Implementing Secure Broadcast Ambients in Isabelle using Nominal Logic

Ayesha Yasmeen and Elsa L. Gunter

Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, USA
yasmeen@uiuc.edu,egunter@cs.uiuc.edu

Abstract. In this work we present our modeling of a calculus being developed by us namely Secure Broadcast Ambients, using the nominal package for Isabelle. Our calculus is an extension of the ambient calculus where the nature of communication is broadcast within domains modeled by ambients. We allow reconfigurable configurations of communication domains, access restrictions to domains and the capability of modeling cryptographic communication protocols in broadcast scenarios.

Key words: nominal logic, process calculi, ambient calculi, broadcast, bisimulations

1 Introduction

In a world of increasing dependence on electronic communications between reconfigurable and mobile devices, there is a clear need for accurate formal systems to model these devices and their communications to facilitate guaranteeing such properties as functionality, security and privacy, and data integrity. Ambients [6], including boxed ambients [4,3], are formalisms that have been developed to model such mobile devices and their communication. Ambients have an associated topology that confine their movement and their communication options. This topology has traditionally been restricted to tree structures, and communication and movement have been restricted to adjacent ambients. The tree structure implies that an ambient can only be "in" one other ambient at a given time. This poses problems for modeling aspects of networks, such as routers. A router is most naturally modeled as being "in" multiple domains at once. Similarly, a laptop with an ethernet connection, a bluetooth connection and a dialup-modem connection, can be thought of as being "in" three different domains at once. The restriction of the topology to tree structures prevents modeling these devices that way. In this work, we loosen this constraint to allow the topology to be that of a dynamically reconfigurable directed acyclic graph, thus allowing one ambient to be in more than one other at a given time, or possibly none at all.

In theoretical models of systems, and ambients in particular, communication is often modeled using point-to-point channels. Depending on the particular calculus, many processes may have access to a given channel, but each communication will have a unique recipient. Within such frameworks, modeling broadcast and multicast communications must be done using multiple unicasts. Alternatives to this have been devised using broadcast communications. These include broadcast communication limited to a specified domain. However, in frameworks with broadcast within a domain, the domains are relatively static with only at most code moving among them. Because the code alone is mobile, it carries no identity with it, which limits the ability to concisely and accurately model the organization of the domains, and model the restriction of access to the domains by other domains. In our work, we have broadcast communication within ambients. Messages announced to the ambient are heard by the ambient and all ambients directly within it. Ambients may restrict access to themselves, and hence to the privilege of the communication within them, based on the identity of potential entrants, without requiring that their names be hidden.

Having broadcast communication, our model has the potential to naturally model communication protocols in the presence of eavesdroppers. The standard way to establish private communication using a broadcast medium is to use encryption. To facilitate reasoning about such protocols, we have enriched our language with cryptographic primitives, much in the manner of the spi-calculus [1]. To further facilitate reasoning about such protocols, which often involve information shared among certain parties, we have inductively defined the `knowledge` of an ambient, in the style of [8], and give tests sufficient to prove that secrets held between interacting parties do not become a part of the `knowledge` of non-participating ambients. This notion of `knowledge` reflects the ability of an ambient to possibly synthesize new information from all information already obtained, in keeping with the Dolev-Yao model [5].

To see an example of the desirability of multiple communication domains, and the ability to span multiple domains, let us consider the scenario of a local area network comprising of a router connecting some home desktop computers and laptops to the outside world. This router is capable of directly communicating to the outside internet network and the home computers. Hence, virtually it is present in multiple communication domains simultaneously. If it is additionally a wireless router, then MAC filtering can be modeled by access restriction based on ambient identity. The ability to dynamically reconfigure the communication topology can be used to model the laptop entering the network, and later leaving it as it shuts off, or moves to another communication domain.

Continuing the example, let us consider the Ethernet local area network architecture in the home. Ethernet is built around a principle of localized broadcasting. Hence potentially, every computer in a subnet of the Ethernet can see the packets going to every other computer on that subnet. This situation has led to the advent of sniffer software, which can intercept all data on an Ethernet subnet. As a result, active attackers can use sniffing techniques to capture sensitive information and use it maliciously. As an example of how to use **Secure Broadcast Ambients**, we can model a scenario involving ethernet sniffer software running on a computer,*Sniffer*, in the Ethernet home local area network where someone wants to use another computer, *Laptop*, to log in to a website, giving their username and password. Unfortunately, if this information is not communicated in encrypted form, then *Sniffer* has every opportunity to capture it. Using the encryption primitives of **Secure Broadcast Ambients**, we may model protocols that allow the username and password to be communicated without being revealed to *Sniffer*.

We are developing a calculus **Secure Broadcast Ambients**, which allows broadcast communication inside regions. These regions are capable of restricting access to itself and their reconfigurable topology is allowed to form a directed acyclic graph. Mobile agents are capable of moving among the regions. The syntax of this calculus has multiple different sorts of variables, different sorts of binding constructs, many syntactic categories where several syntactic categories are mutually recursive to each other. All these features poses challenges to our goal of formalizing this calculus's syntax and semantics in a logic like HOL. We used the nominal package of Isabelle [11] to aid us in achieving our goal. In this work we describe our experience in modeling **Secure Broadcast Ambients** using the nominal package of Isabelle. We intend to provide more guideline in addition to those provided in [11]. We hope that our experience will help other researchers working on modeling calculi which presents challenges similar to ours. We present the syntax of **Secure Broadcast Ambients** in Section 2. We then describe our implementation in Section 3. We present the formal semantics of **Secure Broadcast Ambients** in a condensed form while describing our implementation in that section. Finally we conclude in Section 4.

2 Syntax of Secure Broadcast Ambients

In order to define the syntax of **Secure Broadcast Ambients** we use the following categories of identifiers: ambient names: $n, m \in \mathsf{Amb}$, capability variables: $i \in \mathsf{CapVar}$, message variable: $x \in \mathsf{MessVar}$, key variables: $k \in Keys$. The syntax of **Secure Broadcast Ambients** is presented in Table 1. *Messages*, *Processes* and *Systems,* are the main syntactic categories. Messages include *message identifiers*, ambient names, capabilities, key variables and data. We also allow encrypted and compound messages. Capabilities, ranged over by C, can be either the capabilities for entering and exiting an ambient, capability variables or a "path", which is a sequence of capabilities describing a mobility path. An ambient can indicate the intention to move into another ambient by $\mathsf{in}\ m$. However, this movement capability can only be successful if a corresponding co-capability is there to permit this move. The corresponding co-capability can be either $\overline{\mathsf{in}\ m}$ allowing specifically m to enter, or $\overline{\mathsf{in}\ \_}$ indicating permission for any ambient. The only further restriction placed on entrance is that an ambient is not allowed to enter a descendant of itself. This interpretation of ambient movement leads to a directed acyclic graph structure for the hierarchy of ambients.

Ambient List:	
$\mathcal{L}$ ::= empty	nil process
$\mid$ m ; $\mathcal{L}$	composition
Capabilities:	
C ::= $\mathsf{in}\, m$	enter
$\mid$ $\mathsf{out}\, m$	exit
$\mid$ $C; C'$	path
$\mid$ i	capability var
Ambient Pattern:	
μ ::= $\_$	any ambient
$\mid$ m	ambient name
Format:	
F ::= $m : P$	ambient name
$\mid$ $i : P$	capability var
$\mid$ $0 : P \ ; \ \mathsf{suc}(x) : Q$	natural numbers
$\mid$ $\{x\}_k : P$	decryption
$\mid$ $(x, y) : P$	pairs
Actions:	
π ::= C	capability
$\mid$ $\overline{\mathsf{in}\, \mu}$	allow enter
$\mid$ $(x)^m$	input
$\mid$ νk	new key
$\mid$ $\langle M \rangle^m$	output
$\mid$ $\mathsf{create\_amb}(m, P)$	ambient creation

Messages:	
M, N ::= x	message ident
$\mid$ m	ambient name
$\mid$ C	capability
$\mid$ 0	natural number
$\mid$ $\mathsf{suc}(M)$	natural number
$\mid$ (M , N)	pairing
$\mid$ k	key
$\mid$ $\{M\}_k$	encrypted message
Processes:	
P, Q ::= nil	nil process
$\mid$ $P \mid Q$	composition
$\mid$ $!P$	replication
$\mid$ $\pi.P$	prefixing
$\mid$ $\mathsf{cond}\ M\ \mathsf{is}\ N\ \mathsf{in}\ P$	data comparison
$\mid$ $\mathsf{case}\ M\ \mathsf{of}\ F$	case analysis
Systems:	
$\mathcal{S}$ ::= $\mathsf{nilsystem}$	empty system
$\mid$ $m[P]$	ambient
$\mid$ $(x)^m(\mathcal{S})$	broadcast receive
$\mid$ $\nu k.(\mathcal{S})$	key restrict
$\mid$ $\nu m :: \mathcal{L}.(\mathcal{S})$	ambient restrict
$\mid$ $\mathcal{S}_1 \parallel \mathcal{S}_2$	parallel

Table 1. Syntax of Broadcast Ambients

In **Secure Broadcast Ambients**, an ambient can be in multiple ambients at the same time. An ambient may even fail to be in any ambient, for example, a laptop that has been turned off. An ambient n exits from the ambient m by the $\mathsf{out}\, m$ action without requiring any permission from any other ambient, and without effecting the relationship of n to any other ambient.

Another important aspect of **Secure Broadcast Ambients** is that we have removed the channels used for inter-ambient communication. In our calculus the name of a parent ambient acts as the broadcast channel for both itself and its children. This way, any ambient can listen to any conversation that is going on between any of its parents and their children. Henceforth channels are synonymous with ambients.

A process can be an empty process, nil. It can be a parallel composition of two processes. A process can be replicated. A process can be prefixed with some action. The actions can be to move into an ambient or to exit from an ambient, to allow entrance, to send or receive a message or to restrict keys. A process can create an ambient and at the same time define the process inside that ambient. A process can also perform matching, or case analysis on a message much in the manner of [1]. The case analysis patterns are given by the formats. The most interesting pattern is that of decryption. In this work we only consider symmetric encryption and so the decryption key is the same as the encryption key for every encrypted message. A system can be an empty system, an ambient, a system waiting to receive a message, or multiple systems in parallel. Systems can create a new key or a new ambient with a given parent list.

Ambient and key restriction, message input, ambient creation and case analysis are the binding constructs. In light of these binding constructs the free variables of messages, processes and systems are defined in the usual way.

We now show the encoding of the home router using our syntax.

$$HomeRouter[!((x)^{HomeLan}.\langle x\rangle^{ISP}.\mathsf{nil}) \mid !((x)^{ISP}.\langle x\rangle^{HomeLan}.\mathsf{nil})]$$

Its job is to capture all outgoing packets in the home network and forward them to the ISP and vice versa. Now, each computer will have to determine which of the messages (representing TCP/IP packets) arriving at *HomeLan* are meant for them.

3 Implementation in Isabelle

Our goal is to encode **Secure Broadcast Ambients** in Isabelle and to use this implementation later on to model and reason about cryptographic communication protocols in scenarios where the method of communication is broadcast. We first analyzed the calculus to determine what the most important aspects of this calculus are that should be considered before modeling in Isabelle. Examining the calculus one can observe that there are four different types of variables: message variables, capability variables, ambient names and keys. Moreover we have ambient name restrictions and key restrictions. We also have bound occurrences of all different types of variables. Also the processes are mutually recursive with some other categories. Hence we need a mechanism which will allow us to reason about α-equivalence classes of the main syntactic categories like processes and ambient systems. We decided to use the nominal package for handling α-equivalence classes of terms.

3.1 The first step: Atom declarations

As per the norm of using nominal package we first define the different types of data with corresponding binders that can be bound in our datatypes. They are called atoms because their internal structure is immaterial compared to their being distinguishable. From the syntax of **Secure Broadcast Ambients** we have observed earlier that it uses four different types of such bound entities: message and capability variables, ambients and keys. We use the atom `var` to denote a message variable, `amb` to denote ambients, `cvar` to denote capability variables and `key` to denote encryption key variables.

```
atom_decl var cvar amb key
```

We next show how we encoded the syntax of the calculus using nominal package.

3.2 Capabilities and Messages

We do not have any binding in the definition of messages or capabilities. Hence their declaration is quite simple. We first define the capabilities as follows:

```
nominal_datatype capability = IN amb
                            | OUT amb
                            | capas capability capability
                            | CapaVar cvar
```

Then we define the messages.

```
nominal_datatype ambmsg = Var var
                          | AmbM amb
                          | Key key
                          | Capable capability
                         ...
                          | TPair ambmsg ambmsg
                          | Enc ambmsg key
```

3.3 Processes and Systems

Processes make up the first syntactic category that has bindings. In order to associate the bindings in the actions with the corresponding bound variables in the processes we found it necessary to inline the actions in the processes. There are several noteworthy aspects of the processes. Processes have message variable bindings in their message receiving action. Processes have key restrictions and ambient restrictions. Processes have ambient binding for the new ambient creating construct. The ambient binding case is even a little bit more involved in the sense that the newly created ambient's name has to be bound both in the process that creates it and also in the code that gets put inside it. Hence it needs to be bound in two different elements at the same time. However, as nominal datatypes do not allow a variable to be bound to a nested datatype as described in [11], we could not do that in a straight forward manner. We had to create a datatype for a pair of processes which is mutually recursive with processes. A newly created ambient's name is now restricted to a process-pair datatype element. We made the case analyzing formats a separate datatype which is mutually recursive with the processes. The processes are defined in Isabelle as follows:

```
nominal_datatype Proc = Pnil
                      | par "Proc" "Proc"
                      | bang "Proc"
                      | capa "capability" "Proc"
                      | entrycocapable "muamb" "Proc"
                      | recv "<<var>>Proc" "amb"
                      | send "ambmsg" "amb" "Proc"
                      | createamb "<<amb>> Proc_pair"
                      | cond "ambmsg" "ambmsg" "Proc"
                      | ambcase "ambmsg" "Format"
                      | keypres "<<key>> Proc"
    and Proc_pair     = Proc_pair "Proc" "Proc"
     and Format       = FA "<<amb>> Proc"
                      | FC "<<cvar>> Proc"
                      | FN "<<var>> Proc" "Proc"
                      | FMP "<<var>> <<var>> Proc"
                      | FK "key" "<<var>> Proc"
```

Finally the ambient systems were defined as follows:

```
nominal_datatype AmbSystem =
...
| WholeAmb amb Proc
| Listener amb "<<var>> AmbSystem"
| ambres "<<amb>> AmbSystem" amblist
| keyres "<<key>> AmbSystem"
```

3.4 Substitution Functions

After defining a datatype, nominal package automatically proves lots of necessary lemmas. For a datatype of name D the four most prominent ones are `D.perm, D.supp, D.fresh and D.inject` [11]. However, substitution of terms for atoms appearing in a new datatype is not automatically defined. Hence the step after declaring a new datatype is to define how each atom appearing in a datatype can be substituted. We defined substitution for the different types of atoms in the various datatypes that we have. Let us consider the substitution functions we defined for messages. All four types of atoms appear in messages. Hence we defined substitution for each of them. We first defined substituting a variable with a message inside a message.

```
consts subst_ambmsgvar :: "ambmsg ⇒ var ⇒ ambmsg ⇒ ambmsg"
("_ M[_::= _ ]" [100,100,100] 100)
```

We then defined substituting a capability variable with a capability in a message.

```
consts subst_ambmsgC :: "ambmsg ⇒ cvar ⇒ capability ⇒ ambmsg" ("_ MC[_::= _ ]" [100,100,100]
100)
```

We defined ambient and key atom substitutions similarly.

3.5 Proving lemmas for the user defined functions

- In the nominal logic framework, a very important concept is that of equivariance [9]. However, whereas the nominal package automatically proves equivariance lemmas for the datatypes it generates, the onus of proving the necessary lemmas for user defined functions lies on the user. After defining every new function we need to prove that the function is equivariant. As mentioned in [10] a function f is equivariant if for atom permutation π, we have that: $\pi \bullet (f x_1 \ldots x_n) = f(\pi \bullet x_1) \ldots (\pi \bullet x_n)$ where x_i are the inputs of the function f.
  As a result we next prove the equivariance lemmas for the variable substitution function. However one important aspect that should be remembered is that since we have four different types of atoms, we need to prove that variable substitution is equivariant for permutations of each type of atom. Hence we end up proving four equivariance theorems for each variable substitution function.
  As an example, for the function `_M[_::=_]`, which substitutes a variable with a message in a message, we have four equivariance lemmas looking like:
  `lemma subsvar_var_eqvt[eqvt, simp]:`
  ...
  `shows` $\pi \bullet (m\ M[x ::= t]) = ((\pi \bullet m)\ M[(\pi \bullet x) ::= (\pi \bullet t)])$
  where π is message variable, capability variable, ambient or key permutations respectively. Every such equivariance theorem should be given the equivariant attribute `eqvt`.
- We also defined support for the variable substitution function. Basically it says that the free message variables of a message m after substituting a variable x with a message t is a subset of the free variables of m and t sans x.
  `supp(m M[x::=t]) ⊆ (supp(m)-{x}) ∪ supp(t):: (var set)`
  Here we need to explicitly mention that the support is being defined for `var` atoms. Here support lemma is only needed for the type of atom that is being replaced in a datatype.
- We also proved freshness lemmas for the substitution functions.

In a nutshell the pattern that we followed were:

- for each new datatype do the following:
  - define substitution for every type of atom appearing in the datatype
  - define equivariance for every type of atom for each substitution function.
  - define support for the substitution function for the type of atom that is being replaced
  - define freshness for the substitution function

This procedure has to be followed for every datatype and for every atom if the datatype is to be used later in some other function. For example while defining substitution in processes we needed to reason about substitution in capabilities, messages and other nominal datatypes appearing in the processes. Hence substitution for these datatypes have to have been properly defined and their equivariance should already have been proved.

3.6 Substitution in Processes

Substitution of terms for atoms in processes were challenging for many reasons. A process has all different types of atoms appearing in it and also it is mutually recursive with two other nominal datatypes. It also contains various bindings. Hence defining substitution and proving the required lemmas about them have a slightly different flavor.

Message variable substitution in processes is defined as follows:

```
consts subst_proc :: "Proc ⇒ var ⇒ ambmsg ⇒ Proc" ("_ P[_::= _ ]" [100,100,100] 100)
consts subst_form :: "Format ⇒ var ⇒ ambmsg ⇒ Format"
("_ F[_::= _ ]" [100,100,100] 100)
consts subst_Proc_pair :: "Proc_pair ⇒ var ⇒ ambmsg ⇒ Proc_pair"
("_ PP[_::= _ ]" [100,100,100] 100)
   nominal_primrec
(* "_ P[_::= _ ]" is defined here *)
 "Pnil P[ x ::= t ] = Pnil"
...
" x1 # (m, x, t)⟹ (recv x1 P m) P[ x ::= t ] = (recv x1
(P P[ x ::= t]) m)"
" (send m1 m2 P) P[ x ::= t ] = (send (m1 M[ x ::= t ]) m2
(P P[ x ::= t ]))"
"m1 # (x, t) ⟹ (createamb m1 PP ) P[ x ::= t ] = (createamb m1
(PP PP[ x ::= t ]) )"
...
" (ambcase m f1) P[ x ::= t ] = (ambcase (m M[ x ::= t ])
(f1 F[ x ::= t ]))"
"[| k # (x,t)|]⟹ ( keypres k P) P[x::=t] = (keypres k
(P P[x::=t]))"

and (* "_ PP[_::= _ ]" is defined here:*)
"(Proc_pair P1 P2) PP[x ::=t] = (Proc_pair (P1 P[x::= t] )
(P2 P[x::= t] ))"

and (* "_ F[_::= _ ]" is defined here *)
"m # (x, t)⟹ subst_form (FA m P) x t = (FA m (P P[ x ::= t ]))"
"c # (x ,t)⟹ (FC c P1) F[x ::= t] = (FC c (P1 P[ x ::= t ]))"
"x1 # (P1, x, t)⟹ (FN x1 P2 P1) F[x ::= t] =
(FN x1 (P2 P[ x ::= t ]) (P1 P[ x ::= t ]))"
"[|x1 # (x2, x, t); (x2 # (x1,x,t)|]⟹
(FMP x1 x2 P) F[x::= t]= (FMP x1 x2 (P P[ x ::= t ]))"
" x1 # (k, x, t) ⟹(FK k x1 P) F[x::= t] =
( FK k x1 (P P[x::=t]))"
```

The reason for displaying such a big chunk of code was to illustrate various points.

- Defining substitution for a mutually recursive nominal datatype requires substitution functions for all the mutually recursive elements to be defined together. The reason being that the recursion combinator expects an equation for every term constructor.
- Special care needs to be taken for the constructs with bound atoms. The binders must be fresh in the variable that is being substituted, the term with which it is being substituted and any other bound atoms in that construct. If the freshness constraints are not provided properly then definition of the substitution function will result in subgoals which are either false or vacuous or hard to prove. For example, the rule for substitution in the new ambient creation construct is: `"m1 # (x, t) ⟹ (createamb m1 PP ) P[ x ::= t ] = ..."`. Here the bound ambient atom `m1` has to be fresh for both `x` and `t`. Also if there are multiple bound atoms in a term they have to be distinct from each other. For example in the rule `"[|x1 # (x2, x, t); (x2 # (x1,x,t)|]⟹ (FMP x1 x2 P) F[x::= t]= (FMP x1 x2 (P P[ x ::= t ]))"` where we match up a pair of messages, both the atoms `x1` and `x2` are bound in `P`. Hence the condition for `x1` is `x1 # (x2, x, t)` and similarly for `x2`.
- Nominal package needs a lot of properties of each recursive function operating on nominal datatypes to be proved before it can be used. The properties mainly deal with the finiteness of the support of

the function and its parameters and freshness constraints. However, the list of subgoals thrown at the user can be non-trivial. For functions like variable substitution in messages we find that the subgoals are trivial. They merely ask us to prove `True`. Repeated application of the rule `TrueI` takes care of all of them. It would have been simpler if this was done automatically by the nominal package. The substitution for messages is easier as it does not have binders or mutual recursion. But the substitution for processes is not easy and the nominal package gives us a total of 255 subgoals. We describe the tactics used by us to dispose of them next.

– A lot of the subgoals returned by the substitution in processes can be gotten rid of by simplifying with equivariance lemmas and other lemmas like one type of atom being fresh for other types of atoms and so on. Then the the subgoals which needed to prove some finiteness of supports were taken care of by the tactic `finite_guess`. Then the rest of the goals were about proving freshness conditions which were easily taken care of by the tactic `fresh_guess`. However we must assert that for these tactics to succeed properly, you have to have proven equivariance, support lemmas and freshness lemmas for each and every function used in the definition of the substitution function. For example, consider substituting in the case of sending a message over an ambient given by
`"(send m1 m2 P) P[ x ::= t ] = (send (m1 M[ x ::= t ]) m2 (P P[ x ::= t ]))"`.
Here we substitute the variable `x` with the message `t` inside the message `m1` that is being sent over the ambient `m2`. Hence we use the appropriate substitution function `_ M[_ ::=_ ]`. If all the necessary lemmas for that function are not proved earlier then the substitution for processes gives us subgoals that can be harder to prove.
– After defining a function we need to prove equivariance, support and freshness results for that function. For the functions dealing with simpler nominal datatypes like messages which do not have any bound variables, defining these lemmas are very easy. However, for processes it was a little bit more complex. For example, for equivariance of processes we need to define what it means for all of the mutually recursive elements together in the following format:
`"π• (P P[ x ::= t]) = (π• P) P[ (π• x) ::= (π• t)]"`
`and "π• (pp PP[ x ::= t]) = (π• pp) PP[ (π• x) ::= (π• t)]"`
`and "π• (f F[ x ::= t]) = (π• f) F[ (π• x) ::= (π• t)]"`
We have four such lemmas where π is `var prm`, `cvar prm`, `amb prm` and `key prm` respectively.
An example freshness lemma is given here:
`"[| x # P |] ⟹ (P P[ x ::= t ] = P)" and`
`"[| x # PP |] ⟹ (PP PP[ x ::= t ] = PP)" and`
`"[| x # F |] ⟹ (F F[ x ::= t ] = F)"`
where `x` is of type `var`.
– While proving the equivariance, support and freshness results for functions substituting atoms in processes, we needed to induct on the processes. We needed to use an inductive hypothesis that will induct on all the related mutually recursive datatypes together. In our case, we have `Proc`, `Proc_pair` and `Format` which are mutually recursive. The inductive hypothesis was named:
`Proc_Proc_pair_Format.inducts`.
Notice that you have to use `inducts` not `induct`. Hence the rule is that for a mutually recursive datatype P where $P_1, P_2, \ldots, P_n$ are mutually recursive, the inductive hypothesis looks like: $P_1\_P_2\_\ldots P_n$.inducts

We had to follow the above mentioned tasks for all different types of atoms appearing in a datatype. As a result, we ended up defining four different substitution functions for the processes. They were very similar in appearance. However, we always had to take care so that required freshness constraints are properly provided, so that there is no inadvertent free variable capture and that we do not attempt to substitute a bound variable.

3.7 Formal Semantics

After we have defined our datatypes with binders where necessary, defined substitution functions for all of them for all atoms as necessary and proven necessary lemmas for these functions, we find ourselves with the appropriate background for using the machinery to encode the formal semantics of the calculus.

(StrRepPar) $!P \equiv P \mid !P$	(StrPathPref) $(C_1;C_2).P \equiv C_1.C_2.P$
(StrNilProc) $m[\mathsf{nil}] \equiv \mathsf{nilsystem}$	(StrSysPar) $m[P_1 \mid P_2] \equiv m[P_1] \parallel m[P_2]$
(StrCond)	cond M is M in $P \equiv P$
(StrAmbMsg)	case m of $n\ : P \equiv P[m/n]$
...	
(StrProcSys)	$P \equiv Q \Rightarrow m[P] \equiv m[Q]$
(StrMakeAmb)	$n[\mathsf{create\_amb}(m,P).Q] \equiv \nu m :: \mathsf{empty}.$ $(n[Q] \parallel m[P])$, if $n \neq m$
(StrBrdcstListen)	$\nu m :: \mathcal{L}.\nu\overline{p} :: \overline{\mathcal{L}'}.m[(x)^n.P] \equiv \nu m :: \mathcal{L}.\nu\overline{p} :: \overline{\mathcal{L}'}.$ $((x)^n(m[P]))$, if $m = n$ or $m \notin \overline{p}$ and $n \in \mathcal{L}$
(StrResNil) $\nu u.\mathsf{nilsystem} \equiv \mathsf{nilsystem}$	(StrPKey) $\nu k_1.\nu k_2.P \equiv \nu k_2.\nu k_1.P$
...	
(StrNeutral)	$(x)^n.(S_1 \parallel S_2) \equiv S_1 \parallel ((x)^n.S_2)$ if $x \notin fv(S_1)$
(StrNoListen)	$(x)^n(\mathsf{nilsystem}) \equiv \mathsf{nilsystem}$
(StrCombListen)	$(x)^n.S_1 \parallel (y)^n.S_2 \equiv (z)^n.(S_1[z/x] \parallel S_2[z/y])$, z fresh in S_1, S_2

Table 2. Structural Equivalence

Structural Equivalence: Structural equivalence is defined for each of processes and systems. It is the smallest congruence containing the rules in Table 2, closed under alpha equivalence (where ν, message receipt, ambient creation, and case analysis are the binding constructs), and the associativity and commutativity of parallel composition of each of processes and systems, with nil and nilsystem as the respective identities. In the table we use u to denote $m :: \mathcal{L}$ or k.

The last three rules in Table 2 are the rules that enable broadcast communication. The rule (StrBrdcstListen) allows an ambient to lift the receive action on a particular broadcast channel of a process within it up to the level of the ambient system. The rule (StrCombListen) can then be used to combine the listening systems. It is the principal rule that is used to model broadcast systems. This rule combines multiple ambients listening on the same channel so that later on only one reaction is needed to send a message simultaneously to all the ambients listening on this ambient. In addition to being able to send a message to multiple parties simultaneously, in a broadcast scenario, the broadcaster can send out a message even if no one is listening on the broadcast channel being used. The rule (StrNoListen) enables us to model this scenario. Structural equivalence of two ambient systems are provided at the system level and the process level. First we encode the structural equivalence relation for processes. It is defined as follows:

```
inductive StructEquivProc :: "Proc * Proc ⇒ bool" where
nilproc[intro!]:"StructEquivProc((par Pnil P), P)"|
...
ambcompcase[intro!]: "m2 # m1 ⟹ StructEquivProc (
(ambcase (AmbM m1) (FA m2 P)), (P PA[ m2 ::= m1 ]))" |
...
```

The inductive definition also has some rules similar to substitution function definition. For the cases where binders are handled, appropriate freshness constraints have to be inserted [11]. Inductively defined relations also need to be equivariant. Nominal package provides an easier way of determining whether an inductively defined relation is equivariant or not. They provide a simple tactic for performing this task. The name of the tactic is `"equivariance"`. In order to prove that a relation R is equivariant, after inductively defining a relation we simply need to type `equivariance R`. In our case, the command we used was `equivariance StructEquivProc`.

Similar to proving equivariance of functions, we need to have that all other user defined function used in the inductive definition be equivariant themselves. For example in the `ambcompcase` case above, if the equivariance proving lemma for the ambient substitution function `_PA[_::=_]` is removed, then the nominal package will give an error message saying that it failed to prove that

...

Exit:
$$\frac{m \in \mathcal{L}_n}{\varDelta \mathbin{\#} n :: \mathcal{L}_n \mathbin{\#} \varPi \triangleright n[\mathsf{out}\, m\,.P] \longrightarrow \varDelta \mathbin{\#} n :: \mathcal{L}_n \setminus \{m\} \mathbin{\#} \varPi \triangleright n[P]}$$

FormationEquiv:
$$\frac{\varDelta_1 \triangleright S_1 \equiv \varDelta_1' \triangleright S_1' \quad \varDelta_1' \triangleright S_1' \rightarrow \varDelta_2' \triangleright S_2' \quad \varDelta_2' \triangleright S_2' \equiv \varDelta_2 \triangleright S_2}{\varDelta_1 \triangleright S_1 \rightarrow \varDelta_2 \triangleright S_2}$$

...

KeyRestrict:
$$\frac{\varDelta \triangleright S_1 \rightarrow \varDelta' \triangleright S_2,\ k \notin \mathrm{fv}(S_1), k \notin \mathrm{fv}(S_2)}{\varDelta \triangleright \nu k.S_1 \rightarrow \varDelta' \triangleright \nu k.S_2}$$

AmbRestrict:
$$\frac{\varDelta \mathbin{\#} n :: \mathcal{L} \triangleright S_1 \rightarrow \varDelta' \mathbin{\#} n :: \mathcal{L}' \triangleright S_2}{\varDelta \triangleright \nu n :: \mathcal{L}.S_1 \rightarrow \varDelta' \triangleright \nu n :: \mathcal{L}'.S_2}$$

Table 3. Unlabeled Transition System

`StructEquivProc` is equivariant. We encode structural equivalence of systems using the relation `StructEquivSys`(omitted here).

Unlabeled Transition System: We impose topological structure on the ambients using “configurations”. The configurations keep track of the topological layout of the ambients in a system. Roughly, for every ambient in a system, it lists the ambients it is in. A configuration is a list of pairs where each pair has the name of an ambient and the list of ambients it is in. The pair of a system, S and its topological structure that is its configuration, $\varDelta$, is called a formation denoted by $\varDelta \triangleright S$ and implemented as “ConfigSys” in Isabelle.
```
datatype Config = EmptyConfig
| Conf amb amblist Config
datatype ConfigSys = CS Config AmbSystem
```
We provide part of the unlabeled transition system in Table 3. We encoded the unlabeled transition system as follows:
```
inductive UTS :: "ConfigSys * ConfigSys ⇒ bool" where
...
xsn_exit[intro!]:"(InL (findlist C n) m)⟹UTS((CS C (WholeAmb n
(capa (OUT m) P))),(CS (delfromconf C n m ) (WholeAmb n P)))"|
xsn_formequiv[intro!]:"[| UTS((CS C1 S1),(CS C1a S1a)) ;
UTS ((CS C1a S1a),(CS C2a S2a)) ; StructEquivCS((CS C2a S2a),(CS C2 S2))|]⟹UTS((CS C1 S1),(CS
C2 S2))"|
...
```
In this definition the relation `StructEquivCS` stands for the relation defining structural equivalence for formations. We omit the definition of structural equivalence for formations in this work. The rule `xsn_exit` stands for the Exit transition rule in Table 3. If an ambient `n` wants to exit another ambient `m`, then we first check that in the configuration `C` (or $\varDelta$ in the Exit rule) asserts that the ambient `n` is actually inside ambient `m`. After ambient `n` exits ambient `m`, the configuration `C` is updated by deleting ambient `m` from ambient `n`'s parent list.

Observational Equivalences Observational(barbed) equivalences focus only on the observable actions and do not consider the messages being exchanged in the transitions. We donote a formation $\varDelta \triangleright S$ exhibiting a barb ξ by $\varDelta \triangleright S \downarrow \xi$. We present some of the barbs in Table 4. The barbs exhibited by a formation is defined inductively in Isabelle as follows:
```
inductive showsbarb :: "ConfigSys * barb ⇒ bool" where
barbsend[intro!]:"(InL (ALC n (findlist c n)) m)⟹showsbarb ((CS c (WholeAmb n (send msg1
```

Barb Send: $$\frac{m \in (\{n\} \cup \Delta(n))}{\Delta \triangleright n[\langle M\rangle^{m}.P] \downarrow \mathsf{send}\ m}$$

Barb In: $$\frac{m \in \mathrm{dom}(\Delta) \wedge m \notin \mathcal{L}_n}{\Delta \# n :: \mathcal{L}_n \# \Pi \triangleright n[\mathsf{in}\ m\,.P] \downarrow \mathsf{in}(n,m)}$$

...

NewKeyBarb: $$\frac{\Delta \triangleright S \downarrow \xi}{\Delta \triangleright \nu k.S \downarrow \xi}$$

FormationStructBarb: $$\frac{\Delta_1 \triangleright S_1 \equiv \Delta_2 \triangleright S_2 \quad \Delta_2 \triangleright S_2 \downarrow \xi}{\Delta_1 \triangleright S_1 \downarrow \xi}$$

SysParBarb: $$\frac{\Delta \triangleright S_1 \downarrow \xi}{\Delta \triangleright (S_1 \| S_2) \downarrow \xi}$$

Table 4. Barbs

```
m P ))),(Send m))"|
barbin[intro!]:" (InL (findlist c n) m) & (InConfig (findpreconfig
c m EmptyConfig) m )⟹showsbarb ((CS c (WholeAmb n (capa (IN m) P
))),(INB n m))" |
...|
barbformstruct[intro!]:"StructEquivCS(C1,C2) & showsbarb(C2,b)⟹
showsbarb(C1,b)"
```

Barbed equivalence is defined next. Roughly speaking two formations are barbed bisimilar if one can exhibit the barbs the other one can exhibit and if one can take a transition step, the other one can mimic it and after that transition they both result in systems which are also bisilimar.

```
consts BSConfig :: " (ConfigSys ⇒ ConfigSys ⇒ bool)⇒bool"
defs
BSConfig_def: "BSConfig R ==
(∀ C1 C2 S1 S2. (R (CS C1 S1)(CS C2 S2)) ⟶
( ∀ b. showsbarb ((CS C1 S1), b) ⟶
showsbarb((CS C2 S2),b) ) & (∀ C1a S1a. UTS ((CS C1 S1),(CS C1a S1a)) ⟶ ( ∃ C2a S2a. UTS
((CS C2 S2),(CS C2a S2a)) & (R (CS C1a S1a) (CS C2a S2a)) ) ) ) "
```

Two ambient systems are barbed congruent if they are barbed equivalent in the presence of all possible contexts executing in parallel with them where contexts are represented by arbitrary systems.

```
constdefs BC :: "AmbSystem ⇒ AmbSystem ⇒ bool"
"BC S1 S2 == (∀ (t :: AmbSystem) . BE (Sysp S1 t) (Sysp S2 t) )"
```

In order to determine what secrets a spy has gleaned in a system, we want to reason about the knowledge of an ambient in an ambient system. The knowledge of an ambient in a system is a set of messages. It is calculated by syntactically traversing a system and determining what information is available to a particular ambient in that system. Knowledge may be finite or it may be infinite. While laying out the foundation for providing the security lemma, we realized that there are some lemmas about freshness in sets that are missing in the nominal package. Generally speaking, if we have multiple atoms say $a, b, c, \ldots$, then it should be the case that `a# (X:: b set)`, `a# (X:: c set)` and so on. However, there is no such generic lemma in the nominal package which we felt were due to the reason that this requires reasoning about a set which may well be infinite, and in general providing the machinery automatically for functions and infinite sets are not easy to perform automatically. Still we feel that lemmas like these should be included in the nominal package. Knowledge of an ambient `m` in a given ambient system is defined (in part) as follows:

```
function knowledge :: "AmbSystem ⇒ amb ⇒ (ambmsg set) ⇒
(ambmsg set)" where
"knowledge NilSystem m I = {} "
| "knowledge (WholeAmb n P) m I = (if (m=n) then ((procinfo P I)∪
(AmbM m-I)) else )"
...
```

Here `procinfo` is a function which calculates the information contained in a process. We are now working on proving a secrecy theorem which tries to figure out under which conditions elements from a set of secrets will not end up in the knowledge of some spy ambients. We have defined the function that calculates knowledge of ambients in a system.

4 Conclusion and Related Work

We modeled a calculus namely **Secure Broadcast Ambients** which are capable of modeling cryptographic communication protocols of mobile agents where the nature of communication is broadcast within a domain. Our calculus allows a directed acyclic graph topology of the location of our agents, hence our topology is more flexible than the usual tree structured one. Our domains or systems are also capable of restricting access to themselves. Our focus was on delineating the steps we had to take and hurdles that we had to overcome to implement the calculus using nominal package. We encoded structural equivalence, unlabeled transition system, behavioral equivalence and congruence. We introduce the idea of knowledge of an ambient so that we can reason about how much information untrusted ambients have acquired in a given ambient system. Our future goal is to prove a secrecy theorem which will roughly give the conditions under which processes that do not reveal secrets to untrusted agents can be indistinguishable. Many work has been done on implementing various calculi in Isabelle using the nominal package. Bengston *et al.* has formalized the π-calulus using the nominal package [2]. Kahsai *et al.* has formalized the spi-calculus using the nominal datatype [7]. The examples directory of the nominal package presents various theories. Theories like `Class.thy` which implements term calculus was very beneficial to us as it deals with multiple atoms. We were also helped a lot by the nominal mailing list regarding issues of mutually recursive nominal datatypes.

References

1. Martín Abadi and Andrew D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. *Inf. Comput.*, 148(1):1–70, 1999.
2. Jesper Bengtson and Joachim Parrow. Formalising the *pi*-calculus using nominal logic. In *FoSSaCS*, volume 4423 of *Lecture Notes in Computer Science*, pages 63–77. Springer, 2007.
3. Michele Bugliesi, Giuseppe Castagna, and Silvia Crafa. Reasoning about security in mobile ambients. In *CONCUR*, volume 2154 of *Lecture Notes in Computer Science*, pages 102–120. Springer, 2001.
4. Luca Cardelli and Andrew D. Gordon. Mobile Ambients. *Theoretical Computer Science*, 240(1):177–213, 2000. Special Issue on Coordination, Daniel Le Métayer Editor.
5. Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–207, 1983.
6. Andrew Gordon and Luca Cardelli. Equational properties of mobile ambients. *Mathematical Structures in Computer Science*, 13(3):371–408, 2003.
7. Temesghen Kahsai and Marino Miculan. Implementing spi calculus using nominal techniques. In *CiE*, volume 5028 of *Lecture Notes in Computer Science*, pages 294–305. Springer, 2008.
8. Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.
9. Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Inf. Comput.*, 186(2):165–193, 2003.
10. Andrew M. Pitts. Alpha-structural Recursion and Induction. *Journal of the ACM*, 53:459–506, 2006.
11. Christian Urban, Julien Narboux, and Stefan Berghofer. The Nominal Datatype Package. http://isabelle.in.tum.de/nominal/. 2007.