Implementing Spi Calculus using Nominal techniques

Temesghen Kahsai[1]*, Marino Miculan[2]

[1] Department of Computer Science, Swansea University, UK. csteme@swan.ac.uk
[2] DiMI, University of Udine, Italy. miculan@dimi.uniud.it

Abstract. The aim of this work is to obtain an interactive proof environment based on Isabelle/HOL for reasoning formally about cryptographic protocols, expressed as processes of the spi calculus (a π-calculus with cryptographic primitives). To this end, we formalise syntax, semantics, and *hedged bisimulation*, an environment-sensitive bisimulation which can be used for proving security properties of protocols. In order to deal smoothly with binding operators and reason up-to α-equivalence of bound names, we adopt the new *Nominal datatype* package. This simplifies both the encoding, and the formal proofs, which turn out to correspond closely to "manual proofs".

1 Introduction

It is well known that proving security properties of communication protocols is difficult and error-prone. Since Paulson's seminal work [19], *interactive (semi-automatised) proof assistants* (such as Isabelle [18]) have been recognised as valid aid tools to this end. In principle, (a model of) the system can be formalised in a proof assistant, the security property can be formally stated as a "theorem", and a formal proof can be carried out interactively by the user, and checked by the environment. Sophisticated semi-automatic proof search tactics may simplify some derivation steps. This approach should be seen as complementary, and not in contrast, to the many fully automatised tools, such as those based on (symbolic) model checking, SAT solvers, etc.; see e.g. [16,10,3,6]. Actually, both approaches have strong and weak points: automatic tools are quite successful on particular finite-state systems, but they naturally suffer of state-explosion problems and usually cannot be used for proving general properties. On the other hand, interactive tools may be tedious and slow to use, but in principle can be used for proving any valid property. In fact, the best solution would be *integrated proof assistants*, that is semi-automatic interactive tools where the verification of decidable subgoals can be left to automatic proof search tools; see e.g. [24,17] for some application of this approach to model checking problems.

Among the many formal models for security, a particularly successful one is the *spi calculus* [2], a process calculus intended to describe and reason about the

* The author is supported by EPSRC under the grant EP/D037212/1.

behaviour of cryptographic protocols; security properties can be expressed rigorously as statements of *behavioural equivalence* between processes. These equivalences can be characterised by *environmental (i.e. context-sensitive) bisimulations*, where the environment keeps track of the knowledge accumulated by an attacker that observes the evolution of the protocols. Some fully automatised approaches for deciding these equivalences have been proposed (notably [14,8,6]), but their applicability is limited to small decidable fragments.

In this paper, we intend to complement these tools with an interactive proof assistant for spi calculus. More precisely, we give a formalisation of the spi calculus in the proof assistant Isabelle/HOL, using the recently developed *Nominal package*. We formalise syntax, operational semantics, and two environmental bisimulations. In our opinion this work is useful for many reasons. First, we readily obtain an interactive environment which can be effectively used for proving (formally and error-free) security properties of protocols expressed as spi calculus processes, as well as meta-theoretic results about the spi calculus itself. Secondly, the rigorous encoding of a calculus in the metalanguage of a logical framework is normative, since it forces to spell out in full detail all aspects of the calculus, thus giving the possibility of identifying and fixing problematic issues which may be overlooked on paper. Thirdly, this formalisation can be used for integrating automatised tactics and proof search strategies, as described above. Fourthly, this is the first application of these techniques to environmental bisimulations, and this study can be ported to other environmental bisimulations such as those recently studied for higher-order languages in [21]. Finally, extensive case studies like this are useful test-beds for state-of-art and still under development logical frameworks and proof assistants.

Regarding this last aspect, one feature of our work is that we use Isabelle/HOL extended with the *Nominal package* [22] (or Isabelle/Nominal for short). The Nominal package implements in Isabelle/HOL the ideas of Nominal Logic, introduced in the seminal works by Gabbay and Pitts [11,20]. These techniques aim to simplify the manipulation and reasoning of data with binding operators, by automatically identifying terms up to α-equivalence of bound names. This aspect is fundamental in the spi calculus, where (bound) variables and names are crucially used for representing channels, keys, nonces, etc. Of course other encoding methodologies are possible in principle, but the hassle of dealing explicitly with different representations of the same process would hinder the usability of the resulting encoding in interactive proofs.

Although it is still under development, the Nominal package is already quite usable; see [4] for an extensive implementation of the theory of π-calculus in Isabelle/Nominal, which have been inspiration for the present work. In fact, we think that case studies like the present work can give useful insights for further improvements of the Nominal package, thanks to the several distinguishing features of spi calculus with respect to π-calculus (such as message passing and peculiar context-sensitive bisimulations).

Synopsis In Section 2 we recall the spi calculus: its syntax, semantics and hedged bisimilarity. In Section 3 we give a brief introduction to the Nominal package. In

Section 4 we describe the implementation of the spi calculus in Isabelle/Nominal, together with an example of a formal proof. Section 5 concludes the paper with some future and related work.

The complete Isabelle source code, with further examples and an encoding of "framed bisimulation" (which we cannot report in this paper due to lack of space), can be found at `http://www.cs.swan.ac.uk/~csteme/SpiInIsabelle/`.

2 Spi Calculus

The spi calculus [2] is a process calculus extending π-calculus [15] with primitives for describing and reasoning about the behaviour of cryptographic protocols. Security properties, such as secrecy, authenticity (and also authentication via a reduction to secrecy [5]), can be expressed as statements of behavioral equivalence. In this section we recall the syntax and the semantics of the shared-key spi-calculus, following [9].[3]

Syntax We assume an infinite set of *names* $\mathcal{N}$, ranging from $a, b, c, \dots k, l, m, n$ and x, y, z. The set of *expressions* is defined by the following grammar:

$$\begin{array}{rll} \zeta, \eta ::= a \mid \mathsf{E}_\eta(\zeta) \mid \mathsf{D}_\eta(\zeta) & & \text{expressions } \mathcal{E} \\ \delta ::= a \mid \mathsf{E}_\delta(\delta) & & \text{decryption-free expressions } \mathcal{D} \\ M, N ::= a \mid \mathsf{E}_k(M) & & \text{messages } \mathcal{M} \end{array}$$

$\mathsf{E}_\eta(\zeta)$ represents the cipher-text obtained by encrypting the expression ζ with the expression η as key, using some given (perfect) shared-key cryptosystem. $\mathsf{D}_\eta(\zeta)$ represents the decryption of ζ using the key η, if successful.

The *guards* $\mathcal{G}$ are defined by the following grammar:

$$\phi, \psi ::= tt \mid \phi \wedge \psi \mid \neg\phi \mid let\ z = \zeta\ in\ \phi \mid is\_Name(\delta) \mid [\delta = \delta]$$

Decryption constructors can occur only in the ζ of the "let" construct: the formula $let\ z = \zeta\ in\ \phi$ evaluates the expression ζ, and if evaluation succeeds (i.e., ζ contains no encrypted expressions which cannot be decrypted), binds its value to z and evaluates ϕ. Equality and name tests can be performed only on decryption-free expressions; this means that before comparing two expressions, or checking whether an expression is a name, all pending decryptions have to be solved.

Finally, the set of *processes* is defined as follows:

$$\begin{array}{rl} P, Q, R ::= & \mathbf{0} \mid \overline{\delta}\langle N\rangle.P \mid \delta(x).P \mid P + P \mid P \mid Q \\ & \mid (\nu n)P \mid !P \mid \phi P \mid let\ x = \zeta\ in\ P \end{array}$$

The name n is bound in $(\nu n)P$, and x is bound in P by $let\ x = \zeta\ in\ P$ and $M(x).P$. For an intuitive description of spi calculus processes, see [1]. Some

[3] Of course more expressive languages, e.g. with primitives for public-key cryptography, can be dealt with easily; for the sake of simplicity, in this paper we prefer to consider a simpler language.

syntactic conventions: $fn(P)$ indicates the sets of names free in process P. A *concretion* is an expression of the form $\nu m_1, \ldots, m_k \langle M \rangle P$, where M is a term, P is a process, $k \geq 0$, and the names $m_1, \ldots, m_k$ are bound in M and P. An *agent* is a process or a concretion. The meta-variables A and B range over arbitrary agents, and $fn(A)$ stands for the sets of free names of an agent A. An *action* is either a name a with a message M (representing input), or a co-name $\overline{m}$ (representing output) or the distinguished *silent action* τ.

Operational semantics Expressions and boolean guards are evaluated by two functions $[\![.]\!] : \mathcal{E} \to \mathcal{M} \cup \{\bot\}$, $[\![.]\!] : \mathcal{G} \to \{tt, \mathit{ff}\}$. The behaviour of processes is described by the *commitment relation* ($P \xrightarrow{\alpha} A$), where P is a process, α is an action, and A is an agent. See [9] for a complete description of these two notions.

Environment-sensitive bisimulations Bisimulations for spi calculus are based on the idea of an environment observing a pair of processes, trying to distinguish one from the other using the knowledge accumulated during the evolution of these processes. The environment typically observes the transitions derived from the operational semantics of the processes. *Hedged bisimulation* is an improved form of environment-sensitive bisimulation. It has been introduced in [9], following ideas of [7], in order to highlight the differences between different environment-sensitive bisimulation. The main idea of hedges is to keep track of the correspondence of different names which play the same role.

Definition 1. *A* hedge *is a finite subset of $\mathcal{M} \times \mathcal{M}$. $\mathcal{H}$ denotes the set of all hedges.*

A hedge h is consistent iff for $(M, N) \in h$:

1. *$M \in \mathcal{N}$ iff $N \in \mathcal{N}$;*
2. *if $(M', N') \in h$ then $M = M'$ iff $N = N'$*
3. *if $M = \mathsf{E}_{M_2}(M_1)$ and $N = \mathsf{E}_{N_2}(N_1)$ then $M_2 \notin \pi_1(h)$ and $N \notin \pi_2(h)$, where M_1, M_2, N_1 and N_2 are expressions (and $\pi_1(h)$ is the first projection).*

The *synthesis* $S(.)$ of a hedge is defined inductively. We write $h \vdash M \leftrightarrow N$ for $(M, N) \in S(h)$. Intuitively, $h \vdash M \leftrightarrow N$ means that, using the knowledge h, the environment is unable to distinguish two processes P and Q, if the first emits M and the second N. The rules for synthesis are the following:

$$(\textsc{Synth. hedge})\frac{(m,n) \in h}{h \vdash m \leftrightarrow n} \qquad (\textsc{Synth. enc})\frac{h \vdash M \leftrightarrow M' \quad h \vdash N \leftrightarrow N'}{h \vdash \mathsf{E}_N(M) \leftrightarrow \mathsf{E}_{N'}(M')}$$

The *analysis* $A(h)$ is the smallest subset of $\mathcal{M} \times \mathcal{M}$ containing h and satisfying the following rule:

$$\frac{(\mathsf{E}_a(M), \mathsf{E}_b(N)) \in A(h) \quad (a,b) \in A(h)}{(M,N) \in A(h)}$$

The *irreducibles* $I(h)$ of a hedge h is $I(h) \triangleq A(h) \backslash \{(\mathsf{E}_a(M), \mathsf{E}_b(N)) \mid (a,b) \in A(h) \wedge M, N \in \mathcal{N}\}$. Intuitively, the analysis decrypts as much as possible pairs of messages using pairs of names that are considered equivalent by the environment; irreducibles terms are those which cannot be decrypted further.

Definition 2 (Hedged simulation). *A* hedged relation $\mathcal{R}$ *is a subset of* $\mathcal{H} \times P \times P$. *We say that* $\mathcal{R}$ *is consistent if* $h \vdash P\mathcal{R}Q$ *implies that* h *is consistent.*

A consistent hedged relation $\mathcal{R}$ *is a* hedged simulation *if whenever* $h \vdash P\mathcal{R}Q$ *the following conditions hold:*

1. *If* $P \xrightarrow{\tau} P'$ *then there exists a process* Q' *such that* $Q \Longrightarrow Q'$ *and* $h \vdash P'\mathcal{R}Q'$.
2. *If* $P \xrightarrow{a\ M} P'$, $h \vdash a \leftrightarrow b$, B *is a finite set of names disjoint from* $fn(P) \cup fn(Q) \cup fn(h)$, *and* N *is a message such that* $h \cup Id_B \vdash M \leftrightarrow N$, *then there exists* Q' *such that* $Q \overset{b\ N}{\Longrightarrow} Q'$ *and* $h \cup ID_B \vdash P'\mathcal{R}Q'$
3. *If* $P \xrightarrow{\bar{a}} (\nu\vec{c})\langle M\rangle P'$, $h \vdash a \leftrightarrow b$ *and* $\vec{c}$ *is disjoint from* $fn(P) \cup fn(\pi_1(h))$, *then there exists* $Q', N, \vec{d}$ *with* $\vec{d}$ *disjoint from* $fn(Q) \cup fn(\pi_2(h))$ *such that* $Q \overset{\bar{b}}{\Longrightarrow} (\nu\vec{d})\langle N\rangle Q'$ *and* $I(h \cup \{(M, N)\}) \vdash P'\mathcal{R}Q'$.

$\mathcal{R}$ is a *hedged bisimulation* if both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are *hedged simulations.* The *hedged bisimilarity* is the greatest hedged bisimulation, and it is denoted by $\sim_h$.

It turns out that hedged bisimilarity corresponds to barbed equivalence [9].

3 Isabelle/Nominal

The *Nominal package* [22] for Isabelle/HOL aims to provide a framework for reasoning about process calculi and programming languages with binding operators in a convenient way, so that formal proofs should be easy to carry out as informal “pencil-and-paper” proofs. The work is based on the nominal logic [20]; the main technical novelty introduced by Urban et al. [22] is that the construction for α-equivalent terms is done without adding any axiom to the Isabelle/HOL logic; therefore the theory is implemented just as a package of Isabelle/HOL, without the need of changing the underlying proof assistant.

A *nominal datatype* definition is like an ordinary datatype, but it explicitly tags the binding occurrences of names. For instance, in the syntax of the usual untyped λ-calculus the notation `<<name>>term` stands for “a `term` abstracted over `name`”, that is, with a name bound in `term`. The package automatically provide the α-equivalence between terms; e.g., `(lam x (var x))` and `(lam y (var y))` are equal. Moreover, the package generates automatically powerful induction rules over terms up-to α-equivalence (among other useful properties). This saves the user much hassle in large proofs.

The core of the nominal logic relies on the notion of *name swapping.* Atoms (i.e., names) are manipulated not by renaming substitutions but by permutations (bijective mappings from atoms to atoms). In the Nominal package, permutations are represented as finite lists of atom swappings (*i.e*, pairs of atoms). The operation of permutation applies to all names in a term, including the binding and bound occurrences: if T be a term, and a and b are names then $(a\ b) \bullet T$ denotes the term where all instances of a in T becomes b and vice versa. For further details, the reader can refer to `http://isabelle.in.tum.de/nominal/`.

4 Encoding spi calculus in Isabelle/Nominal

In this section, we describe the implementation of spi calculus in the general purpose proof assistant Isabelle [18], using its instantiation HOL-Nominal implementing higher-order intuitionistic logic and including the Nominal package. For improving readability of theories and proofs, we use *Isar* [23] (Intelligible semi-automatized reasoning), with occasionally some syntactic sugar.

4.1 Implementation of syntax and semantics

We introduce one type of nominal atoms *name*, which will be used in binders. *Expressions*, *decryption free expressions*, *messages* and *processes* are declared as nominal datatypes; actually, only processes have binders, but in the current version of the Nominal package, building nominal datatypes over normal datatypes is not easy.[4]

Due to the use of Isabelle syntax, there may be a slight change of notation from Section 2, but the rationale will be clear.

```
nominal_datatype expr = Name name    | Sk_enc expr expr    | Sk_dec expr expr
nominal_datatype dfexpr = Df_Name name     | Df_Sk_enc dfexpr dfexpr
nominal_datatype mess = M_Name name        | M_Sk_enc name mess
nominal_datatype Proc = Pnil                         | in_pref dfexpr ≪name≫Proc
                      | out_pref dfexpr dfexpr Proc  | par Proc Proc
                      | res ≪name≫Proc               | bang Proc
                      | boolean_guard guard Proc     | letp expr ≪name≫Proc
```

Sk_enc and *Sk_dec* represent $\mathsf{E}_\eta(\zeta)$ and $\mathsf{D}_\eta(\zeta)$ respectively. By declaring those datatypes as nominal datatype, the nominal package generates a powerful induction rule, where bound names occurring in the inductive cases will be automatically chosen to be different from any name (or variable) already used in a proof. Thus, naming clashing are avoided automatically. The Nominal package derives several proofs so that Isabelle's type system can in most circumstances automatically infer when a type is a permutation type. Nominal datatypes are always permutation types and their elements are finitely supported.

Substitution operators for the different datatypes are implemented as functions by using the recursion combinator that is automatically generated by the Nominal package for the datatypes terms defined above; this allows us to define recursively functions that respect α-equivalence classes.

```
subst_mess_Name :: mess ⇒ name ⇒ name ⇒ mess
 subst_mess_Name m n ≡ λa. (M_Name a)

subst_mess_enc :: mess ⇒ name ⇒ name ⇒ mess ⇒ mess ⇒ mess
 subst_mess_enc m n ≡ λt _ m2. (M_Sk_enc t m2)
```

[4] The problem is that datatypes used within nominal datatypes must satisfy several "equivariance properties". These properties are automatically proved for nominal datatypes, but are left to the user for normal datatypes.

```
 subst_mess :: mess ⇒ mess ⇒ name  ⇒ mess   (_[_ ~~ _])
  e[m ~~ n] ≡ (mess_rec (subst_mess_Name m n) (subst_mess_enc m n)) e
```

mess_rec denote the recursion combinator for *mess*. The notation $e[m \sim\sim n]$ is a syntactic sugar, and it can be read as e with m for n, and represent the substitution of all occurances of n of type *name* in the message e with m. Evaluation operators for expressions and boolean guards $[\![.]\!]$ are implemented as a nominal partial recursive function(**nominal-primrec**). The output of the function that evaluates expressions is of type *mess option*, i.e. if the evaluation is succesfull we expect a message otherwise an error. In Isabelle/HOL the type *t option* models the result of a computation that may terminate with an error (represented by *None*) or return the value v (represented by *Some* v).

```
consts
  eval_expr :: expr ⇒ mess option
nominal-primrec
   eval_expr (Name a) =  Some (M_Name a)
   eval_expr (Sk_enc e1 e2)= (case eval_expr(e2) of
                  None ⇒ None
                  | Some M ⇒ (case eval_expr(e1) of
                 None ⇒ None
                 | Some k ⇒ mess_case k (%l. Some(M_Sk_enc l M)) (%x y. None)))

eval_expr (Sk-dec e1 e2)= (case eval-expr(e2) of
                None ⇒ None
               |Some X ⇒ (mess_case X (%l. None)
                           (%k M. (case eval_expr(e1) of
                     None ⇒ None
                    |Some N ⇒ mess_case N (%k'.(if k=k' then (Some M) else None))
                                                (%x y. None)))))
```

eval_expr (Sk_enc e1 e2) asserts if the evaluation of $e2$ is *None* then the evaluation of *(Sk_enc e1 e2)* is *None*; otherwise if the evaluation is of type (*Some M*) we make a case distinction: if the evaluation of $e2$ is *None* the function return *None* otherwise if the evaluation is of type (*Some k*) it returns *M_Sk_enc l M* otherwise it returns *None*. *eval_expr (Sk_dec e1 e2)* follows the same kind of evaluation. In Isabelle "case" expressions are just sugared syntax for a special case combinator which is automatically defined whenever we define a datatype. For nominal datatypes, however, this is not yet supported, hence we define a case combinator (*mess_case*) for this purpose.

```
consts mess_case :: mess ⇒ (name ⇒ 'a) ⇒ (name ⇒ mess ⇒ 'a) ⇒ 'a
nominal-primrec
  mess_case (M_Name n) c1 c2 = c1 n
  mess_case (M_Sk_enc n m) c1 c2 = c2 n m
```

The commitment relation is defined by induction; as an example, we report the commitment of (GUARD) rule.

```
comm_guard: ⟦ eval_guard(g);  P − α  ⟼ P' ⟧ ⟹  (g γ P) − α  ⟼ P'
```

In $[\![\ldots]\!]$ we have both the precondition and the side condition of the rule. $P - \alpha \longmapsto P'$ stands for the commitment of P doing an action α and then behaves like P'. Guards are evaluated to a boolean value by *eval_guard*, which is the (encoding of) evaluation of guards defined in Section 2. *eval_guard* is defined by recursion on the syntax of guards, much like *eval_dfexpr*; notice that the guards have a binder. ($g\gamma P$ is the guarded process, denoted by ϕP in Section 2.)

4.2 Implementation of Hedged bisimulation

In Isabelle/Nominal, hedges are just sets of term pairs. The consistency of hedges (*Cons H*) is defined by a predicate formed by three clauses corresponding to Definition 1. The notions of analysis (*Analysis H*) and irreducibles (*Irreducibles H*) are implemented as inductive sets, via the *least fixed point*.

Let us focus now on hedged bisimilarity, which is defined by coinduction. We describe in detail the second condition (about input transitions):

consts *HedgedBisim* :: (*hedge* × *Proc* × *Proc*) *set*
coinductive *HedgedBisim*
intros
HedgedBisim_Def: $[\![$*H* ∈ *Finites*; (*Cons H*);
(clauses for τ, omitted)
∀ *P′ a b M N B*.(*P* − (*a M*) ⟼ *P′*)∧ (*H*⊢(*a*↔ *b*)) ∧ (*B*∈ *Finites*) ∧ (*B* ♯ (*H*,*P*,*Q*)) ∧ (((*H*∪(*ID B*)))⊢(*M*↔*N*)) ⟶ (∃ *Q′*.(*Q*,*b*,*N*,*Q′*)∈ *commIn* ∧ (((*H*∪(*ID B*)),*P′*,*Q′*) ∈ *HedgedBisim*));
(symmetric clause for input, omitted)
∀ *P′* (*c*::*name list*) *a b M*.(*c* ♯ (*H*,*P*)) ⟶(*P* − *cobarb*(*a*) ⟼ (*concAgent* (*ConcChain c M P′*))) ∧ ((*H*)⊢((*M_Name a*)↔(*M_Name b*))) ⟶ (∃ *Q′* (*d*::*name list*) *N*.(*d* ♯ (*H*,*Q*)) ∧ ((*Q*,*b*,(*ConcChain d N Q′*))∈ *commOut*) ∧ (((*Irreducibles*(*H* ∪{(*M*,*N*)})),*P′*, *Q′*) ∈ *HedgedBisim*));
(symmetric clause for output, omitted) $]\!]$ ⟹ (*H*,*P*,*Q*) ∈ *HedgedBisim*

(H ∈ Finites) and *(Cons H)* require the hedge H to be a consistent finite set of message pairs. $(Q, b, N, Q') \in commIn$, where b is of type *name* and N is of type *mess*, stands for the commitment relation of the process Q and abstraction Q' under the input $b\ N$, possibly preceded by τ transitions. Notice that the freshness of the names in B is easily ensured by the hypothesis $(B\#(H, P, Q))$.

The clauses for the output transitions are represented similarly; here again, we use the freshness predicate from Nominal package, to encode the freshness of locally scoped (i.e., newly created) names c, d.

4.3 Example: "Perfect encryption"

In order to explain how the implementation of spi calculus presented above can be used, we give an example proof by proving a simple bisimilarity, that is the *simple encryption* property taken from [1]. We want to prove that for all M, M', the processes

$$(\nu k)\bar{c}\langle \mathsf{E}_k(M)\rangle \quad \text{and} \quad (\nu k)\bar{c}\langle \mathsf{E}_k(M')\rangle \tag{1}$$

are hedged bisimilar. This means that there is no way to distinguish the cleartext message if the encryption key is kept secret.

In the proof given in [1], a bisimulation S is defined, such that

$$(\{c, n\}, \{\}) \vdash \ (\nu\ k)\overline{c}\langle \mathsf{E}_k(M)\rangle\ S\ (\nu\ k)\overline{c}\langle \mathsf{E}_k(M')\rangle$$

However, instead of explicitly define such S and prove that it is a bisimulation, we can take advantage of the coinductive support provided by the Isabelle/Nominal environment to directly prove that

$$(\{c, n\}, (\nu\ k)\overline{c}\langle \mathsf{E}_k(M)\rangle, (\nu\ k)\overline{c}\langle \mathsf{E}_k(M')\rangle) \in \mathit{HedgedBisim}$$

and the Isar proof sketch is the following:

```
lemma perfectEncyption:
 shows ({c, n},(⟨ν(a)⟩( c<(a∫ M∫ )>.PNil)), (⟨ν(a)⟩(c<(a∫ N∫ )>.PNil))) ∈ Hedged-
Bisim   (is ?x ∈ _ )
proof −
  have ?x : {?x} by simp
  then show ?thesis
  proof coinduct
    case (HedgedBisim z)
      have ?HedgedBisim_Def  sorry
  then show ?case by blast
  qed
qed
```

Let us analyze how the proof proceeds. Basically, the proof is done by using forward reasoning, through the *coinduct* proof method; then the proof splits into several sub-cases, before working towards one of the disjuncts. We also need to fill in a sensible starting point $?x : \{?x\}$ and take special care of the types here. $?x : \{?x\}$ is solved by *simp* (an Isabelle method that solves the goal using simplification rules); *?thesis* stands for the current goal to be proved. At this point the proofs is done by case analysis. Applying the rule *HedgedBisim_Def*, we are left with eight subgoals (the 6 clauses of coinductive definition, plus finiteness and consistency of hedge). Each case is then proved on its own (quite tediously); in the proof sketch above, this part is replaced by the command **sorry** (which proves anything but it is very convenient for top down proof development; this command can be replaced by the actual proofs later on). In particular, the proof of these cases exploits the features provided by the Nominal package for handling freshness of bound names. We are able to finish the coinduction step, working from the case assumptions to the conclusion *?case*, via the *blast* method (a classical reasoner which tries to solve automatically the current goal).

5 Conclusions and future work

In this paper we have presented a formalization of the spi calculus, a calculus of cryptographic processes, in the Isabelle/HOL proof assistant using the new

Nominal package. The proof environment so obtained allows for formal proofs which closely correspond to the (traditional) manual proofs, and in some sense are even simpler because we don't have to figure out and define explicitly bisimulations beforehand, thanks to the support provided by Isabelle to coinductive proofs. The Nominal package really played an important role to this end: it allows for a smooth handling of binding operators, thus reducing the overhead in encoding the system and conducting a formal proof.

However, although the Nominal package is already usable and fruitful, in our opinion some details need to be improved, in particular the support for recursively defined functions and case analysis over nominal datatypes. Moreover, at the moment is not possible to do *reflection*, i.e. implementing computations inside the logic rather than in the meta-language, and this due to the fact that the current version does not support co-generation.

Related work Theorem provers and proof assistants have been widely used to model process algebra and reason about correctness. Paulson's work [19] is arguably the first application of Isabelle to the verification of cryptographic protocols. Actually, the π-calculus is a paradigmatic example for proof environments and encoding techniques designed to handle binding operators. The closest development to ours is [4], where Bengtson and Parrow have formalised the π-calculus in Isabelle/HOL using the Nominal package, providing a library which allows users to carry proofs about π-calculus. Other approaches to binding management are de Bruijn indexes [12], and *(weak) higher order abstract syntax* [13]. De Bruijn indexes are quite cumbersome to use in interactive proofs, because names disappear completely. On the other hand, the (weak) HOAS approach needs to postulate some key properties (the so called *Theory of Contexts*) as axioms. Although proved to be consistent with (classical) higher order logic, the Theory of Contexts is inconsistent with the Axiom of Choice; therefore, its portability to the Isabelle environment is still under discussion.

Future work A first research stemming from the present work is to prove some general meta-theoretic results about spi-calculus, similarly to the work done by Briais in Coq [8]. Another interesting possibility is to implement special tactics to be used during the proof developments for proving decidable equivalences. These new commands can be written completely inside Isabelle; a possible way, following ProVerif approach, is to translate the protocol and the goal into classical logic, and then take advantage of the powerful support provided by Isabelle/HOL to classical reasoning. Alternatively (and more efficiently), the new commands can call auxiliary tools, external to Isabelle. To this end, the recent works about symbolic bisimulation of spi calculus [8] may be useful.

References

1. M. Abadi and A. D. Gordon. A bisimulation method for cryptographic protocols. *Nord. J. Comput.*, 5(4):267–, 1998.

2. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The Spi calculus. *Journal of Information and Computation*, 148(1):1–70, 1999.
3. A. Armando, D. A. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuéllar, P. H. Drielsma, P.-C. Héam, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron. The AVISPA tool for the automated validation of internet security protocols and applications. In *Proc. CAV*, LNCS 3576, pages 281–285. Springer, 2005.
4. J. Bengtson and J. Parrow. Formalising the π-calculus using nominal logic. In *Proc. FOSSACS*, LNCS 4423, pages 63–77, 2007.
5. B. Blanchet. From secrecy to authenticity in security protocols. In *Proc. SAS*, LNCS 2477, pages 342–359. Springer, 2002.
6. B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. In *Proc. 20th LICS*, pages 331–340. IEEE, 2005.
7. M. Boreale, R. D. Nicola, and R. Pugliese. Proof techniques for cryptographic processes. *SIAM J. Comput.*, 31(3):947–986, 2001.
8. J. Borgström, S. Briais, and U. Nestmann. Symbolic bisimulation in the spi calculus. In *Proc. CONCUR*, LNCS 3170, pages 161–176. Springer, 2004.
9. J. Borgström and U. Nestmann. On bisimulations for the spi calculus. *Mathematical Structures in Computer Science*, 15(3):487–552, 2005.
10. E. M. Clarke, S. Jha, and W. Marrero. Verifying security protocols with brutus. *ACM Trans. Softw. Eng. Methodol.*, 9(4):443–487, 2000.
11. M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax involving binders. In *Proc. 14th LICS*, pages 214–224, 1999. IEEE.
12. D. Hirschkoff. Bisimulation proofs for the π-calculus in the Calculus of Constructions. In *Proc. TPHOL'97*, LNCS 1275. Springer-Verlag, 1997.
13. F. Honsell, M. Miculan, and I. Scagnetto. π-calculus in (co)inductive type theory. *Theoretical Computer Science*, 253(2):239–285, 2001.
14. H. Hüttel. Deciding framed bisimilarity. In *Proceedings of Infinity'02*, volume 68 of *Electronic Notes in Theoretical Computer Science*, pages 1–18, 2003.
15. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Inform. and Comput.*, 100(1):1–77, 1992.
16. J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murϕ. In *IEEE Symposium on Security and Privacy*, pages 141–151. IEEE Computer Society, 1997.
17. K. S. Namjoshi. Certifying model checkers. In *Proc. CAV*, LNCS 2102, pages 2–13. Springer, 2001.
18. T. Nipkow and L. C. Paulson. Isabelle-91. In *Proc. of the 11th CADE*, LNCS 607, pages 673–676, 1992. Springer-Verlag.
19. L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1-2):85–128, 1998.
20. A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.
21. D. Sangiorgi, N. Kobayashi, and E. Sumii. Environmental bisimulations for higher-order languages. In *Proc. LICS*, pages 293–302. IEEE Computer Society, 2007.
22. C. Urban and C. Tasson. Nominal techniques in Isabelle/HOL. In *Proc. CADE*, LNCS 3632, pages 38–53. Springer, 2005.
23. M. Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In *Proc. TPHOLs*, LNCS 1690, pages 167–184. Springer, 1999.
24. S. Yu and Z. Luo. Implementing a model checker for LEGO. In *Proc. FME*, LNCS 1313, pages 442–458. Springer, 1997.