now in a position to code the behavior of Alice, Bob and Mallory. Coding Alice's behavior follows the description we gave in Example 2 in Section 2. The code is provided in Figure 5. Although Alice wants to send a first encrypted message to Bob, she can just put her message on the network, hoping that it will reach Bob. The network is simulated here by the store, which leaves room to Mallory to intercept it. As a result, the first action is for Alice to start of a session. Hopefully it is with Bob but, to test for a possible attack, we have to take into account the fact that Mallory can take Bob's place. This is coded by offering a choice between Bob and Mallory by the `GSum([bob,mallory])(...)` construct. Calling this actor Y, Alice's first action is to tell the initialization of the session with Y, thanks to the `a_running(Y)` si-term being told and then to tell the first encrypted message with her nonce, her identity and the public key of Y. The sender and receiver of this message are respectively *Alice* and Y. Then Alice waits for a second encrypted message with her nonce and what she hopes to be Bob's nonce, this message being encrypted by her public key. As the second nonce is unknown a new choice is offered with the *WNonce* si-term. Finally, Alice sends the third encrypted message with this nonce, encoded with the public key of Y and terminates the session by telling the `a_commit(Y)` si-term. It is worth noting that `public_key(Y)` consists of a call to a Scala function that returns the public key corresponding to the Y argument.

```
val Bob = Agent {
   GSum( List(alice,mallory),  Y => {
       tell(b_running(Y)) *
       GSum( List(alice,mallory), VAg => {
           get( message(Y,bob,encrypt_i(na,VAg,pkb)) ) *
           tell( message(bob,Y,encrypt_ii(na,nb,public_key(Y))) ) *
           get( message(Y,bob,encrypt_iii(nb,pkb)) ) *
           tell( b_commit(VAg) )
       })
   })
}
```

Fig. 6. Coding of Alice in B2Scala

```
lazy val Mallory:BSC_Agent = Agent {

   ( GSum( List(na,nb,nm), VNonce => {
       GSum( List(alice,bob), VAg => {
        GSum( List(pka,pkb,pkm), VPK => {
           get( message(alice,mallory,encrypt_i(VNonce,VAg,VPK)) ) *
           ( if ( VPK == pkm) {
                tell( message(mallory,bob,encrypt_i(VNonce,VAg,pkb)) )
             } else {
                tell( message(mallory,bob,encrypt_i(VNonce,VAg,VPK)) )
             } ) * Mallory
        })
       })
       }) )      + ...
```

Fig. 7. Coding of Mallory in B2Scala

Coding Bob's behavior proceeds in a dual manner. This time the coding has to take into account that Mallory can have taken Alice's place. Hence the first choice `GSum([alice,mallory])(...)` with Y denoting the sender of the message. Moreover, the identity of the agent in the first message being got can be different from Y. A second choice `GSum([alice,mallory])(...)` results from that. The whole agent is given in Figure 6.

As an intruder, Mallory gets and tells messages from Alice and Bob, possibly modifying some parts in case the messages are encrypted with his public key. This applies for the three kinds of message sent/received by Alice and Bob. Figure 7 provides the code for the first message. It presents three `GSum` choices resulting from the three unknown arguments `VNonce`, `VAg`, `VPK` of the message. In all the cases, Bob's attitude is to get the message and to resend it, by modifying the public key if he can decrypt the message, namely if `VPK` is his public key.

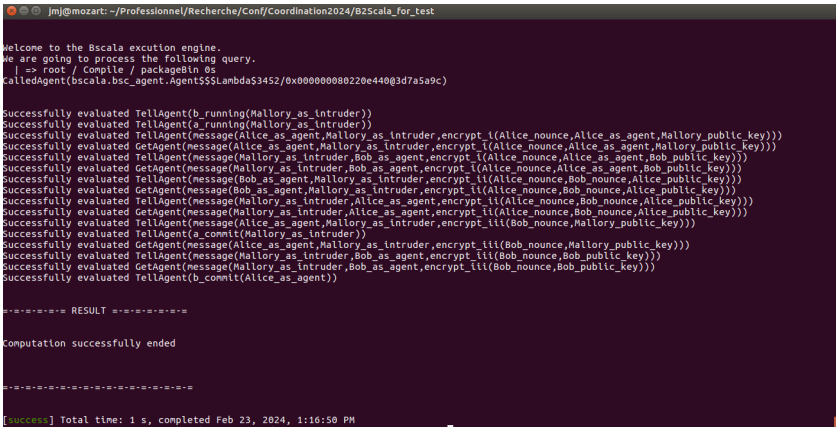

Fig. 8. Screenshot of the computation

To conclude the encoding of the protocol in B2Scala, a bHM-formula F is specified, on the one hand, by excluding a session starting between Bob and Alice and, on the other hand, by requiring the end of the session by Bob with Alice. These two requirements are obtained through the bf-formulae `inproper_init` and `end_session`, as specified below:

```
val inproper_init = not( bf(a_running(bob)) or bf(b_running(alice)) )
val end_session = bf(b_commit(alice))
```

Formula F is then coded recursively by requiring F after a step meeting `inproper_init` and by stopping the computation once a step is done that makes `end_session` holds. This is specified as follows.

```
val F = bHM { (inproper_init * F) + end_session }
```

Computations are started by invoking the following Scala instructions

```
val Protocol = Agent { Alice || Bob || Mallory }

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

The result is given in Figure 8. As claimed it produces G. Lowe's attack. A key ingredient for this is that imposing `inproper_init` to hold forces the first choice in Alice's code and Bob's code to be made such that Y takes Mallory as value.

6 Conclusion

As a complementary line to previous work [9, 10], this paper has proposed an incarnation of the coordination language Bach in Scala, in the form of an internal

Domain Specific Language, named B2Scala. It has also proposed an Hennessy-Milner like logic that allows for constraining executions. The Needham-Schroeder protocol has been modeled with our proposal to illustrate its interest in practice.

Our work is also closely linked to the work on Caos [20], which provides, by using Scala, a generic tool to implement structured operational semantics and to generate intuitive and interactive websites. In practice, one has however to define the semantics of the language under consideration by using Scala. In contrast, we take an opposite approach which already offers an implementation of the Bach constructs and in which programmers need to code Bach-like programs in a Scala manner. Moreover we propose a logic to constraint executions, which is not proposed in [20].

Scafi ([3]) is another research effort to integrate a coordination language in Scala. It targets a different line of research in the coordination community by being focus on aggregate computing. Moreover, to the best of our knowledge, no support for constrained executions is proposed.

Finally the Needham-Schroeder protocol has been modeled in process algebras. In [15] the author uses CSP and its associated FDR tool to produce an attack on the protocol. This analysis has been complemented in [1] by using the mCRL process algebra and its associated model checker. Our work differs by using a process algebra of a different nature. Indeed the Bach coordination languages rests on asynchronous communication which happens to a shared space. This allows to naturally model messages being put on the network as si-terms told on the store. Similarly the action of an intruder is very intuitively modeled by getting si-terms. In contrast, [15] and [1] use synchronous communication which does not naturally introduce the network as a communication medium and which technically forces them to model the intruder by duplicating Alice and Bob's send and receive actions by intercept and fake messages.

7 Acknowledgment

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).

References

1. Blom, S., Groote, J., Mauw, S., Serebrenik, A.: Analysing the BKE-security Protocol with μCRL. In: Ulidowski, I. (ed.) Proceedings of the 6th AMAST Workshop on Real-Time Systems. Electronic Notes in Theoretical Computer Science, vol. 139, pp. 49–90. Elsevier (2004)
2. Carriero, N., Gelernter, D.: Linda in Context. Communications of the ACM **32**(4), 444–458 (1989)
3. Casadei, R., Viroli, M., Aguzzi, G., Pianini, D.: ScaFi: A Scala DSL and Toolkit for Aggregate Programming. SoftwareX **20**, 101248 (2022)

4. Ciatto, G., Mariani, S., Serugendo, G.D.M., Louvel, M., Omicini, A., Zambonelli, F.: Twenty Years of Coordination Technologies: COORDINATION Contribution to the State of Art. Journal of Logical and Algebraic Methods in Programming **113**, 100531 (2020)
5. Cremonini, M., Omicini, A., Zambonelli, F.: Coordination and Access Control in Open Distributed Agent Systems: The TuCSoN Approach. In: Porto, A., Roman, G. (eds.) Proceedings of 4th International Conference on Coordination Languages and Models. Lecture Notes in Computer Science, vol. 1906, pp. 99–114. Springer (2000)
6. Darquennes, D., Jacquet, J.M., Linden, I.: On Multiplicities in Tuple-Based Coordination Languages: The Bach Family of Languages and Its Expressiveness Study. In: Serugendo, G.D.M., Loreti, M. (eds.) Proceedings of the 20th International Conference on Coordination Models and Languages. Lecture Notes in Computer Science, vol. 10852, pp. 81–109. Springer (2018)
7. Gelernter, D., Carriero, N.: Coordination Languages and Their Significance. Communications of the ACM **35**(2), 97–107 (1992)
8. Hennessy, M., Milner, R.: On Observing Nondeterminism and Concurrency. In: de Bakker, J., van Leeuwen, J. (eds.) Proceedings of the International Conference on Automata, Languages and Programming. Lecture Notes in Computer Science, vol. 85, p. 299–309. Springer (1980)
9. Jacquet, J.M., Barkallah, M.: Scan: A Simple Coordination Workbench. In: Nielson, H.R., Tuosto, E. (eds.) Proceedings of the 21st International Conference on Coordination Models and Languages. Lecture Notes in Computer Science, vol. 11533, pp. 75–91. Springer (2019)
10. Jacquet, J.M., Barkallah, M.: Anemone: A workbench for the Multi-Bach Coordination Language. Science of Computer Programming **202**, 102579 (2021)
11. Jacquet, J.M., Linden, I.: Coordinating Context-aware Applications in Mobile Adhoc Networks. In: Braun, T., Konstantas, D., Mascolo, S., Wulff, M. (eds.) Proceedings of the first ERCIM workshop on eMobility. pp. 107–118. The University of Bern (2007)
12. Kozen, D.: Results on the Propositional mu-Calculus. Theoretical Computer Science **27**, 333–354 (1983)
13. Liu, K., Ye, J., Wang, Y.: The Security Analysis on Otway-Rees Protocol Based on BAN Logic. In: Proceedings of the Fourth International Conference on Computational and Information Sciences. pp. 341–344 (2012)
14. Loreti, M., Lafuente, A.L.: Programming with Spaces (2024), `https://github.com/pSpaces/Programming-with-Spaces/blob/master/README.md`
15. Lowe, G.: Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR. In: Margaria, T., Steffen, B. (eds.) Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 1055, p. 147–166. Springer (1996)
16. Lowe, G.: Breaking and fixing the Needham-Schroeder Public-Key Protocol using FDR. In: Tiziana, M., Steffen, B. (eds.) Proceedings of the Internation Conference on Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 1055, pp. 147–166. Springer (1996)
17. Needham, R.M., Schroeder, M.D.: Using Encryption for Authentication in Large Networks of Computers. Communication of the ACM **21**, 993–999 (1978)
18. Odersky, M., Spoon, L., Venners, B.: Programming in Scala, A comprehensive step-by-step guide. Artemis (2016)

19. Ouardi, D., Barkallah, M., Jacquet, J.M.: Coding and Breaking the Needham-Schroeder Protocol using B2Scala (2024), `https://staff.info.unamur.be/douardi/Coordination24 or https://staff.info.unamur.be/mbarkall/Coordination24 or https://staff.info.unamur.be/jmj/Coordination24`, created on February 26th, 2024
20. Proença, J., Edixhoven, L.: Caos: A Reusable Scala Web Animator of Operational Semantics. In: Jongmans, S., A.Lopes (eds.) Proceedings of the 25th International Conference on Coordination Models and Languages. Lecture Notes in Computer Science, vol. 13908, pp. 163–171. Springer (2023)