Protocol Analysis via The Chase

A Major Qualifying Project Report
submitted to the faculty of

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements
for the degree of Bachelor of Science

in

Computer Science

by

JUSTIN POMBRIO

April 29, 2011

APPROVED:

Professor Joshua Guttman, project advisor

Professor Daniel Dougherty, project co-advisor

Professor Peter Christopher, project co-advisor

Abstract

We expound a method of analyzing cryptographic protocols using geometric logic and the Chase. Geometric logic is a formal system of logic comparable to first order logic, and the Chase is an algorithm which finds models for a given geometric logic theory. We use the Strand Space formalism as a model of protocol execution. Our work includes a rigorous translation of the Strand Space formalism, developed at MITRE, into geometric logic, a compiler that translates cryptographic protocols into geometric logic theories, and an algorithm for checking isomorphism between protocol executions in a special case in linear time.

Contents

Chapter 1

Introduction

One technique for analyzing a cryptographic protocol is to find a set of executions of the protocol that are representative of all possible executions. This method is used by other tools such as Scyther[4] and CPSA[13]. In the Strand Space formalism one execution is representative of another if there is a homomorphism from it to the other. Once these executions are enumerated, they can be used to reason about the protocol. For instance, if a message comes from a friendly source in all representative executions, then it must come from a friendly source in all possible executions. In general, if a property is preserved by homomorphisms and holds in each of the representative executions, then it must hold in *all* executions of the protocol.

We said above that the Chase finds a set of models of a geometric theory, but it does more than that. The set of models produced by the Chase is *jointly universal*, meaning that for any model M of the theory, there is a homomorphism from one of the models produced by the Chase to M. Notice the close analogy between this property of the Chase and the method of protocol analysis. This is our primary motivation for analyzing protocols via the Chase.

This study of protocol analysis also led our research to two purely mathematical areas. First, we extend an algorithm by Aho, Hopcroft, and Ullman for checking rooted tree isomorphism in linear time to work on a larger class of graphs. Specifically, we extend it to work on *single-inbound digraphs*: directed graphs in which every vertex has in-degree at most one.

Second, a number of graph-like structures, such as relational structures and directed graphs, are similar to graphs in that their homomorphism and isomorphism problems have the same complexities as the corresponding graph problems. We define and give several examples of *structure-reductions* - functions from one kind of structure to another which preserve homomorphisms and isomorphisms.

We also overcame challenges presented directly by the conversion of protocols into theories. The theory for any protocol can be divided into two parts. Part of the theory - the *dynamic* part - differs among protocols, but the majority of it - the *static* part - is exactly the same for any protocol. We implemented a protocol compiler, written in Haskell, which takes a protocol description in a standard format, and outputs the dynamic part of its theory. Upon discovering that the static part of the theory did not work well with the Chase, we modified it, and showed that the modified theory is logically equivalent to the original.

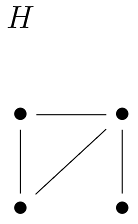
Chapter 2

Background

2.1 Graphs

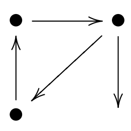
2.1.1 Graphs and Digraphs

A *graph* is a set of *vertices* together with a set of unordered pairs of vertices called *edges*. A finite graph can be drawn like so, with the dots denoting vertices and the line segments denoting edges,



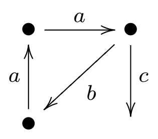
A *directed graph*, or *digraph*, is like a graph, but its edges have direction. Formally, a digraph is a set of vertices together with a set of ordered pairs of vertices called edges. Digraphs can be drawn like so,

H



Edge-labelled digraphs are digraphs whose edges have been arbitrarily labeled. Formally, an *edge-labeled digraph* over a set of labels L is a set of vertices V together with a set of ordered triples of the form (l, x, y), called edges, with $l \in L$ and $x, y \in V$.

H



We will refer to all of these graph-like objects as *structures*, and uniformly denote their vertices by $V(A)$ and their edges by $E(A)$.

In general, a *homomorphism* h from one structure A to another B is a mapping from the vertices of A to the vertices of B that preserves edges. Thus if $e \in E(A)$ then $h(e) \in E(B)$, where $h(e)$ is defined by applying h to each vertex in e. For instance, here is a drawing of a homomorphism from an edge-labeled digraph G to an edge-labeled digraph H. The numbers on the vertices of G describe which vertex in H they are being mapped to. Notice that not every vertex in H is the image of a vertex in G; this is not a requirement of a homomorphism.

$$\begin{tikzcd}
\bullet_3 \arrow[d, "b"'] & \bullet_2 \arrow[l, "b"] & \bullet_1 \arrow[l, "r"] \\
\bullet_1 \arrow[r, "r"] & \bullet_2 \arrow[r, "b"] & \bullet_3 \arrow[u, "b"]
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \bullet_1 \arrow[dl, "r"] \arrow[r, "g"] & \bullet \\
\bullet_2 \arrow[r, "b"] & \bullet_3 \arrow[u, "b"] &
\end{tikzcd}$$

An *isomorphism* ϕ from one structure A to another B is a homomorphism whose inverse is also a homomorphism. This means that ϕ is a bijection between the vertices of A and the vertices of B which preserves edges in both directions. Here is an example of an isomorphism between two graphs,

$$\begin{array}{cc} G & H \end{array}$$

$$\begin{tikzcd}
\bullet_1 \arrow[r, dash] \arrow[d, dash] & \bullet_3 \arrow[dl, dash] \\
\bullet_2 \arrow[r, dash] & \bullet_4
\end{tikzcd}
\qquad
\begin{tikzcd}
\bullet_1 \arrow[r, dash] \arrow[d, dash] & \bullet_2 \arrow[dl, dash] \arrow[d, dash] \\
\bullet_3 & \bullet_4
\end{tikzcd}$$

The homomorphism relation is *reflexive* because the identity function is always a homomorphism from a thing to itself. It is also *transitive*, because the composition of a homomorphism from X to Y and a homomorphism from Y to Z is itself a homomorphism from X to Z. Thus the homomorphism relation forms what is called a *quasi-order*. But homomorphisms are not, in general, antisymmetric (that is, they do not necessarily form a *partial order*). Take, for instance, the following digraphs

$$\begin{tikzcd}
\bullet \arrow[r] & \bullet & \bullet \arrow[l]
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
\bullet \arrow[r] & \bullet
\end{tikzcd}$$

These digraphs are not isomorphic, yet there are homomorphisms from each to the other. When there are homomorphisms from each of two things to the other, we call them *homomorphically equivalent.*

2.1.2 Relational Structures

Relational structures are generalizations of digraphs in which edges are colored and may have more (or fewer) than two vertices. An edge's color, however, fixes the number of vertices it has. Colors are called *relation symbols*, the number of vertices of an edge is called its *arity*, and the mapping from relation symbols to arities is called a *signature*.

More formally, a signature is a pair (S, A), where S is a set of objects called relation symbols, and A is a function from relation symbols to natural numbers, called the symbol's arity.

A *relational structure* over a given signature (S, A) is a set of objects called *vertices* and a set of *facts*. Each fact is a relation symbol R from S applied to an n-tuple of vertices, where n is $A(R)$. A fact with relation symbol R applied to vertices $x_1, ..., x_n$ is written $R(x_1, ..., x_n)$.

2.1.3 Cores of Graphs

Core An object for which every endomorphism is also an automorphism.

Antichain A set of objects unrelated by homomorphisms.

The concept of the *core* of a graph is important when studying homomorphisms, because the image of a core under a homomorphism must be an identical core. This property follows immediately from the definition of a core. The following theorems appear in class notes from Peter Cameron[3]. The proofs are largely our own, however, and do not assume the graphs are finite as Cameron's proofs do.

Lemma 1. *A homomorphism equivalence class has at most one core.*

Proof. If an equivalence class has two cores, then there are homomorphisms from each to the other, ϕ and ϕ'. Consider the compositions $\phi \circ \phi'$ and $\phi' \circ \phi$. The first is an endomorphism from the first object to itself, and hence an automorphism, and the second is an endomorphism from second object to itself, hence an automorphism. Since both $\phi \circ \phi'$ and $\phi' \circ \phi$ are bijections, so are ϕ and ϕ'. Now we can show that ϕ is an isomorphism. We already know that it is a bijective homomorphism, so we need only show that it's inverse ϕ^{-1} is a homomorphism. ϕ^{-1} is equal to $(\phi' \circ \phi)^{-1} \circ \phi'$, which is the composition of an automorphism and a homomorphism, which is a homomorphism. Thus ϕ is an isomorphism and the equivalence class's cores are isomorphic.

We have made use of the fact that if $f \circ g$ and $g \circ f$ are both bijections, then f and g are bijections. This claim warrants demonstration. Let $f :: A \to B$ and $g :: B \to A$. First, since $f \circ g$ is onto, for all a in A, $\exists x \in A.\ f(g(x)) = a$. Thus $\exists b \in B.\ f(b) = a$, namely $b = g(x)$, and f is onto. And since $f \circ g$ is one-to-one, for all x and y in A, $g(x) = g(y) \implies f(g(x)) = f(g(y)) \implies x = y$, so g is one-to-one. By symmetry, f must also be one-to-one and g must also be onto, so both f and g are bijections. $\square$

Lemma 2. *A core is uniquely represented as an antichain of connected cores.*

Proof. Every core is the disjoint union of some connected components. Each component must be a core, or else it would have an endomorphism which is not an automorphism and so would the whole object. Likewise, there can be no homomorphism between components, since it could be used to construct an endomorphism which is not an automorphism by mapping one component to the other, and every other component to itself. Thus the components of any core, which are themselves connected cores, form an antichain. $\square$

Lemma 3. *A graph G is uniquely represented as the infinite sequence $|Hom(F_i, G)|$ for any enumeration of all finite graphs F_i.*

In "Homomorphisms on Infinite Directed Graphs"[2], Bauslaugh points out that cores ought be defined as graphs for which every endomorphism is an automorphism, and *not* as a vertex-minimal member of a graph homomorphism equivalence class as suggested by Cameron[3]. For finite graphs, these definitions are equivalent, but for infinite graphs only the latter results in cores being unique. Consider, for instance, the (countably) infinite graph with vertices $\{0, 1, 2, ...\}$ and edges $\{(x, y)|x < y\}$. Under the vertex-minimal core definition, this graph has an infinite number of cores, given by the subgraphs induced by $\{n, n + 1, n + 2, ...\}$ for any $n \geq 1$. These are in the same homomorphism equivalence class: a forward homomorphism maps x to $x+n$, and a reverse homomorphism maps x to x. And each core is indeed vertex minimal: they each have infinitely many vertices, and there is no homomorphism to any finite graph, since that graph would have to include a clique of every order.

2.2 Geometric Logic

Geometric logic is a formal system of logic comparable to first order logic. A formula in geometric logic, called a *geometric theory*, is a conjunction of implications between positive existential formulas. *Positive existential* formulas are similar to a first order logic formulas, but lack negation and universal quantification. They may also have infinitary disjunctions, though we will not make use of this fact. The form of geometric theories makes them particularly amenable to logical analysis.

2.2.1 Definitions

A *signature* is a set of *function symbols* F together with a set of *relation symbols* R and a mapping arity :: $F \cup R \to \mathbb{N}$. The arity function will give the number of arguments each relation symbol and function symbol takes. Fix a signature (F, R, arity) and an infinite set of variables V.

1. A *term* is either a variable $x \in V$ or a function application $f(t_1, ..., t_n)$ where $f \in F$ and $\text{arity}(f) = n$ and $t_1, ..., t_n$ are terms.

2. An *atom* has the form $r(t_1, ..., t_n)$, where $r \in R$, $\text{arity}(r) = n$, and $t_1, ..., t_n$ are terms.

3. An *atomic formula* is true, or false, or an equation over terms $t_1 = t_2$, or an atom.

4. A *conjunctive formula* has the form $\alpha_1 \wedge ..., \alpha_n$, where $\alpha_1, ..., \alpha_n$ are atomic.

5. *Positive existential formulas* are the closure of existential quantifiers, finitary conjunctions, and infinitary disjunctions over atomic formulas. More formally, a positive existential formula is,

   - An atomic formula, or
   - $\exists x.\alpha$, where α is positive existential, or
   - $\alpha \wedge \beta$, where α and β are positive existential, or
   - $\bigvee_{i \in I} \alpha_i$, where I is any set and each α_i is positive existential.

6. A *geometric logic formula* has the form $\alpha \Rightarrow \beta$, where α and β are positive existential. A geometric logic formula may be open, and if so it is implicitly universally quantified over its free variables.

7. A *geometric logic theory*, *geometric theory*, or just *theory*, is a set of geometric logic formulas. Semantically, it is the conjunction of those formulas.

8. A *relational structure* consists of a set D called its *domain* together with an *interpretation* function I that assigns a meaning to each relation symbol and function symbol. Specifically, I assigns a function of type $D^n \rightarrow D$ to each function symbol of arity n and a relation of type D^n to each relation symbol of arity n.

9. A relational structure M *satisfies* a geometric theory T, written $M \models T$ if every mapping from the free variables of the theory to domain elements of the structure makes the formulas of the theory true, under the usual semantics of logic. We will also write $M \models_E T$ to mean

that M makes the formulas of T true under the particular mapping E. Any such mapping from variables to domain elements is called an *environment*. The *models* of a theory are the relational structures which satisfy it. Two theories are *equivalent* if they have the same set of models.

10. A *homomorphism* from one relational structure to another is a mapping from the domain of the first to the domain of the second that preserves functions and relations. We write $A \xrightarrow{h} B$ to mean that h is a homomorphism from A to B, and $A \to B$ to mean that some homomorphism exists. In the simple case that A and B contain no function symbols, then for all relation symbols R,

    $$\text{If } (x_1, ..., x_n) \in A(R) \text{ then } (h(x_1), ..., h(x_n)) \in B(R)$$

    A homomorphism $A \to B$ should capture the idea that A is a generalization of B.

11. Let S be a set of models of a theory T. If for every model M of T there is a model $M' \in S$ such that $M' \to M$, then we call S *jointly universal.* If M_1 can be thought of as representative of M_2 whenever $M_1 \to M_2$, then a jointly universal set of models of a theory is representative of *all* models of a theory. This fact will be fundamental to our approach to protocol analysis.

2.2.2 Eliminating Redundant Constructs

A number of the "features" a geometric logic are redundant, in that any geometric theory can be written without them. Either functions or existential quantifiers can be eliminated in lieu of the other, and equations may be eliminated outright.

This rewriting will take a geometric theory T and produce another theory

T' whose formulas do not use functions (or existential quantifiers, or equations), but whose signature is an extension of the signature of T. It would be convenient to say that T and T' are equivalent, but equivalence not well defined because their signatures differ. All that can be said is that they are effectively equivalent in that there is a mapping from models of T to models of T' that preserves homomorphisms in both directions.

Existential quantifiers may be eliminated in a very simple way. For each existentially quantified formula $\exists x.\alpha$ with free variables $\vec{y}$, introduce a new function symbol f and return $\alpha[f(\vec{y})/x]$ (that is, α with all occurrences of x replaced by $f(\vec{y})$).

Equations $t_1 = t_2$ may be eliminated from a function-free theory by introducing a relation E to describe equality. First, equality must be reflexive, commutative, and transitive, so introduce the formulas,

$$\begin{aligned} \text{true} &\Rightarrow E(x,x) \\ E(x,y) &\Rightarrow E(y,x) \\ E(x,y) \wedge E(y,z) &\Rightarrow E(x,z) \end{aligned}$$

Second, the relations of the theory must be well-defined with respect to equality, so for each relation R of arity n and for each parameter position $1 \leq i \leq n$ of R, add the formula

$$R(x_1, ..., x_n) \wedge E(x_i, y) \Rightarrow R(x_1, ..., x_{i-1}, y, x_{i+1}, ..., x_n)$$

Finally, replace each equation $t_1 = t_2$ with $E(t_1, t_2)$.

Conversely, functions may be eliminated from a geometric theory in lieu of existential quantifiers by the following transformation. For each function symbol f of arity n, introduce a new relation symbol F of arity $n{+}1$ to denote that "the last argument is equal to f applied to the first n arguments". Since functions should be complete and well-defined, add the formulas,

$$\text{true} \Rightarrow \exists y.F(x_1, ..., x_n, y)$$
$$F(x_1, ..., x_n, y) \wedge F(x_1, ..., x_n, y) \Rightarrow x = y$$

Now any formula that makes use of f, such as

$$R(f(x, x, x)) \Rightarrow \exists y.R(f(x, y, z))$$

may be replaced by

$$F(x, x, x, a) \wedge R(a) \Rightarrow \exists y.F(x, y, z, b) \wedge R(b)$$

For conceptual clarity, we will focus on function-free theories with the intention that theories that do make use of functions can always be transformed into ones that do not.

2.2.3 C.D.E. Form

Lemma 4. *Positive existential formulas can always be written in* D.E.C *form (for* disjunction of existentials*).*

$$\bigvee_{i \in I} \exists \vec{x}_i \alpha_i$$

where α_i are conjunctive formulas.

Proof. Rewrite a formula using the following identities,

$$\begin{aligned}
(\exists x.\alpha) \wedge \beta &= \exists x.(\alpha \wedge \beta) \\
\left(\bigvee_{i\in I} \alpha_i\right) \wedge \beta &= \bigvee_{i\in I}(\alpha_i \wedge \beta) \\
\exists x.\left(\bigvee_{i\in I} \alpha_i\right) &= \bigvee_{i\in I} \exists x.\alpha_i
\end{aligned}$$

□

Likewise, geometric logic theories can always be written in such a way that their formulas have the form

$$\alpha \to \bigvee_{i\in I} \exists \vec{x}_i \beta_i$$

where α and β are conjunctive. We will call this the *C.D.E* form.

Lemma 5. *For every geometric theory T, there is an equivalent theory T' in C.D.E. form.*

Proof. To see this, first write the left and right parts of each formula in the form $\bigvee_{i\in I} \exists \vec{x}_i \alpha_i$, then consider the following equivalences,

$$\begin{aligned}
\bigvee_{i\in I} \exists \vec{x}_i \alpha_i \Rightarrow \bigvee_{j\in J} \exists \vec{y}_j \alpha_j &= \bigvee_{i\in I} \alpha_i \Rightarrow \bigvee_{j\in J} \exists \vec{y}_j \alpha_j \\
&= \bigwedge_{i\in I}\left(\alpha_i \Rightarrow \bigvee_{j\in J} \exists \vec{y}_j \alpha_j\right)
\end{aligned}$$

□

2.3 The Chase

The Chase is an algorithm which finds a jointly universal set of models of a geometric theory. It is based upon a nondeterministic algorithm which may find a single model of a theory (or may fail). This nondeterministic algorithm is called a *run* of the Chase. We will describe the Chase in terms of theories expressed without function symbols, and with only finite disjunctions; it can be extended to cover these possibilities.

2.3.1 Chase Runs

The behavior of a run of the Chase is intuitive. It keeps a model and repeatedly picks an unsatisfiable formula from the theory and augments the model to satisfy one of the disjuncts. If all of the formulas of a theory become satisfied, a model of the theory has been found and is returned. On the other hand, if an unsatisfied disjunct (such as $\bot$) is chosen, the run fails.

An example of a Chase run may be clarifying. Consider the following simple theory about message transmissions in the Strand Space Formalism:

```
Send(n, m) => Node(n) & Mesg(m)
Recv(n, m) => Node(n) & Mesg(m)
Node(n) => Exists m. Send(m)
        | Exists m. Recv(m)
Send(n, m1) & Send(n, m2) => m1 = m2
Recv(n, m1) & Recv(n, m2) => m1 = m2
Send(n, m1) & Recv(n, m2) => false
Recv(n, m) => Exists n'. Send(n', m)
```

The relation `Send(n, m)` means that n is a transmission node and $msg(n) = m$, and `Recv(n, m)` means that n is a reception node with $msg(n) = m$.

Let's begin with the model

```
{ Node(0) }
```

By formula 3 with binding $n \to 0$, choosing the second disjunct,

`{ Node(0), Recv(0, 1) }`

By formula 2 with binding $n \to 0, m \to 1$,

`{ Node(0), Recv(0, 1), Mesg(1) }`

By formula 7 with binding $n \to 0, m \to 1$,

`{ Node(0), Recv(0, 1), Mesg(1), Send(2, 1) }`

By formula 2 with binding $n \to 2, m \to 1$,

`{ Node(0), Recv(0, 1), Mesg(1), Send(2, 1), Node(2) }`

All formulas are now satisfied, so the run has found a model of the theory.

2.3.2 The Chase Algorithm

The Chase as a whole works by branching to explore many runs simultaneously. It keeps track of several models and repeatedly selects a model, then selects a formula and environment which that model does not satisfy. Next it constructs several new models from the old one by forcing each disjunct, in turn, to be satisfied, and adds this models to its list.

2.3.3 Algorithms

Chase Runs
A *run* of the Chase is a nondeterministic algorithm which takes a geometric theory and may either produce a model of that theory or fail.

```
# run :: Theory -> Model | Fail
run(thy) = loop(emptyModel) where
loop(M) =
  if satisfies(M, thy)
  then yield M
```

```
else Let (E, 'A => B1 | ... | Bn') = pickFormula(M, thy)
     Nondeterministically choose a natural number i in 1..n
     case coerce(Bi, M) of
       Fail → Fail
       M' → loop(M')
```

`pickFormula`(M, T) nondeterministically picks an environment E and formula $f \in T$ such that M does not satisfy f under E.

It is essential that the Chase never leave a (formula, environment) pair unevaluated forever. An implementation that always eventually examines any (formula, environment) pair is called *fair*, and one that doesn't is called *unfair*. We will continue to assume that Chase implementations are fair, since their universality property relies on this.

Definition 1. *Let r be an infinite run of the chase on theory T which has models $M_0 \to M_1 \to M_2 \to ...$ at each step as it runs, with M_0 being the empty model. r is* fair *if for all i, whenever f is a formula of T and E is an environment over the domain of M_i such that $M_i \not\models_E f$, there is a step $M_j \to M_{j+1}$ for some $j \geq i$ in which `pickFormula`(M_j, T) returns (E, f).*

A nonterminating fair run of the Chase will yield as its limit an infinite model of the theory. In this way, nonterminating runs of the Chase are correct, though not very helpful in practice.

Disjunct Coercion

The coercion algorithm takes a conjunctive formula f, a model M, and an environment E and either produces a new model M' such that $M \to M'$ and $M' \models_E f$ or fails. It is most easily described as a stateful algorithm over M and E (that is, it will modify M and E as it runs, and its output is the value of M when it finishes).

```
# algorithm
# coerce :: Conjunctive Formula -> Model -> Environment -> Model | Fail
coerce('true') = return;
coerce('false' = FAIL;
coerce('R(x1, ..., xn)') =
  Add (E(x1), ..., E(xn)) to the interpretation of R in M;
  return;
coerce('x = y') =
  Replace all occurrences of E(x) with E(y) in M;
  Bind x to E(y) in E;
  return;
coerce('a & b') =
  coerce(a);
  coerce(b);
  return;
coerce('Exists x. a') =
  Generate a fresh domain element c in M;
  Bind x to c in E;
  coerce(a);
  return;
```

The Chase

```
# chase :: Theory -> [Model]
chase(thy) = loop([emptyModel]) where
loop(models) =
  if empty(models)
  then []
  else let M = first(models)
       if satisfies(M, thy)
       then yield M
            loop(rest(models))
       else let (E, 'A => B1 | ... | Bn') = pickFormula(M, thy)
                newModels = []
            for i = 1..n do
              case coerce(M, E, Bi) of
                Fail → do nothing
                M' → add M' to newModels
            loop(models ++ newModels)
```

2.3.4 Theorems

We prove here two theorems claimed in the notes[7] of Professor Dougherty of WPI.

Lemma 6. *If a step of the Chase changes M to M', then $M \to M'$.*

Proof. Let h be the identity homomorphism from M to M. When coercing β_i, update the homomorphism h. Let E' be an extension of $h \circ E$ under which both α and β_i are true. The only danger is that the act of coercing β_i will invalidate the homomorphism h. There are three cases in which this could occur. First, if an atomic formula $R(x_1, ..., x_n)$ is encountered, coercion will add $(E(x_1), ..., E(x_n))$ to R_M. But $R(x_1, ..., x_n)$ is already satisfied by M' under $h \circ E$. Second, if an existential formula $\exists x.\gamma$ is encountered, coercion will bind x to a fresh domain element in M. In this case, set $h(E(x)) = E'(x)$. Third, if $x = y$ is coerced, $E(x)$ and $E(y)$ will be identified. But since M' already satisfies this formula under $h \circ E$, $h(E(x)) = h(E(y))$, so nothing needs to be done. □

Theorem 1. *For any model M' of a theory T, there is a run of the Chase on T which yields a model M such that $M \to M'$.*

Proof. We will use the model M' as an *oracle* to guide the run. Keep a map h from the domain of the Chase model M to the domain of M'. To begin, the Chase model is empty and so is h. As the Chase progresses, we will maintain the invariant that h describes a homomorphism $M \to M'$. Suppose the Chase picks a formula $f = \alpha \Rightarrow \beta_1 \vee ... \vee \beta_n$ and environment E. $M \models_E \alpha$, so $M' \models_{h \circ E} \alpha$. Since M' satisfies the theory, we also know $M' \models_{h \circ E} f$. To satisfy both α and β, M' must satisfy one of the disjuncts, say β_i. So $M' \models_{h \circ E} \beta_i$. Have the Chase run choose β_i. The homomorphism h can be updated per the above lemma. □

Theorem 2. *A geometric theory is satisfiable if and only if there is a fair run of the Chase which does not fail.*

Proof. First, see that if a geometric theory is satisfiable then there is a run of the Chase which does not fail. This follows from the previous theorem - the theory is satisfiable, so it has a model M, so there is a run of the Chase that produces a model M' with $M' \rightarrow M$.

Next, see that if a geometric theory is unsatisfiable then every fair run of the Chase fails. Clearly no run could succeed, because it succeeds only when it finds a model of the theory. Nor could the Chase run forever, because then the theory would be satisfied by the model formed by taking the union of the structures formed by the Chase at each iteration (by the chain theorem above). $\square$

2.4 The Strand Space Formalism

The *Strand Space formalism* is a mathematical model for formally reasoning about cryptographic protocols. It was developed by researchers at MITRE, including Joshua Guttman, Javier Thayer, and Jonathan Herzog, beginning in 1998 [14] [11] [9]. Our description of strand spaces is largely adapted from the relevant publications.

The Strand Space formalism distinguishes between two different kinds of participants: regular participants and an adversary. A single physical entity can be represented as multiple regular strands if it executes more than one session.

Participants in a protocol run are represented by *strands*, and communicate with each other by sending and receiving messages. A regular participant is represented by a regular strand and must follow the protocol. The adversary is represented by zero or more adversary strands, and can manipulate the messages that regular strands send and receive.

The actions of both the regular participants and the adversary are abstracted into message passing; this is assumed to be able to capture all relevant information. For instance, if a private key is assumed insecure and the adversary may be able to learn it, this could only be represented by an adversary strand receiving the key. As such, every strand consists of a (non-empty) sequence of message passing events called *nodes*. Each node either sends a message or receives a message. A *term* is any possible message.

In our representation of this formalism, nodes and terms are the first-class objects. Strands are represented only indirectly through nodes.

2.4.1 Messages

The Strand Space formalism admits different message algebras. In the algebra we will use, a message may either be *basic* and indivisible, or it may be the encryption or pairing of two other messages. To be precise,

- Every *basic message* is a message.
- If x and y are messages, then the pair (x, y) is a message.
- If x and k are messages, then the encryption $\{|x|\}_k$ is a message. We call x the *plain-text* and k the *key*.

We call the pairing operation *pairing* rather than *concatenation* because it is not assumed to be associative.

One message is an *ingredient* of another, written $m_1 \sqsubseteq m_2$ when it can be obtained by repeated unpairing and decryption operations. Formally, $\sqsubseteq$ is defined inductively by,

- $x \sqsubseteq x$
- $x \sqsubseteq (x, y)$
- $y \sqsubseteq (x, y)$
- $x \sqsubseteq \{|x|\}_y$
- If $x \sqsubseteq y$ and $y \sqsubseteq z$ then $x \sqsubseteq z$

The *sub-message* relation, written $m_1 \ll m_2$ is like the ingredient relation, except that an encryption key is considered a sub-message of an encryption.

- $x \ll x$
- $x \ll (x, y)$
- $y \ll (x, y)$
- $x \ll \{|x|\}_y$
- $y \ll \{|x|\}_y$
- If $x \ll y$ and $y \ll z$ then $x \ll z$

A message m *originates* at a node n if n is a transmission node, $m \sqsubseteq msg(n)$, and whenever $n' \Rightarrow^+ n$, $m \not\sqsubseteq msg(n')$. A message *uniquely originates* if it originates at a unique node, and it is *non-originating* if it does not originate at any node.

We are going to assume that unspecified messages are basic. For instance, if a protocol specification states that a principal A receives a message $\{|x|\}_{k_A}$ where k_A is A's public key and x is unspecified, x may *not* be a pair (y, z). And if A receives a message of the form $\{|(y, z)|\}_{k_A}$ for any y and z, the receiving strand should terminate. We make this assumption for simplicity. It makes it easier to unify messages. Removing it may be a consideration for future work.

A *homomorphism* on messages is a function $h :: M_1 \to M_2$ such that

- $h(x)$ is basic whenever x is basic
- $h((x, y)) = (h(x), h(y))$
- $h(\{|x|\}_k) = \{|h(x)|\}_{h(k)}$

This definition of message homomorphisms is *not* identical to the usual Strand Space definition, simply because in our algebra indeterminants must be basic, while in the proper definition they may be replaced by any message.

2.4.2 Strand Spaces

Fix a set of possible messages M. A *directed message* is a pair (m, d) such that $m \in M$ and d is either + or -. We will call nodes for which $d = -$ *reception* nodes, and nodes for which $d = +$ *transmission* nodes. We will write $msg(n) = m$ and $dir(n) = d$ when $n = (m, d)$. Let D denote the set of all directed messages. A *Strand Space* is a set of *strands* Σ together with a partial function $trace :: \Sigma \rightarrow D^+$ from strands to finite sequences of directed messages. Strand represent participants in the protocol, and directed messages are their actions: message transmissions and receptions.

Now fix a strand space (Σ, tr). A *node* is a pair (s, n) such that $s \in Sigma$ and $n \in \mathcal{N}$ such that n is no larger than the length of s. We will write $msg((s, n))$ to mean the nth directed message of s. We will say that one node n_1 is the *parent* of another node n_2 when they are in the same strand and n_1 immediately precedes n_2; in other words, when $n_1 = (s, i)$ and $n_2 = (s, i + 1)$ for some strand $s \in \Sigma$ and natural number $i \in \mathcal{N}$.

2.4.3 Infiltrated Skeletons

We ended up using a model of protocol executions that we call *infiltrated skeletons*. An infiltrated skeleton consists of partially executed regular and adversarial strands, a set of *links* on the nodes of the strands, a set of messages *unique* representing unique-origination assumptions, and a set of messages *non* representing non-origination assumptions. For those familiar with Strand Space terminology, an infiltrated skeleton can be thought of as a combination of a bundle and a skeleton; it contains both the adversarial strands of a bundle and the origination assumptions of a skeleton. The following definition is largely an adaption of the definition of a skeleton in the Strand Space formalism[9]. Formally, an infiltrated skeleton over a set of messages M consists of,

- A finite set of (regular and/or adversarial) nodes N

- A binary relation $\rightarrow$ on N
- A finite set of basic messages *non*
- A finite set of basic messages *uniq*
- A partial order $\leq$ on N

such that

- If $n_1 \Rightarrow n_2$ and $n_2 \in N$ then $n_1 \in N$.
- If $n_1 \rightarrow n_2$ then $dir(n_1) = +$, $dir(n_2) = -$, and $msg(n_1) = msg(n_2)$.
- Whenever $dir(n) = -$, there is another node n' such that $n' \rightarrow n$.
- $\leq$ is a subset of the reflexive transitive closure of $\rightarrow \cup \Rightarrow$.
- $\forall k \in non. \forall n \in N. k \not\sqsubseteq msg(n)$
- $\forall k \in non. \exists n \in N.$ either $k \ll msg(n)$ or $k^{-1} \ll msg(n)$.
- $\forall a \in uniq. \exists n \in N. a \ll msg(n)$
- $\forall a \in uniq.$ if a originates at $n \in N$ then
  - a originates at no other node in N
  - If $a \ll msg(n')$ where $n' \in N$, then $n \leq n'$ (where $\leq$ is the reflexive transitive closure of $\rightarrow$ and $\Rightarrow$)

We will sometimes refer to infiltrated skeletons simply as *skeletons*. When we do, we are speaking loosely, and are *not* referring to traditional Strand Space skeletons.

A *link* is an edge from a sending node to a receiving node that have the same message. If there is a node n with a link to a node m, n sends m a message and m receives it unaltered. There is also a partial order $\leq$ on nodes which is a superset of the transitive closure of $\rightarrow$ and $\Rightarrow$. We say that a node n_1 *precedes* another node n_2 when $n_1 \leq n_2$.

2.4.4 Homomorphisms

A *homomorphism* from one infiltrated skeleton A to another B is a pair (α, ϕ), where α is a homomorphism on messages and $\phi :: N_A \to N_B$ such that

1. Whenever $n_1 \Rightarrow n_2$, $\phi(n_1) \Rightarrow \phi(n_2)$
2. $\forall n \in N_A . dir(n) = dir(\phi(n))$
3. $\forall n \in N_A . msg(\phi(n)) = \alpha(msg(n))$
4. $n_1 \leq n_2$ implies $\phi(n_1) \leq \phi(n_2)$
5. $\alpha(non_A) \subseteq non_B$
6. $\alpha(uniq_A) \subseteq uniq_B$
7. If $a \in uniq_A$ and a originates at n, then $\alpha(a)$ originates at $\phi(n)$

2.4.5 Protocols

In the Strand Space formalism, a protocol is a finite set of strands called *roles*, together with a set of unique-origination constraints and a set of non-origination constraints. The roles describe the actions of the regular participants in the protocol. A *unique-origination constraint* over a set of roles mentions a message appearing in one of the roles (possibly as a sub-message). It denotes the assumption that that message originates only in that strand. For instance, if a protocol involves one of the principals generating a fresh, never-seen-before nonce, that could be captured by a unique origination constraint on the message representing that nonce in the role of that principal. Likewise, a *non-origination constraint* mentions a message appearing in one of the roles, and it denotes the assumption that that message does not originate in any strand. When a principal is trusted only to execute a role with a well-kept secret, this can be expressed as a non-origination constraint on that secret.

2.4.6 The Adversary

The Adversary in the Strand Space formalism is of typical Dolev-Yao style. He has full control of the network (and may thus obtain any message transmitted by a regular strand), and is capable of the following operations:

Generation The Adversary may obtain any basic message that is not otherwise assumed to be secure.

Unpairing If the Adversary can obtain the pair (x, y), then he may deconstruct it to obtain x and y individually.

Decryption If the Adversary can obtain the encryption $\{|x|\}_k$ and the decryption key k^{-1}, he can also obtain the plain-text m.

Pairing If the Adversary can obtain messages x and y, then he may also obtain the pair (x, y).

Encryption The Adversary may perform an encryption to obtain the message $\{|x|\}_k$ if he has obtained the plain-text m and the encryption key k.

The capabilities of the Adversary can be captured by strands that perform the operations. A pairing operation, for instance, may be represented by a strand with three nodes: the first receives a message x, the second receives a message y, and the third transmits the message (x, y). These classes of strands are called *adversarial strands*, and are shown below. We call these strands *generation*, *pairing*, *encryption*, *unpairing*, and *decryption* strands respectively. They can be divided into two categories: those that involve constructing a message and those that involve deconstructing a message to obtain its ingredients. We will call generation, pairing, and encryption strands *constructive*, and we will call unpairing and decryption strands *deconstructive*. Likewise, a *constructive node* is one that is part of a constructive strand, and a *deconstructive node* is one that appears on a deconstructive strand.

$$\begin{tikzcd}
\bullet \arrow[r, "x"] & {} & {} \arrow[r, "(x{,}y)"] & \bullet \arrow[d, Rightarrow] & {} & {} \arrow[r, "\{|x|\}_k"] & \bullet \arrow[d, Rightarrow] & {} & {} \arrow[r, "x"] & \bullet \arrow[d, Rightarrow] & {} & {} \arrow[r, "x"] & \bullet \arrow[d, Rightarrow] & {} \\
 & & & \bullet \arrow[r, "x"] \arrow[d, Rightarrow] & {} & {} \arrow[r, "k^{-1}"] & \bullet \arrow[d, Rightarrow] & & {} \arrow[r, "y"] & \bullet \arrow[d, Rightarrow] & & {} \arrow[r, "k"] & \bullet \arrow[d, Rightarrow] & \\
 & & & \bullet \arrow[r, "y"] & {} & & \bullet \arrow[r, "x"] & {} & & \bullet \arrow[r, "(x{,}y)"] & {} & & \bullet \arrow[r, "\{|x|\}_k"] & {}
\end{tikzcd}$$

2.4.7 Analysis

The Strand Space formalism is a well-established mathematical model of cryptographic protocols. It can be used to reason about protocols and their possible executions, and under the assumption of ideal cryptography, any conclusion proved using the formalism will be true. This makes it an excellent model with which to perform protocol analysis.

Chapter 3

Protocol Analysis

Our approach to protocol analysis has two stages. In the first, a protocol is translated into a geometric theory such that the models of the theory represent infiltrated skeletons describing the possible executions of the protocol. In the second stage, the Chase is used to find a jointly universal set of models, which can be interpreted as possible executions of the protocol.

The theory for any protocol can be divided into two parts: a small *dynamic* set of formulas that vary between protocols, and a larger *static* set of formulas that are always the same. We generate the dynamic formulas via our compiler, and we wrote the static formulas by hand. Our protocol compiler takes as input a protocol, written as an S-Expression, in a format also used by some other protocol analysis tools, such as Scyther[4] and CPSA[13]. It outputs the dynamic portion of a theory, such that the models of the combined theory represent the possible executions of the protocol. The source for the compiler itself is in appendix D. A sample compiler input - that for the Needham-Schroeder protocol - can be found in appendix A, and its output may be found in appendix B.

The Needham-Schroeder protocol is a simple communication protocol between two parties. It is often used as an example because it contains a vulnerability which is not obvious. Below is a drawing of a correct execution of the protocol, as it would be represented in the Strand Space formalism.

$$\begin{tikzcd}
A & & B \\
\bullet \arrow[rr, "\{\mid N_A{,}A\mid\}_{K_B}"] \arrow[d, Rightarrow] & & \bullet \arrow[d, Rightarrow] \\
\bullet \arrow[d, Rightarrow] & & \bullet \arrow[ll, "\{\mid N_A{,}N_B\mid\}_{K_A}"'] \arrow[d, Rightarrow] \\
\bullet \arrow[rr, "\{\mid N_B\mid\}_{K_B}"] & & \bullet
\end{tikzcd}$$

$$N_A, N_B \in uniq$$

$$K_A^{-1}, K_B^{-1} \in non$$

Here K_A and K_B are the public asymmetric keys of A and B, and N_A and N_B are large randomly generated messages called *nonces.* They can be assumed to originate uniquely because of the extreme improbability of the same message being generated by someone else. There is a vulnerability in this protocol from B's perspective, but not from A's perspective. We won't delve into the attack, other than to say that it stems from the fact that A never announces her intention to communicate with B, and can be fixed by adding B to the second message.

3.1 The Correctness Criterion

The fact that the models returned by the Chase are jointly universal is fundamental to our purpose. Since we use models to represent protocol executions, we would like the set of executions returned by the Chase to be representative of *all* possible executions of a protocol. Specifically, we would like them to be jointly universal under skeleton homomorphisms. This leads to the following correctness criterion.

For any protocol P, let T_P be the geometric theory obtained by taking the union of the output of the compiler with the static theory. We would like the finite models of T_P to mimic as closely as possible the infiltrated skeletons of P. Specifically, there should be a mapping from the infiltrated skeletons of P to the finite models of T_P with a one-to-one correspondence between the homomorphisms of the skeletons and the homomorphisms of the models.

This idea is captured by the notion of a *full and faithful functor* from category theory. A *category* is a set of objects together with a set of *morphisms* between them that obey certain axioms; for a full definition see [12]. We will only concern ourselves with categories whose morphisms are homomorphisms.

Definition 2. *A* functor *$\mathcal{F}$ from a category A to a category B is a mapping that,*

- *Associates to each object $a \in A$ an object $b \in B$*
- *Associates to each morphism $x \to y \in A$ a morphism $\mathcal{F}(x) \to \mathcal{F}(y) \in B$*

such that,

- *$\mathcal{F}$ maps identity morphisms to identity morphisms*
- *If g is a morphism $x \to y \in A$ and h is a morphism $y \to z \in A$ then $\mathcal{F}(g \circ h) = \mathcal{F}(g) \circ \mathcal{F}(h)$*

$\mathcal{F}$ is faithful *if it maps distinct morphisms to distinct morphisms. It is* full *if for every morphism $h : \mathcal{F}(x) \to \mathcal{F}(y)$ there exists an $h' : x \to y$ such that $\mathcal{F}(h') = h$. It is* invertible *if for every object $b \in B$, there exists an object $a \in A$ such that $\mathcal{F}(a) = b$.*

Definition 3. *For every protocol P, let $\mathcal{S}_P$ be the category whose objects are the infiltrated skeletons of P and whose morphisms are the homomorphisms* *among those skeletons. Likewise, let $\mathcal{M}_P$ be the category whose objects are the* finite *models of T_P and whose morphisms are the homomorphisms over those models.*

Definition 4. *A theory T* correctly represents *a protocol P if there exists an invertible full and faithful functor $\mathcal{F}$ from $\mathcal{S}_P$ to $\mathcal{M}_P$. We say that $\mathcal{F}$* witnesses *the correctness of T_P.*

We can now state the correctness criterion.

Theorem 3 (Correctness)**.** *For any protocol P, T_P correctly represents P.*

Corollary 1. *Let P be any protocol, let $\mathcal{F}$ be a witness of the correctness of T_P, and let M be a set of jointly universal models of T_P. Then $\mathcal{F}^{-1}(M)$ is a set of jointly universal infiltrated skeletons of P.*

We believe this theorem to be true, though it is infeasible to prove it. Such a proof would involve formally reasoning about our compiler - a nontrivial Haskell program. We will, however, present evidence in favor of this theorem in sections 3.4 to 3.4.2. Specifically, we will explain how to construct the functor $\mathcal{F}$ from the above theorem, and argue that it will be full, faithful, and invertible.

3.2 Normalization and Efficiency

In the Strand Space Formalism, the Adversary is often capable of accomplishing the same task in multiple ways. Without loss of generality, however, we can assume that he refrains from certain sequences of actions that are unproductive. Guttman and Thayer[10] proved that two such conditions, *normality* and *efficiency* do not limit the adversary's capabilities.

The *efficiency* condition requires that the adversary always takes a message from the earliest point at which it appears. To be precise, if $n_1 \leq n_2$, $dir(n_1) = dir(n_2) = +$, $msg(n_1) = msg(n_2)$, and $n_2 \rightarrow n_3$, then $n_1 = n_2$.

The second condition is called *normalization* and requires that the adversary never performs a deconstructive operation (unpairing or decryption) immediately following a constructive operation (pairing or encryption). To be precise, this is saying that there is never a link (n_1, n_2) such that n_1 is a deconstructive node and n_2 is a constructive node.

We added formulas expressing the normalization and efficiency constraints to the static portion of the theory. These two conditions help prune the adversarial actions the Chase will search for and may thereby increase the number of protocols for which it will terminate.

3.3 Chains

Even after introducing the normalization and efficiency constraints, the Chase generally never terminated on any protocol. The culprit was a single formula that states that when a node receives a message, some other node must have sent it. We will call this the *link formula.*

```
Recv(n, m) => Exists n'. Link(n', n)
```

This led the Chase to explore the possible nodes that could have sent the message, which include nodes on strands that include another reception node. The same formula can be applied to this new reception node, and so forth.

The solution is a concept put forth by Cas Cremers[4] [5] called *chaining.* The chaining condition is a generalization of the normalization condition. Simply put, the idea is that the ingredients of a message constructed by the adversary can always be traced back to regular strands.

To describe chains in more detail, we will need some terminology. A *path* is a sequence of nodes $n_1, n_2, ...$ such that for any two nodes n_i, n_{i+1} that are adjacent in the sequence, either $n_i \Rightarrow n_{i+1}$ or $n_i \rightarrow n_{i+1}$. A *proper path* from a node n_1 to a node n_k is a path $n_1, ..., n_k$ such that $n_1 \rightarrow n_2$ and $n_{k-1} \rightarrow n_k$. Finally, a *deconstruction chain*, or just *chain*, is a proper path on which all nodes, except possible the first and last nodes, are deconstructive. We will write $Chain(n, n')$ to mean that there is a chain from n to n'. Next, notice that every node must be either constructive, deconstructive, or regular. We will make use of this fact.

Now imagine tracing an ingredient of a message back along a path. Either that message came from a regular strand, or it was received immediately after being constructed by the adversary (via pairing or encryption), or it was received immediately after being extracted from a larger message by the adversary (via unpairing or decryption). Suppose the message came from a deconstructive adversary node. Then there must be a proper path back to a regular node, with nothing but deconstructive adversary nodes in between. The reason is that while tracing the path backwards, you could never encounter a constructive node, because that would violate the normality condition, so you must encounter only deconstructive nodes until you find a regular node. Thus every reception node must be linked to a constructive or regular node, or it must be the end of a chain that began with a regular node. This conclusion is expressed by the following formula, which we will call the *chain formula*,

```
Recv(r, mr) =>
  Exists n. Link(n, r) & Constructive(n)
| Exists s, ms. Regular(s) & Send(s, ms) & Chain(s, r)
```

Here `Recv`(n, m) means that n is a reception node with $msg(n) = m$. And `Link`(n_1, n_2) means that $n_1 \to n_2$.

To be certain that it is safe to replace the link formula with the chain formula, we will prove that they are logically equivalent. We will make two assumptions: first, we will assume normality, and second, we will assume that every node has a finite number of predecessors. While the latter assumption is true of infiltrated skeletons - which are by definition finite - it might not be true of all models - which may be infinite. In practice, a protocol execution in which a node has an infinite number of predecessors is unrealistic. But by the compactness theorem, we cannot hope to write a theory which excludes these infinite models.

Lemma 7. *If n is a deconstructive reception node with a finite number of inbound predecessors, then there is a regular node n' such that $Chain(n', n)$.*

Proof. We will induct on the number of inbound predecessors of n (an inbound predecessor of n is a reception node n' such that $n' \leq n$). Say that n has p inbound predecessors. Since n is deconstructive, it is part of either an unpairing or a decryption strand, so there is a node n' and message $m_{n'}$ such that $n' \Rightarrow^+ n$ and $Recv(n', m_{n'})$. Since $Recv(n', m_{n'})$, there is a node q such that $Link(q, n')$. Consider the type of q,

1. If q is constructive, then normality is violated.

2. If q is regular, we also know that $Regular(q)$ and $Chain(q, n)$, so we are done.

3. If q is deconstructive, then it has at most $p - 1$ inbound predecessors. By the inductive hypothesis, there is a regular node s such that $Chain(s, q)$. Thus $Regular(s)$ and $Chain(s, n)$ hold, so we are done.

□

Theorem 4. *The link formula and the chain formula are equivalent in models in which every node has a finite number of inbound predecessors.*

Proof. (Forward) Assume that the chain formula holds. Now suppose that $Recv(r, m_r)$. By the chain formula, either

1. there is a constructive node n such that $n \to r$, or

2. there is a chain from s to r.

In the latter case, the existence of a path $s, n_1, ...n_k, r$ implies that $n_k \to r$. Either way, $\exists n.n \to r$. Thus the formula $Recv(n, m) \Rightarrow \exists n'.Link(n', n)$ holds.

(Reverse) Assume that the link formula holds. Now suppose that $Recv(r, m_r)$. We would like to show that one of the disjuncts of the chain formula holds. Then $\exists n.Link(n, r)$ and $Send(n, m_r)$. n must be constructive, deconstructive, or regular.

1. If n is constructive, then $Constructive(n)$ holds and we are done.
2. If n is regular, then $Chain(n, r)$ holds and we are done.
3. Otherwise, n is deconstructive. By the above lemma, there is a regular node s such that $Chain(s, n)$. Since n is deconstructive, we also have $Chain(s, r)$, and we are done.

□

The static theory, including the efficiency and chaining conditions, can be found in appendix C.

3.4 The Functor $\mathcal{F}^P_{rep}$

We will now give evidence toward the correctness theorem. First, we will define a functor $\mathcal{F}_{rep}$ from all possible infiltrated skeletons to all finite models of our static theory, and argue that $\mathcal{F}_{rep}$ is full, faithful, and invertible. We call it $\mathcal{F}_{rep}$ because given any infiltrated skeleton s, it gives the *representation* of s as a model. Next we will explain how to augment $\mathcal{F}_{rep}$ to act on skeletons of any given protocol P, and argue for its correctness in the particular case of Needham-Schroeder.

Let $\mathcal{S}$ be the category whose objects are infiltrated skeletons of all possible protocols and whose morphisms are the homomorphisms between those skeletons. Likewise, let $\mathcal{M}$ be the category whose objects are the finite models of our static theory, and whose morphisms are the homomorphisms between these models. $\mathcal{F}_{rep}$ is a functor from $\mathcal{S}$ to $\mathcal{M}$. This means it maps skeletons to models and skeleton homomorphisms to model homomorphisms. Let us first specify how $\mathcal{F}_{rep}$ acts on skeletons.

Recall that an infiltrated skeleton is a tuple $(N, \rightarrow, \leq, non, uniq)$ obeying the axioms given in section 2.4.3. Let M be the set of all messages, and let $s = (N, \rightarrow, \leq, non, uniq)$ be any infiltrated skeleton in $\mathcal{S}$. Then $\mathcal{F}_{rep}(s)$ is a model with domain $N \cup M$ and the following relations,

$\texttt{Node}(n) \forall n \in N$		
$\texttt{Mesg}(m) \forall m \in M$		
$\texttt{Send}(n, m)$	iff	$dir(n) = +$ and $msg(n) = m$
$\texttt{Recv}(n, m)$	iff	$dir(n) = -$ and $msg(n) = m$
$\texttt{NonOrig}(m)$	iff	$m \in non$
$\texttt{UniqOrig}(m)$	iff	$m \in uniq$
$\texttt{Link}(n_1, n_2)$	iff	$n_1 \rightarrow n_2$
$\texttt{Precedes}(n_1, n_2)$	iff	$n_1 \leq n_2$
$\texttt{Ingredient}(x, y)$	iff	$x \sqsubseteq y$
$\texttt{Pair}(x, y, z)$	iff	$z = (x, y)$
$\texttt{Enc}(x, y, z)$	iff	$z = \{\|x\|\}_y$
$\texttt{Basic}(x)$	iff	x is a basic message
$\texttt{Inverse}(x, y)$	iff	$x = y^{-1}$
$\texttt{Parent}(n_1, n_2)$	iff	$n_1 \Rightarrow n_2$
$\texttt{E1}(n)$	iff	n is the first node of an adversarial encryption strand
$\texttt{E2}(n)$	iff	n is the second node of an adversarial encryption strand
$\texttt{E3}(n)$	iff	n is the third node of an adversarial encryption strand

... *and so on for the other adversarial nodes* where n, n_1, n_2 vary over the elements of N and m, x, y, z vary over the messages of M.

The behavior of $\mathcal{F}_{rep}$ on homomorphisms is easy to describe. Recall that a homomorphism on infiltrated skeletons is a pair (α, ϕ), where α is a homomorphism on messages, ϕ is a mapping from nodes to nodes, and α and ϕ obey the axioms of section 2.4.4. We define $\mathcal{F}_{rep}((\alpha, \phi)) = \alpha \cup \phi$.

3.4.1 Correctness of $\mathcal{F}_{rep}$

We will now argue that $\mathcal{F}_{rep}$ is an invertible full and faithful functor.

Theorem 5. *$\mathcal{F}_{rep}$ is an invertible full and faithful functor from $\mathcal{S}$ to $\mathcal{M}$.*

Proof. There are four properties we must verify. $\mathcal{F}_{rep}$ must be full, faithful, a functor, and well-defined. Faithfullness is easy to verify. It says that if $(\alpha_1, \phi_1) \neq (\alpha_2, \phi_2)$ then $\alpha_1 \cup \phi_1 \neq \alpha_2 \cup \phi_2$. This follows from the disjointness of nodes and messages in the Strand Space formalism.

Assuming that $\mathcal{F}_{rep}$ is well-defined, it is also easy to show that it is a functor. This means that (i) it maps identity homomorphisms onto identity homomorphisms, which it does because $\mathcal{F}_{rep}((id, id)) = id \cup id = id$, and that (ii) it maps the composition of homomorphisms onto the composition of their images, which it does because $\mathcal{F}_{rep}((\alpha_1, \phi_1) \circ (\alpha_2, \phi_2)) = \mathcal{F}_{rep}((\alpha_1 \circ \alpha_2, \phi_1 \circ \phi_2)) = (\alpha_1 \circ \alpha_2) \cup (\phi_1 \circ \phi_2) = (\alpha_1 \cup \phi_1) \circ (\alpha_2 \cup \phi_2) = \mathcal{F}_{rep}((\alpha_1, \phi_1)) \circ \mathcal{F}_{rep}((\alpha_2, \phi_2))$.

In order for $\mathcal{F}_{rep}$ to be well-defined, it must map skeletons to models, and it must map homomorphisms to homomorphisms. To see that $\mathcal{F}_{rep}$ is well-defined on skeletons, we need that for any skeleton $s \in S$, $\mathcal{F}_{rep}(s)$ satisfies the static theory. Likewise, for it to be invertible, we need that for any model $m \in M$, $\mathcal{F}_{rep}^{-1}(m)$ obeys the axioms of 2.4.3.

For example, consider the fourth axiom of 2.4.3, which states that $\leq$ is a subset of the reflexive transitive closure of $\rightarrow \cup \Rightarrow$. Verifying that this axiom is satisfied involves showing that the formulas involving `Precedes` in the static theory guarentee that it includes the transitive closure of `Link` and `Parent`.

We are left with the well-defined property for homomorphisms and the fullness property. The well-defined property for homomorphisms says that if (α, ϕ) is a skeleton homomorphism then $\alpha \cup \phi$ is a model homomorphism, and the fullness property says that if $\alpha \cup \phi$ is a homomorphism then (α, ϕ) is a homomorphism. Proving this is a matter of showing that each of the axioms of 2.4.4 is implied by relations being preserved, and that the preservation of each relation is implied by some axiom of 2.4.4. Additionally, message homomorphisms must be examined in a similar fashion.

As an example, suppose that $\alpha \cup \phi : \mathcal{F}_{rep}(s_1) \to \mathcal{F}_{rep}(s_2)$. Now consider the first axiom of 2.4.4. It states that $n_1 \Rightarrow n_2$ implies $\phi(n_1) \Rightarrow \phi(n_2)$. This axiom is satisfied, because if $n_1 \Rightarrow n_2$ in s_1, then `parent`(n_1, n_2) in $\mathcal{F}_{rep}(s_1)$, so `parent`$(\phi(n_1), \phi(n_2))$ in $\mathcal{F}_{rep}(s_2)$, so $\phi(n_1) \Rightarrow \phi(n_2)$ in s_2.

Verifying the full correctness of $\mathcal{F}_{rep}$ is a matter of examining the rest of the axioms of 2.4.3 and 2.4.4 and the rest of the relations of appendix C, in order to check that,

- S is an infiltrated skeleton iff $\mathcal{F}_{rep}(S)$ is a model of the static theory *(Well-defined on skeletons and invertible).*
- (α, ϕ) is a homomorphism on skeletons iff $\alpha \cup \phi$ is a homomorphisms on models *(Well-defined on homomorphisms and full).*

□

3.4.2 Augmenting $\mathcal{F}_{rep}$

We will describe how to modify $\mathcal{F}_{rep}$ to handle any particular protocol P. We will call the modified functor $\mathcal{F}^P_{rep}$; we would like that $\mathcal{F}^P_{rep}$ be an invertible full and faithful functor from $\mathcal{S}_P$ to $\mathcal{M}_P$. The theory T_P contains two kinds of relations not found in the static theory: (i) relations which say that a node is the nth node of a role in that protocol, and (ii) relations which say that a message takes the place of a particular parameter in a role. Modifying $\mathcal{F}_{rep}$ to handle a given protocol involves only augmenting its behavior on skeletons to respect these relations. This is best understood by example; we will consider the case of the Needham-Schroeder protocol from the perspective of the initializer. We will call this protocol NS.

The dynamic portion of T_{NS} is given in appendix B. It contains 14 relation symbols beyond those found in the static theory. Six of them, such as `init2` declare that a node is the nth node of a regular strand of a given role. `init2`(n), for instance, means that n is the second node of an initiator strand. The other eight relation symbols declare that a message takes the place of a parameter in a role. `n1_resp`(n, m), for example, means that n is a node in an instance of a responder strand, and the instance of the $n1$ parameter for that role is m.

We define $\mathcal{F}_{rep}^{NS}$ as follows,

- For any skeleton s of NS, $\mathcal{F}_{rep}^{NS}(s)$ is the model $\mathcal{F}_{rep}(s)$ with the following extra relations,
  - `a_init`(n, m) holds iff $n \in N$, $m \in M$, and m is the value of the parameter a of the initiator role in the strand of n.
  - `b_resp`(n, m) holds iff $n \in N$, $m \in M$, and m is the value of the parameter b of the responder role in the strand of n.
  - `init1`(n) holds iff $n \in N$ and n is the first node of an initiator strand in s.
  - `init2`(n) holds iff $n \in N$ and n is the second node of an initiator strand in s.

  ... and so on for the other 10 relations.
- For any skeleton homomorphism (α, ϕ) from one skeleton of NS to another, $\mathcal{F}_{rep}^{NS}((\alpha, \phi)) = \mathcal{F}_{rep}((\alpha, \phi)) = \alpha \cup \phi$.

By the same argument as before, proving that $\mathcal{F}^P_{rep}$ is an invertible full and faithful functor involves verifying that,

- S is an infiltrated skeleton of P iff $\mathcal{F}^P_{rep}(S)$ is a model of the T_P.
- (α, ϕ) is a homomorphism on skeletons of P iff $\alpha \cup \phi$ is a homomorphisms on models of T_P.

For a skeleton to be a skeleton of NS, (i) it must satisfy the origination assumptions of the initializer, (ii) it must contain at least one initializer strand, and (iii) all of its regular nodes must be part of either intializer or responder strands. The first two conditions are handled by the formula,

```
True -> Exists _n0, a, b, n1: name(a) & name(b)
    & text(n1) & non-orig(Privk<b>) & non-orig(Privk<a>)
    & uniq-orig(n1) & init3(_n0) & a_init(_n0, a)
    & b_init(_n0, b) & n1_init(_n0, n1)
```

Here the origination assumptions are that A's and B's private keys are nonoriginating, and that the message $n1$ is uniquely-originating. The atom `init3`$(\_n0)$ declares the existence of an initiator strand, and the atoms `b_init`$(\_n0, b)$ and `n1_init`$(\_n0, n1)$ tie the messages b and $n1$ to this strand.

For a strand s to be an instance of an initiator or responder role, there must be a mapping ψ from the parameters of that role to messages under which s is identical to an initial segment of that role, and the role's origination assumptions are satisfied. In this case, the roles of NS have no origination assumptions. The mapping ψ is given by the relations `a_init`, `b_init`, ... This mapping forces the messages of a strand to have the correct form by formulas such as,

```
resp1(_n) & n1_resp(_n, n1) & a_resp(_n, a) & b_resp(_n, b)
    -> recv(_n, Enc<Pair<n1, a>, Pubk<b>>)
```

Finally, the fact that a regular strand must appear to be an *initial* segment of a strand is given by formulas such as,

```
resp2(_n) -> Exists _m: Parent(_n, _m) & resp1(_m)
```

The formulas of the above form together state that whenever a node acts as the $n+1$st node of a role for $n \geq 0$, it has a parent that acts as the nth node of that role. Formulas of the form,

```
b_resp(_n, _x) & Parent(_m, _n) -> b_resp(_m, _x)
```

then ensure that its parent shares the same parameters.

We end with the following theorem, which we believe to be true, and which implies the correctness theorem 3,

Theorem 6. *For any protocol P, $\mathcal{F}^P_{rep}$ witnesses the correctness of P.*

3.5 A Chase Implementation

We were able to use our approach to analyze the Needham-Schroeder protocol from the initializer's (Alice's) perspective, finding a single correct execution as expected. At that point in time, we had not yet developed a protocol compiler, so we expressed the protocol in geometric logic by hand. We used an implementation of the Chase written by Michael Ficarra, a student at WPI, for his Major Qualifying Project[8].

Two major stumbling blocks were the inefficiency and lack of support for function symbols in Ficarra's Chase implementation. This should not be taken as criticism of his work. While the Chase algorithm itself is simple, efficiently implementing it is a difficult task. Implementing parts of the algorithm in the obvious way, such as implementing formula satisfaction testing by enumerating all possible environments, often leads to exponential slowdowns. Likewise, introducing functions is convenient, but conceptually muddles the algorithm.

3.6 Future Work

Future work could include improving the Chase, making the protocol compiler more practical to use, and integrating the compiler and the Chase into a single tool. The Chase implementation could be improved by making it fair (see 2.3.3), more efficient, and by adding support for function symbols.

Ideally, a user would be able to provide a protocol description, and receive a list of possible executions in a human-readable format. Another useful feature would be the ability to verify security properties on a protocol by checking that they holds of all its executions. The only properties that we know can be verified in this way are positive existential; whether there is a usable way to verify a larger class of properties is an open question.

We conclude that the use of the Chase for protocol analysis is a promising approach.

Chapter 4

Homomorphism Problems

Considering a special case of protocol executions led to our discovery of the following algorithm. In practice this class of executions turned out to be too restrictive to be useful, but the algorithm stands on its own as a method of testing isomorphism on single-inbound graphs.

4.1 Single-Inbound Graph Isomorphism Testing

We will present an algorithm to test isomorphism between *single-inbound* graphs. A directed graph is *single-inbound* if each of its vertices has in-degree at most one. Our algorithm is an extension of the rooted tree isomorphism testing algorithm by Aho, Hopcroft, and Ullman[1], which we will call the AHU algorithm. It assumes that graphs are represented in such a way that a vertex's inbound edge can be discovered in constant time.

The algorithm has two stages. In the first stage, the components of the graph are discovered, and in the second they are compared for isomorphism. It can be shown that the components of a single-inbound graph are either trees or unicyclic. This simplifies component discovery; applying a cycle-detection algorithm to the inbound path of an unexplored vertex will yield either a cycle, which must be the center of a unicyclic component, or a vertex with no inbound edge, which must be the root of a tree. This classification of the components also simplifies isomorphism testing. The tree components can be compared using the AHU algorithm, and we present an algorithm to compare the unicyclic components in linear time.

Lemma 8. *No component of a single-inbound graph can have more than one cycle.*

Proof. Suppose a component has two cycles, A and B. Select a vertex x that is in A but not in B, and a vertex y that is in B but not A. Since x and y are in the same component, there must be a path from one to the other. Without loss of generality, we can assume that the path goes from x to y. Now consider the first vertex in the path which lies on B. It must have two inbound edges - one from B and one from the path. Thus, by contradiction, no component can have more than one cycle. $\square$

We can conclude that the components of a single-inbound graph are either rooted trees (zero cycles), or several trees whose roots are connected with a cycle (one cycle).

4.1.1 Phase I - Component Discovery

The purpose of the first stage of the algorithm is to discover the components of the graph. To begin, initialize three sets to be empty: a set of cycles C, a set of tree roots R, and a set of explored vertices V. To retain linear time, an implementation of sets with constant insertion and lookup time must be used. Next, for each vertex v in the graph, if v is not currently in E,

- Add v to E.
- Run a cycle detection algorithm on the inbound path to v.

- If a cycle is detected, add it to C.
- If, on the other hand, the inbound path terminates, add the terminal vertex to R.

4.1.2 Phase II - Isomorphism Testing

Once the components of the graph have been discovered, its tree components and the trees within its unicyclic components may be labeled in linear time by the AHU algorithm. The only remaining challenge is to produce an ordering on the unicyclic components. We devised the following algorithm to accomplish this task.

```
-- Compare cycles of the same length
compareCycles :: (Ord a) => Cycle a -> Cycle a -> Ordering
compareCycles (Cycle x) (Cycle y) =
  let xlist = x ++ x
      ylist = y ++ y in
  loop (length x) xlist xlist ylist ylist
  where
    loop n l [] _ _ =
      if length l >= n then EQ else GT
    loop n _ _ l [] =
      if length l >= n then EQ else LT
    loop n xtail xhead ytail yhead =
      case compare (head xhead) (head yhead) of
        EQ -> loop n xtail (tail xhead) ytail (tail yhead)
        LT -> loop n xtail xtail (tail yhead) (tail yhead)
        GT -> loop n (tail xhead) (tail xhead) ytail ytail
```

Our algorithm takes as input two cycles of ordered elements, and compares them cycle-lexicographically. A cycle of length n can be converted into a list in n different ways, by beginning the list with any element of the cycle. For example, the cycle $(1, 2, 1, 1, 2)$ can be converted to $[1, 2, 1, 1, 2]$, $[2, 1, 1, 2, 1]$, $[1, 1, 2, 1, 2]$, $[1, 2, 1, 2, 1]$, or $[2, 1, 2, 1, 1]$. One such cycle will be lexicographically least; in this case $[1, 1, 2, 1, 2]$. The *cycle-lexicographic* ordering orders cycles by comparing their lexicographically least list representations. Our algorithm performs this comparison in linear time.

4.1.3 Single-Inbound Edge-Colored Digraphs

One might wonder whether this approach can be easily extended to larger classes of graphs. One possibility was the set of edge-colored digraphs in which each vertex has at most one inbound edge of each color. We found, however that this is not the case - even in the case of two colors, this problem is as hard as general digraph isomorphism.

Let U_k be the set of edge-colored digraphs with the following two properties,

1. No vertex has two incoming edges of the same color.

2. There are at most k edge colors.

Theorem 7. *The isomorphism problem for digraphs is polynomial-reducible to the isomorphism problem for elements of U_2.*

Proof. We will give a translation $\zeta()$ from digraphs to U_2 such that two digraphs are isomorphic if and only if their images are. The image of a digraph with v vertices and e edges will have e red edges, e green edges, and $v + e$ vertices. Each vertex will be mapped to a vertex, and each edge will be mapped to a new vertex along with two incoming edges: a green one from the image of its source, and a red one from the image of its target.

To be more precise, let D be any digraph. Then,

$$\begin{aligned} V(\zeta(D)) &= V(D) \cup E(D) \\ E_{green}(\zeta(D)) &= (x,(x,y)) : (x,y) \in E(D) \\ E_{red}(\zeta(D)) &= (y,(x,y)) : (x,y) \in E(D) \end{aligned}$$

Now we must show that two digraphs are isomorphic if and only if their images under $\zeta()$ are. It is clear that if two digraphs are isomorphic then their images are. Now suppose that ϕ is an isomorphism between the images of two digraphs D_1 and D_2. Let (x,y) be an edge of D_1. Then

$$\begin{aligned} & x \rightarrow_{green} (x,y) \leftarrow_{red} y \text{ in } \zeta(D_1) \\ \Longrightarrow\ & \phi(x) \rightarrow_{green} z \leftarrow_{red} \phi(y) \text{ in } \zeta(D_2) \text{ (by the isomorphism property)} \\ \Longrightarrow\ & \phi(x) \rightarrow_{green} (\phi(x),\phi(y)) \leftarrow_{red} \phi(y) \text{ in } \zeta(D_2) \\ \Longrightarrow\ & (\phi(x),\phi(y)) \text{ in } D_2 \text{ (by the definition of } \zeta()) \end{aligned}$$

By symmetry, ϕ preserves edges in both directions: from D_1 to D_2 and from D_2 to D_1. Thus the restriction of ϕ onto the vertices of D_1 is an isomorphism between D_1 and D_2. □

4.2 Isomorphism Complete Problems

While the models returned by the Chase are jointly universal, they might not be minimal. There may be a homomorphism, or even an isomorphism, from one Chase model to another. In such a case, we would prefer to eliminate the less general model. Thus the question of the complexity of the homomorphism and isomorphism problems for relational structures naturally arises.

Each execution can be thought of in two equally valid ways. An execution is a relational structure, since it was produced by the Chase running on a geometric theory. But it is also an infiltrated skeleton in a strand space.

Temporarily ignoring its messages and origination assumptions, its remaining structure can be viewed as a directed acyclic graph of nodes. So we may also ask about the complexity of the homomorphism and isomorphism problems for DAGs.

We discovered that the complexities were the same from both perspectives. In fact, for several graph-like structures - graphs, digraphs, DAGs, edge-labeled digraphs, and relational structures - the homomorphism problems are all NP-complete while the isomorphism problems are isomorphism-complete. We will use the word *structure* to refer generically to these graph-like entities from now on.

The best known algorithms for solving NP-complete problems have exponential running time. It is conjectured that they cannot be solved any faster; this conjecture is one of the most important open problems in computer science. Isomorphism-complete problems are defined as being equivalent in complexity to the graph isomorphism problem. These problems have a similar standing; the best known algorithm runs in $O(N^{\log N})$ time[6] (part of the *quasi-polynomial* class), and it is thought, though unproven, that no polynomial-time algorithm for them exists.

4.2.1 Structure Reductions

All of the aforementioned complexity results were known. Our contribution in the area is an elegant and constructive way of demonstrating the equivalence of the homomorphism and isomorphism problems of any two kinds of structures. The general idea is to describe a function r of polynomial time complexity that reduces structures of one kind to structures of another kind in such a way that $A \to B$ iff $r(A) \to r(B)$ and $A \leftrightarrow B$ iff $r(A) \leftrightarrow r(B)$. Homomorphism and isomorphism can then be tested simply by applying the reduction and testing the resulting structures.

$$\begin{tikzcd}
A \arrow[r, dashed, "h"] \arrow[d, "r"] & B \arrow[d, "r"] \\
r(A) \arrow[r, dashed, "h'"] & r(B)
\end{tikzcd}$$

$$A \to B \text{ iff } r(A) \to r(B)$$

If such a reduction exists, it can be concluded that the homomorphism and isomorphism problems on the source structures are no harder than the respective problems on the target structures. We have found a sufficient condition for a reduction function to have this property. We call such functions *structure reductions.* Besides being polynomial-time computable, structure reductions have four properties. Roughly speaking, (1) They replace edges in the graph they are given with more complex substructures, (2) they treat similar edges similarly, (3) they are injective on the vertices of the structure they are given, and (4) there is never any ambiguity about the preimage of a substructure of the resulting structure. For simplicity, we will assume that every vertex in a structure is incident with at least one edge; it should be clear that this will not make the homomorphism and isomorphism problems any easier.

Definition 5 (Structure Reduction)**.** *A* structure reduction *is a function $r :: S_1 \to S_2$ such that for all structures $A, B \in S_1$ and edges $e \in E(A)$,*

1. *r is injective on $V(A)$*
2. *$r(A) = \bigcup_{e \in E(A)} r(e)$*
3. *If $r(A) \xrightarrow{h} r(B)$ then $r^{-1}(h(r(e))) \in E(B)$*
4. *If $A \xrightarrow{h} B$ then there is a homomorphism ϕ_e such that*
    - *$r(e) \xrightarrow{\phi_e} r(h(e))$*

- *For all $v \in V(e)$, $\phi(r(v)) = r(h(v))$*

Theorem 8 (Preservation Theorem)**.** *If a function $r :: S_1 \to S_2$ is a* structure reduction*, then for all $A \in S_1$ and $B \in S_2$,*

- *There is a homomorphism from A to B iff there is a homomorphism from $r(A)$ to $r(B)$*
- *Likewise for isomorphisms*

Proof. (Forward) Suppose that $A \xrightarrow{h} B$. Let $h' = \cup_{e \in E(A)} \phi_e$. h' is well defined for the following reason: For any two edges $e_1, e_2 \in E(A)$, if $v \in V(r(e_1))$ and $v \in V(r(e_2))$ then $v = r(w)$ for some $w \in V(A)$. Thus $\phi_{e_1}(v) = \phi_{e_1}(r(w)) = r(h(w)) = \phi_{e_2}(r(w)) = \phi_{e_2}(v)$, and h' is well-defined. For any edge $e \in E(r(A))$, $e \subseteq r(e')$ for some $e' \in E(A)$. Thus $h'(e) = \phi_{e'}(e) \in E(r(B))$. Since h' preserves edges, $r(A) \xrightarrow{h'} r(B)$.

(Reverse) Suppose that $r(A) \xrightarrow{h} r(B)$. Let $h' :: V(A) \to V(B)$, $h' = r^{-1} \circ h \circ r$. h' is well-defined because r is bijective on $V(B)$. For any edge $e \in E(A)$, $h(r(e)) = r(e')$ for some $e' \in E(B)$. Thus $h'(e) = r^{-1}(h(r(e))) = r^{-1}(r(e')) = e' \in E(B)$. Since h' preserves edges, $A \xrightarrow{h'} B$.

(Isomorphisms) Notice that in the above two cases, if h is bijective, then h' will also be bijective. Thus $A \leftrightarrow B$ iff $r(A) \leftrightarrow r(B)$. □

We conjecture that structure reductions also preserve substructure-isomorphisms.

4.2.2 Examples

Some examples are in order. Each of the following is a structure reduction. Together, they show the equivalence of the homomorphism problems and of the isomorphism problems on graphs, digraphs, DAGs, edge-labeled digraphs, and relational structures.

Graph $\to$ Digraph Replace each undirected edge

$$\bullet_x \text{———} \bullet_y$$

with two directed edges

$$\begin{tikzcd}
\bullet_x \arrow[r, bend left] & \bullet_y \arrow[l, bend left]
\end{tikzcd}$$

Digraph → Graph Replace each edge

$$\begin{tikzcd}
\bullet_x \arrow[r] & \bullet_y
\end{tikzcd}$$

with the following graph

$$\begin{tikzcd}
\bullet \arrow[r, dash] \arrow[d, dash] & \bullet \arrow[dl, dash] & \bullet \arrow[r, dash] \arrow[dr, dash] & \bullet \arrow[d, dash] \\
\bullet_x \arrow[r, dash] & \bullet \arrow[r, dash] & \mathrm{G} \arrow[r, dash] & \bullet_y
\end{tikzcd}$$

where G is the center vertex of a Grotzsch Graph. This reduction relies on the fact the the triangle and the Grotzsch Graph form an antichain; there is no homomorphism from the triangle to G because G is triangle-free, and there is no homomorphism from G to the triangle because G is not three-colorable.

Digraph → DAG Replace each edge

$$\begin{tikzcd}
\bullet_x \arrow[r] & \bullet_y
\end{tikzcd}$$

with

$$\begin{tikzcd}
\bullet_x & \bullet \arrow[l] & \bullet \arrow[l] \arrow[ll, bend right] \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \bullet_y
\end{tikzcd}$$

DAG → Digraph The identity function

Digraph → Edge-labelled Digraph Just label each edge with a '0'.

Edge-labelled Digraph → Digraph 1. Convert the labels into binary sequences all of the same length. For instance, red, green, blue may become $00, 01, 10$.

2. Now translate an edge, say

   $$\bullet_x \xrightarrow{'010'} \bullet_y$$

   into the following structure

   $$\begin{tikzcd}
   \bullet \arrow[d, leftrightarrow] & & \bullet \arrow[d, leftrightarrow] \\
   \bullet_x \arrow[r] & \bullet \arrow[r] \arrow[d] & \bullet_y \\
   & C_3 \arrow[d] & \\
   & C_5 \arrow[d] & \\
   & C_3 \arrow[d] & \\
   & C_7 &
   \end{tikzcd}$$

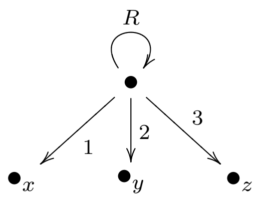
   By C_n we mean a vertex of the digraph cycle of length n. Here C_3 denotes the digit '0', C_5 denotes the digit 1, and C_7 denotes the end of the binary sequence. This reduction relies on the fact that the prime-numbered cycles, $C_2, C_3, C_5, C_7, \ldots$ form an antichain.

Edge-labelled Digraph → Relational Structure

$$\bullet_x \xrightarrow{l} \bullet_y \qquad\qquad l(x, y)$$

Relational Structure $\rightarrow$ Edge-labelled Digraph

$R(x, y, z)$



Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] Bruce Lloyd Bauslaugh. *Homomorphisms on Infinite Directed Graphs*. PhD thesis, Simon Fraser University, December 1994.

[3] Peter J. Cameron. Graph homomorphisms Combinatorics Study Group Notes. In `http://www.maths.qmw.ac.uk/~pjc/csgnotes/hom1.pdf`, November 2006.

[4] Cas J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. PhD thesis, Eindhoven University of Technology, November 2006.

[5] Cas J.F. Cremers. Unbounded verification, falsification, and characterization of security protocols by pattern refinement. In *ACM Computer and Communication Security (CCS-15)*, 2008.

[6] Narsingh Deo, J.M. Davis, and R.E. Lord. A new algorithm for digraph isomorphism. In *BIT International Conference on Numerical Mathematics*, pages 16–30, 1997.

[7] Daniel Dougherty. Some notes on geometric logic. Unpublished notes on geometric logic and the Chase, 2011.

[8] Michael Ficarra. Generating universal models for geometric theories. Major Qualifying Project Report, October 2010.

[9] Joshua D. Guttman. Shapes: Surveying crypto protocol runs. In Veronique Cortier and Steve Kremer, editors, *Formal Models and Techniques for Analyzing Security Protocols*, Cryptology and Information Security Series. IOS Press, 2011.

[10] Joshua D. Guttman, F. Javier, and F. Javier Thayer Fbrega. Authentication tests and the structure of bundles. *Theoretical Computer Science*, 283:2002, 2002.

[11] Joshua D. Guttman and F. Javier Thayer. Authentication tests and the structure of bundles. *Theoretical Computer Science*, 283(2):333–380, June 2002.

[12] Mac Lane and Saunders. *Categories for the Working Mathematician Graduate Texts in Mathematics 5 (2nd ed.).* Springer-Verlag, 1998.

[13] John D. Ramsdell, Joshua D. Guttman, and Paul D. Rowe. *The CPSA Specification: A Reduction System for Searching for Shapes in Cryptographic Protocols.* The MITRE Corporation, 2009. In `http://hackage.haskell.org/package/cpsa` source distribution, `doc` directory.

[14] F. Javier Thayer Fábrega, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Why is a security protocol correct? In *1998 IEEE Symposium on Security and Privacy.* IEEE Computer Society Press, May 1998.

Appendix A

Needham-Schroeder Protocol Definition

```
;;; This protocol definition is from the CPSA distribution

(herald "Needham-Schroeder Public-Key Protocol"
(comment "This protocol contains a man-in-the-middle"
 "attack discovered by Galvin Lowe."))

(defprotocol ns basic
  (defrole init
    (vars (a b name) (n1 n2 text))
    (trace
     (send (enc n1 a (pubk b)))
     (recv (enc n1 n2 (pubk a)))
     (send (enc n2 (pubk b)))))
  (defrole resp
    (vars (b a name) (n2 n1 text))
    (trace
     (recv (enc n1 a (pubk b)))
     (send (enc n1 n2 (pubk a)))
     (recv (enc n2 (pubk b)))))
  (comment "Needham-Schroeder"))

;;; The initiator point-of-view
(defskeleton ns
  (vars (a b name) (n1 text))
  (defstrand init 3 (a a) (b b) (n1 n1))
  (non-orig (privk b) (privk a))
  (uniq-orig n1)
  (comment "Initiator point-of-view"))

;;; The responder point-of-view
(defskeleton ns
  (vars (a name) (n2 text))
  (defstrand resp 3 (a a) (n2 n2))
  (non-orig (privk a))
  (uniq-orig n2)
  (comment "Responder point-of-view"))
```

Appendix B

Needham-Schroeder Compiler Output

The dynamic portion of the theory for the Needham-Schroeder protocol, from the initiator's perspective, as output by our compiler:

```
a_init(_n, _x) -> Node(_n) & name(_x)
a_init(_n, _x) & a_init(_n, _y) -> _x = _y
a_init(_n, _x) & Parent(_m, _n) -> a_init(_m, _x)
b_init(_n, _x) -> Node(_n) & name(_x)
b_init(_n, _x) & b_init(_n, _y) -> _x = _y
b_init(_n, _x) & Parent(_m, _n) -> b_init(_m, _x)
n1_init(_n, _x) -> Node(_n) & text(_x)
n1_init(_n, _x) & n1_init(_n, _y) -> _x = _y
n1_init(_n, _x) & Parent(_m, _n) -> n1_init(_m, _x)
n2_init(_n, _x) -> Node(_n) & text(_x)
n2_init(_n, _x) & n2_init(_n, _y) -> _x = _y
n2_init(_n, _x) & Parent(_m, _n) -> n2_init(_m, _x)
init1(_n) & n1_init(_n, n1) & a_init(_n, a)
    & b_init(_n, b) -> send(_n, Enc<Pair<n1, a>, Pubk<b>>)
init1(_n) -> Exists n1, a, b: n1_init(_n, n1)
    & a_init(_n, a) & b_init(_n, b)
```

```
init2(_n) & n1_init(_n, n1) & n2_init(_n, n2)
    & a_init(_n, a) -> recv(_n, Enc<Pair<n1, n2>, Pubk<a>>)
init2(_n) -> Exists _m: Parent(_n, _m) & init1(_m)
init2(_n) -> Exists n1, n2, a: n1_init(_n, n1)
    & n2_init(_n, n2) & a_init(_n, a)
init3(_n) & n2_init(_n, n2) & b_init(_n, b)
    -> send(_n, Enc<n2, Pubk<b>>)
init3(_n) -> Exists _m: Parent(_n, _m) & init2(_m)
init3(_n) -> Exists n2, b: n2_init(_n, n2) & b_init(_n, b)
a_resp(_n, _x) -> Node(_n) & name(_x)
a_resp(_n, _x) & a_resp(_n, _y) -> _x = _y
a_resp(_n, _x) & Parent(_m, _n) -> a_resp(_m, _x)
b_resp(_n, _x) -> Node(_n) & name(_x)
b_resp(_n, _x) & b_resp(_n, _y) -> _x = _y
b_resp(_n, _x) & Parent(_m, _n) -> b_resp(_m, _x)
n1_resp(_n, _x) -> Node(_n) & text(_x)
n1_resp(_n, _x) & n1_resp(_n, _y) -> _x = _y
n1_resp(_n, _x) & Parent(_m, _n) -> n1_resp(_m, _x)
n2_resp(_n, _x) -> Node(_n) & text(_x)
n2_resp(_n, _x) & n2_resp(_n, _y) -> _x = _y
n2_resp(_n, _x) & Parent(_m, _n) -> n2_resp(_m, _x)
resp1(_n) & n1_resp(_n, n1) & a_resp(_n, a)
    & b_resp(_n, b) -> recv(_n, Enc<Pair<n1, a>, Pubk<b>>)
resp1(_n) -> Exists n1, a, b: n1_resp(_n, n1)
    & a_resp(_n, a) & b_resp(_n, b)
resp2(_n) & n1_resp(_n, n1) & n2_resp(_n, n2)
    & a_resp(_n, a) -> send(_n, Enc<Pair<n1, n2>, Pubk<a>>)
resp2(_n) -> Exists _m: Parent(_n, _m) & resp1(_m)
resp2(_n) -> Exists n1, n2, a: n1_resp(_n, n1)
    & n2_resp(_n, n2) & a_resp(_n, a)
resp3(_n) & n2_resp(_n, n2) & b_resp(_n, b)
    -> recv(_n, Enc<n2, Pubk<b>>)
```

```
resp3(_n) -> Exists _m: Parent(_n, _m) & resp2(_m)
resp3(_n) -> Exists n2, b: n2_resp(_n, n2) & b_resp(_n, b)
True -> Exists _n0, a, b, n1: name(a) & name(b)
    & text(n1) & non-orig(Privk<b>) & non-orig(Privk<a>)
    & uniq-orig(n1) & init3(_n0) & a_init(_n0, a)
    & b_init(_n0, b) & n1_init(_n0, n1)
```

Appendix C

The Strand Space Theory

```
# Infiltrated Skeletons
# Justin Pombrio
# Joshua Guttman
# Daniel Dougherty


##############
### AXIOMS ###
##############

# Every Term specified in the protocol must *explicitly* be
#   a Concatenation, Encryption, or Basic Term;
# Term Origination is not inferred, but should be written
#   with the protocol rules;
# Is 'Precedes' necessary in the AdvDeConstructed rule?
#   It should be forced to be true by the Chain relation;

# Types #

Mesg(x) | Node(x);
Mesg(x) & Node(x) ->;

# Send, Recv #

Send(n, t) -> Node(n) & Mesg(t);
Recv(n, t) -> Node(n) & Mesg(t);

Node(n) -> Exists t: Send(n, t)
        |  Exists t: Recv(n, t);

Send(n, s) & Send(n, t) -> s = t;
Recv(n, s) & Recv(n, t) -> s = t;

Send(n, s) & Recv(n, t) -> false;

# Inverse #

Inverse(k, k’) -> Mesg(k) & Mesg(k’) & Inverse(k’, k);
Inverse(k, k1) & Inverse(k, k2) -> k1 = k2;

Inverse(k, k’) & Basic(k) -> Basic(k’);

# Basic #

Basic(t) -> Mesg(t);

# Pair #

Pair(g, h, t) -> Mesg(g) & Mesg(h) & Mesg(t);
Pair(g, h, s) & Pair(g, h, t) -> s = t;
Pair(G1, h1, t) & Pair(g2, h2, t) -> G1 = g2 & h1 = h2;
```

```
# Enc #

Enc(g, k, t) -> Mesg(g) & Mesg(k) & Mesg(t);
Enc(g, k, s) & Enc(g, k, t) -> s = t;
Enc(G1, k1, t) & Enc(g2, k2, t) -> G1 = g2 & k1 = k2;

# Disjointness of Types #

Basic(t) & Enc(g, k, t) ->;
Basic(t) & Pair(g, h, t) ->;
Enc(g, k, t) & Pair(u, v, t) ->;

# Link (->) #

Link(n, m) -> Exists t : Send(n, t) & Recv(m, t);
Link(n, m) & Link(p, m) -> n = p; # Broadcast semantics;

# Parent (=>) #

Parent(n, m) -> Node(n) & Node(m);
Parent(n, m) & Parent(n, p) -> m = p;
Parent(n, m) & Parent(p, m) -> n = p;

# Uniquely Originating #

UniqOrig(t, n) -> Basic(t) & Node(n);

# Non-Originating #

NonOrig(t) -> Basic(t);
```

```
###################
### DEFINITIONS ###
###################

# Precedes #

Parent(n, m) -> Precedes(n, m);
Link(n, m, t) -> Precedes(n, m);

Precedes(n, m) & Precedes(m, p) -> Precedes(n, p);

# Ingredient #

Mesg(t) -> Ingredient(t, t);
Pair(g, h, s) & Ingredient(s, t) ->
Ingredient(g, t) & Ingredient(h, t);
Enc(g, k, s) & Ingredient(s, t) -> Ingredient(g, t);


#####################
### Pruning Rules ###
#####################

# Non-Origination #

NonOrig(t) & Orig(t, n) ->;

# Shortcut for NonOrigination #

NonOrig(t) & Ingredient(t, s) & Send(n, s) ->;
NonOrig(t) & Ingredient(t, s) & Recv(n, s) ->;
```

```
# Cyclicity #

Precedes(n, n) ->;


##################
### ADVESARIES ###
##################

# ThreeStrand(n1, n2, n3) states that nodes n1, n2, and n3 form a
# strand with precisely three nodes, in that order.

ThreeStrand(n1, n2, n3) & Parent(x, n1) ->;
ThreeStrand(n1, n2, n3) & Parent(n3, x) ->;
ThreeStrand(n1, n2, n3) ->
Parent(n1, n2) & Parent(n2, n3) &
Advesary(n1) & Advesary(n2) & Advesary(n3);

# Generation #

G1(n, t) & Parent(x, n) ->;
G1(n, t) & Parent(n, x) ->;

G1(n, t) -> Send(n, t) & Basic(t) & Orig(t, n) & Advesary(n);

# Pairenation #

C1(n1, g, h) -> Exists n2, n3 :
C2(n2, g, h) & C3(n3, g, h) & ThreeStrand(n1, n2, n3);
C2(n2, g, h) -> Exists n1, n3 :
C1(n1, g, h) & C3(n3, g, h) & ThreeStrand(n1, n2, n3);
C3(n3, g, h) -> Exists n1, n2 :
```

```
C1(n1, g, h) & C2(n2, g, h) & ThreeStrand(n1, n2, n3);

C1(n1, g, h) -> Recv(n1, g);
C2(n2, g, h) -> Recv(n2, h);
C3(n3, g, h) -> Exists t :
Pair(g, h, t) & Send(n3, t) & Orig(t, n3);

# Encryption #

E1(n1, g, k) -> Exists n2, n3:
E2(n2, g, k) & E3(n3, g, k) & ThreeStrand(n1, n2, n3);
E2(n2, g, k) -> Exists n1, n3:
E1(n1, g, k) & E3(n3, g, k) & ThreeStrand(n1, n2, n3);
E3(n3, g, k) -> Exists n1, n2:
E1(n1, g, k) & E2(n2, g, k) & ThreeStrand(n1, n2, n3);

E1(n1, g, k) -> Recv(n1, g);
E2(n2, g, k) -> Recv(n2, k);
E3(n3, g, k) -> Exists t:
Enc(g, k, t) & Send(n3, t) & Orig(t, n3);

# Unpairing #

U1(n1, g, h) -> Exists n2, n3:
U2(n2, g, h) & U3(n3, g, h) & ThreeStrand(n1, n2, n3);
U2(n2, g, h) -> Exists n1, n3:
U1(n1, g, h) & U3(n3, g, h) & ThreeStrand(n1, n2, n3);
U3(n3, g, h) -> Exists n1, n2:
U1(n1, g, h) & U2(n2, g, h) & ThreeStrand(n1, n2, n3);

U1(n1, g, h) -> Exists t:
Pair(g, h, t) & Recv(n1, t);
```

```
U2(n2, g, h) -> Send(n2, g);
U3(n3, g, h) -> Send(n3, h);

# Decryption #

D1(n1, g, k) -> Exists n2, n3:
D2(n2, g, k) & D3(n3, g, k) & ThreeStrand(n1, n2, n3);
D2(n2, g, k) -> Exists n1, n3:
D1(n1, g, k) & D3(n3, g, k) & ThreeStrand(n1, n2, n3);
D3(n3, g, k) -> Exists n1, n2:
D1(n1, g, k) & D2(n2, g, k) & ThreeStrand(n1, n2, n3);

D1(n1, g, k) -> Exists t:
Enc(g, k, t) & Recv(n1, t);
D2(n2, g, k) -> Exists k':
Inv(k, k') & Recv(n2, k');
D3(n3, g, k) -> Send(n3, g);


##############
### CHAINS ###
##############

# Chain rule must come first!

# There is a Chain from m->s to r<-t;
Chain(m, s, r, t) ->
  Link(m, r)
| AdvUnpairLeft(m, s, r, t)
| AdvUnpairRight(m, s, r, t)
| AdvDecrypt(m, s, r, t);
```

```
AdvUnpairLeft(m, s, r, t) -> Exists x, y, n1, n2, n3:
Pair(x, y, s) & U1(n1, x, y) & U2(n2, x, y) & U3(n3, x, y) &
Link(m, n1) & Chain(n2, x, r, t);
AdvUnpairRight(m, s, r, t) -> Exists x, y, n1, n2, n3:
Pair(x, y, s) & U1(n1, x, y) & U2(n2, x, y) & U3(n3, x, y) &
Link(m, n1) & Chain(n3, y, r, t);
AdvDecrypt(m, s, r, t) -> Exists g, k, n1, n2, n3:
Enc(g, k, s) & D1(n1, g, k) & D2(n2, g, k) & D3(n3, g, k) &
Link(m, n1) & Chain(n3, g, r, t);

Recv(n, t) -> Constructed(t, n);

# An Adversary Constructed Term 't' and sent it to Node 'r';
Constructed(t, r) ->
  AdvGenerated(t, r)
| AdvConcatenated(t, r)
| AdvEncrypted(t, r)
| AdvDeConstructed(t, r);

AdvGenerated(t, r) -> Exists n:
G1(n, t) & Link(n, r);
AdvConcatenated(t, r) -> Exists g, h, n:
Pair(g, h, t) & C3(n, g, h) & Link(n, r);
AdvEncrypted(t, r) -> Exists g, k, n:
Enc(g, k, t) & E3(n, g, k) & Link(n, r);
AdvDeConstructed(t, r) -> Exists m, s:
Regular(m) & Send(m, s) & Precedes(m, r) &
Chain(m, s, r, t);
```

Appendix D

The Protocol Compiler

```
{-
Copyright (c) 2011, Justin Pombrio All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:

Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.

Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
```

```
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
-}

import Data.Char
import Data.List
import Control.Monad
import Control.Monad.State
import System.IO
import qualified Data.Set as Set
import qualified Data.Map as Map

import SExpr
import Debug.Trace

{-
Usage:
  Take a CPSA-style protocol definition, and compile it into a
  geometric logic theory.
  > cat prot.scm | ./compiler > prot.glt

Overview:
  The translation pipeline is as follows:

  String ----> SExpr ----> Protocol ----> Theory ----> String
         read         compile         translate    show, format

  If you want to change the output formatting (to make use of, say,
  another implementation of the chase), see the Formatting code. The
  chase implementation should be able to handle variable names with
  whatever characters are used in variables in the protocol
  definition.

Limitations:
  Needs a formula "regular(n) => init_1(n) | ...".
  Interspersed comments cause parse error.
  Syntax errors will be shown as incomplete patterns.
  Protocol algorithm ignored.
  Do not support labels.
  Do not support node pairs.

Terminology:

  init_0(n) & init_x(n, x) -> send(n, pair(x, x))

  'init'          principal (princ)
  '0'             index
  'n'             node
  'x'             parameter (param)
  'pair(x, x)'    mesg
  'init_0'        node relation
  'init_0(n)'     node atom
  'init_x'        parameter relation
  'init_x(n, x)'  parameter atom
  'send'          event relation
  'send(n, x)'    send atom
-}


{- Protocol -}

type Var = String

type Sort = String

data Mesg = Simple Var
          | Compound String [Mesg]

type Direction = String

data Event = Event Direction Mesg

type Trace = [Event]
type Predicate = String

type HTMesg = (Int, Mesg)

data Assumption = Assumption Predicate HTMesg

data SkelAssum = SkelAssum Predicate Mesg

data Precedes = Precedes (Int, Int) (Int, Int)

data Role = Role String (Map.Map Var Sort) Trace [Assumption]

type Node = (Princ, Int)

type Maplet = (Var, Mesg)

data Strand = Strand Node [Maplet]
            | Listener Mesg

data Form = Protocol String [Role]
          | Skeleton (Map.Map Var Sort) [Strand] [SkelAssum]
            [Precedes]


{- Main -}

            readSExprs :: Handle -> IO [SExpr Pos]
            readSExprs handle = do
  pos <- posHandle "" handle
  loop pos
  where
    loop pos = do
      sexpr <- load pos
      case sexpr of
        Nothing -> return []
        Just expr -> do
```

```
          exprs <- loop pos
          return (expr : exprs)

testShowTheory = putStrLn $ show $ Theory "theory"
                 [Rule
                  [Relation "R" [term_x]]
                  (Exists ["y", "z"]
                   [Relation "P"
                     [term_x,
                      Function "f" [term_x, Variable "y"]],
                    Equation
                    (Function "g" [term_x])
                    (Function "g" [Variable "z"])])]


main = do
  exprs <- readSExprs stdin
  putStrLn $ show $ translate "" $ compile exprs


{- Compilation -}

compile :: [SExpr a] -> [Form]
compile [] = [] -- 'sequence' ought just ignore Nothings...
compile (form : forms) = case compileForm form of
  Nothing -> compile forms
  Just result -> result : compile forms

compileForm :: SExpr a -> Maybe Form
compileForm expr@(L _ (S _ head : body)) = case head of
  "defprotocol" -> Just $ compileProtocol expr
  "defskeleton" -> Just $ compileSkeleton expr
  "herald" -> Nothing
  "comment" -> Nothing

compileProtocol :: SExpr a -> Form
compileProtocol (L _ (S _
  "defprotocol" : S _ id : _ : rest)) =
  let (roles, alist) = partition isRole rest in
  Protocol id (map compileRole roles)
    where
      isRole (L _ (S _ "defrole" : _)) = True
      isRole _ = False

compileRole :: SExpr a -> Role
compileRole (L _ (S _
  "defrole" : S _ id : vars : trace : assumptions)) =
  Role id (compileVars vars) (compileTrace trace)
  (concatMap compileAssumption assumptions)

compileVars :: SExpr a -> Map.Map Var Sort
compileVars (L _ (S _ "vars" : vars)) =
  Map.unions $ map compileDecl vars

compileDecl :: SExpr a -> Map.Map Var Sort
compileDecl (L _ decl) =
  let sort = case last decl of S _ s -> s
      vars = map compileVar (init decl) in
  Map.fromList (zip vars (repeat sort))

compileVar :: SExpr a -> Var
compileVar (S _ var) = var

compileTrace :: SExpr a -> Trace
compileTrace (L _ (S _ "trace" : events)) =
  map compileEvent events

compileEvent :: SExpr a -> Event
compileEvent (L _ [S _ direction, mesg]) =
  Event direction (compileMesg mesg)

compileAssumption :: SExpr a -> [Assumption]
compileAssumption (L _ (S _ "comment" : _)) = []
compileAssumption (L _ (S _ pred : mesgs)) =
  map (Assumption pred . compileHTMesg) mesgs

compileHTMesg :: SExpr a -> HTMesg
compileHTMesg (L _ [N _ index, mesg]) =
  (index, compileMesg mesg)
compileHTMesg expr = (1, compileMesg expr)

compileMesg :: SExpr a -> Mesg
compileMesg (S _ id) = Simple id
compileMesg (L _ (S _ head : body)) =
  Compound head (map compileMesg body)
compileMesg (N _ x) = error $ show x
compileMesg (Q _ x) = error x

compileSkeleton :: SExpr a -> Form
compileSkeleton (L _ (S _
  "defskeleton" : S _ id : vars : rest)) =
  let (strands, alist) = partition isStrand rest
      (precs, assums) = partition isPrec alist in
  Skeleton (compileVars vars) (map compileStrand strands)
  (concatMap compileSkelAssum assums)
  (concatMap compilePrec precs)
  where
    isStrand (L _ (S _ "defstrand" : _)) = True
    isStrand _ = False
    isPrec (L _ (S _ "precedes" : _)) = True
    isPrec _ = False

compileSkelAssum :: SExpr a -> [SkelAssum]
compileSkelAssum (L _ (S _ "comment" : _)) = []
compileSkelAssum (L _ (S _ pred : mesgs)) =
  map (\(p, m) -> SkelAssum p m) $
  zip (repeat pred) (map compileMesg mesgs)

compilePrec :: SExpr a -> [Precedes]
compilePrec (L _ (S _ "precedes" : precs)) =
  map compileNodePair precs
```

```
compileNodePair :: SExpr a -> Precedes
compileNodePair (L _ [L _ [N _ x1, N _ x2],
                      L _ [N _ y1, N _ y2]]) =
  Precedes (x1, x2) (y1, y2)

compileStrand :: SExpr a -> Strand
compileStrand (L _ [S _ "deflistener", mesg]) =
  Listener $ compileMesg mesg
compileStrand (L _ (S _
  "defstrand" : S _ id : N _ size : maplets)) =
  Strand (id, size) (map compileMaplet maplets)

compileMaplet :: SExpr a -> Maplet
compileMaplet (L _ [left, right]) =
  (compileVar left, compileMesg right)


{- Theory -}

type Rel = String -- relation symbol
type Fun = String
type Princ = String
type Param = String

data Term = Variable Var
          | Function Fun [Term]

data Atom = Relation Rel [Term]
          | Equation Term Term
atom rel vars = Relation rel (map Variable vars)

type Conjunction = [Atom]

data Existential = Exists [Var] Conjunction

-- Assuming that disjunctions aren't necessary.
```

```
data Rule = Rule Conjunction Existential

data Theory = Theory String [Rule]


freeVars :: Term -> [Var]
freeVars (Variable var) = [var]
freeVars (Function _ subterms) =
  nub $ concatMap freeVars subterms


{- Translation -}

translate :: String -> [Form] -> Theory
translate name forms =
  Theory name $ concatMap translateForm forms

translateForm :: Form -> [Rule]
translateForm protocol@(Protocol _ _) =
  translateProtocol protocol
translateForm skeleton@(Skeleton _ _ _ _) =
  [translateSkeleton skeleton]


{- Protocol Translation -}

translateProtocol :: Form -> [Rule]
translateProtocol (Protocol names roles) =
  concatMap translateRole roles

translateRole :: Role -> [Rule]
translateRole (Role name vars trace assumptions) =
  let varRules = Map.elems $
                 Map.mapWithKey (translateVar name) vars
      traceRules = zipWith (translateEvent name)
                           trace [1..]
      assumRules = map (translateAssumption name)
```

```
                        assumptions in
  concat (varRules ++ traceRules ++ assumRules)

translateVar :: String -> Var -> Sort -> [Rule]
translateVar princ var sort =
  [paramSortRule princ var sort,
   paramFuncRule princ var,
   paramParentRule princ var]

translateEvent :: String -> Event -> Int -> [Rule]
translateEvent princ event index =
  let rule1 = eventRule (princ, index) event
      rule2 = eventParentRule (princ, index)
      rule3 = eventParamRule (princ, index) event in
  case rule2 of
    Nothing -> [rule1, rule3]
    Just rule -> [rule1, rule, rule3]

translateAssumption :: String -> Assumption -> [Rule]
translateAssumption princ (Assumption pred (index, mesg)) =
  [assumptionRule (princ, index) pred
                  (translateMesg id mesg)]

translateMesg :: (Var -> String) -> Mesg -> Term
translateMesg namer (Simple var) =
  Variable (namer var)
translateMesg namer (Compound "pubk" [Simple var]) =
  Function "Pubk" [Variable $ namer var]
translateMesg namer (Compound "privk" [Simple var]) =
  Function "Privk" [Variable $ namer var]
translateMesg namer (Compound "invk" [Simple var]) =
  Function "Invk" [Variable $ namer var]
translateMesg namer
  (Compound "ltk" [Simple var1, Simple var2]) =
  Function "Ltk" [Variable $ namer var1,
                  Variable $ namer var2]
translateMesg namer (Compound "cat" [x, y]) =
  Function "Pair" (map (translateMesg namer) [x, y])
-- Ensure that Cats and Encs have exactly two subterms.
translateMesg namer (Compound "cat" (x : y : ys)) =
  Function "Pair"
  [translateMesg namer x,
   translateMesg namer (Compound "cat" (y : ys))]
translateMesg namer (Compound "enc" [plaintext, key]) =
  Function "Enc" (map (translateMesg namer)
                      [plaintext, key])
translateMesg namer (Compound "enc" subterms)
  | length subterms >= 3 =
  Function "Enc" [translateMesg namer
                  (Compound "cat" (init subterms)),
                  translateMesg namer (last subterms)]


{- Rules -}

-- "init1(n) & b_init(n, b) -> non-orig(Privk<b>)"
assumptionRule :: Node -> Predicate -> Term -> Rule
assumptionRule (princ, index) pred term =
  let vars = freeVars term in
  Rule (nodeAtom (princ, index) term_n :
        paramConj princ term_n
          (zip vars (map Variable vars)))
  (Exists [] [assumAtom pred term])

-- "init1(n) & a_init(n, a) & n1_init(n, n1)
-- -> send(n, Enc<n1, Pubk<a>>)"
eventRule :: Node -> Event -> Rule
eventRule (princ, index) (Event dir mesg) =
  let term = translateMesg id mesg
      params = freeVars term in
  Rule (nodeAtom (princ, index) term_n :
        paramConj princ term_n
          (zip params (map Variable params)))
  (Exists [] [eventAtom dir term_n term])
```

```
-- "init2(n) -> Exists m: Parent(m, n) & init1(m)"
eventParentRule :: Node -> Maybe Rule
eventParentRule (princ, 1) = Nothing
eventParentRule (princ, index) = Just $
  Rule [nodeAtom (princ, index) term_n]
  (Exists [var_m] [parentAtom term_m term_n,
                   nodeAtom (princ, index - 1) term_m])

-- "init1(n) -> Exists a, n1: a_init(n, a) & n1_init(n, n1)"
eventParamRule :: Node -> Event -> Rule
eventParamRule (princ, index) (Event pred mesg) =
  let vars = freeVars (translateMesg id mesg) in
  Rule [nodeAtom (princ, index) term_n]
  (Exists vars (paramConj princ term_n
                (zip vars (map Variable vars))))

-- "a_init(n, x) -> Node(n) & name(x)"
paramSortRule :: Princ -> Param -> Sort -> Rule
paramSortRule princ param sort =
  Rule [paramAtom princ param [term_n, term_x]]
  (Exists [] [sortAtom "Node" term_n,
              sortAtom sort term_x])

-- "a_init(n, x) & a_init(n, y) -> x = y"
paramFuncRule :: Princ -> Param -> Rule
paramFuncRule princ param =
  Rule [paramAtom princ param [term_n, term_x],
        paramAtom princ param [term_n, term_y]]
  (Exists [] [Equation term_x term_y])

-- "a_init(n, x) & Parent(n, m) -> a_init(m, x)"
paramParentRule :: Princ -> Var -> Rule
paramParentRule princ var =
  Rule [paramAtom princ var [term_n, term_x],
        parentAtom term_n term_m]
  (Exists [] [paramAtom princ var [term_m, term_x]])
```

```
translateSkeleton :: Form -> Rule
translateSkeleton (Skeleton vars strands assums precs) =
  let nodeVars = take (length strands) var_ns
      precVars = take (length precs) var_ms
      princs = map (\(Strand (princ, _) _) -> princ) strands in
  Rule []
  (Exists (nodeVars ++ Map.keys vars ++
          map fst precVars ++ map snd precVars)
   (sortConj (Map.toList vars) ++
    assumConj assums ++
    concat (zipWith translateStrand strands nodeVars) ++
    concat (zipWith (precConj princs) precs precVars)))

translateStrand :: Strand -> Var -> Conjunction
translateStrand (Strand (princ, index) maplets) var =
  let params = map (\(v, m) -> (v, translateMesg id m)) maplets in
  nodeAtom (princ, index) (Variable var) :
  paramConj princ (Variable var) params


{- Conjunctions -}

-- "b_init(n, b) & n2_init(n, n2) & ..."
paramConj :: Princ -> Term -> [(Param, Term)] -> Conjunction
paramConj princ node params =
  map (\(name, term) -> paramAtom princ name [node, term]) params

-- "Name(a) & Text(x)"
sortConj :: [(Var, Sort)] -> Conjunction
sortConj = map (\(v, s) -> sortAtom s (Variable v))

-- "Nonorig(Privk<b>) & Orig(x)
assumConj :: [SkelAssum] -> Conjunction
assumConj =
  map (\(SkelAssum pred mesg)
      -> assumAtom pred (translateMesg id mesg))

-- resp2(m0) & init3(m1) & lprec(m0, n2)
-- & lprec(m1, n3) & prec(m0, m1)"
precConj :: [Princ] -> Precedes -> (Var, Var) -> Conjunction
precConj princs (Precedes (s, n) (s', n')) (v, v') =
  let t = Variable v
      t' = Variable v' in
  [nodeAtom (princs !! (s - 1), n) t,
   nodeAtom (princs !! (s' - 1), n') t',
   lprecAtom t (Variable $ var_ns !! (s - 1)),
   lprecAtom t' (Variable $ var_ns !! (s' - 1)),
   precAtom t t']


{- *Atoms* -}

-- "prec(m, n)"
precAtom :: Term -> Term -> Atom
precAtom n1 n2 = Relation "precedes" [n1, n2]

-- "lprec(m, n)"
lprecAtom :: Term -> Term -> Atom
lprecAtom n1 n2 = Relation "lprec" [n1, n2]

-- "n1_init(n, x)"
paramAtom :: Princ -> Param -> [Term] -> Atom
paramAtom princ param args =
  Relation (param ++ "_" ++ princ) args

-- "init1(n)"
nodeAtom :: Node -> Term -> Atom
nodeAtom (princ, index) node =
  Relation (princ ++ show index) [node]

-- "Node(n)"
sortAtom :: Sort -> Term -> Atom
sortAtom sort param = Relation sort [param]

-- "Send(n, Enc<n1, Pubk<b>>)"
eventAtom :: Direction -> Term -> Term -> Atom
eventAtom dir node mesg = Relation dir [node, mesg]

-- "Nonorig(Privk<b>)"
assumAtom :: Predicate -> Term -> Atom
assumAtom pred term = Relation pred [term]

-- "Parent(n, m)"
parentAtom :: Term -> Term -> Atom
parentAtom parent child = Relation "Parent" [child, parent]

-- "Listen(Enc<x, y>)"
listenAtom :: Term -> Atom
listenAtom term = Relation "Listen" [term]


{- *Naming* -}

var_n = "_n"
var_m = "_m"
var_x = "_x"
var_y = "_y"
term_n = Variable var_n
term_m = Variable var_m
term_x = Variable var_x
term_y = Variable var_y
var_ns = map (\x -> "_n" ++ show x) [0..]
var_ms = map (\x -> ("_m" ++ show x, "_m"
                           ++ show (x + 1))) [0, 2..]

compileSort :: SExpr a -> Sort
compileSort (S _ sort) = case sort of
  "text" -> "Text"
  "data" -> "Data"
  "name" -> "Name"
  "skey" -> "SKey"
  "akey" -> "AKey"
  "mesg" -> "Basic"


{- *Transcription* -}

instance Show Theory where
  showsPrec _ (Theory name rules) =
    formatTheory $ map (showsPrec 0) rules

instance Show Rule where
  showsPrec _ (Rule [] rhs) =
    formatRule (showString "True") (showsPrec 0 rhs)
  showsPrec _ (Rule lhs rhs) =
    formatRule (formatConj (map (showsPrec 0) lhs))
    (showsPrec 0 rhs)

instance Show Existential where
  showsPrec _ (Exists vars conj) =
    formatExistential
    (map showString vars)
    (formatConj (map (showsPrec 0) conj))

instance Show Atom where
  showsPrec _ (Relation rel terms) =
    formatAtom (showString rel) (map (showsPrec 0) terms)
  showsPrec _ (Equation left right) =
    formatEquation (showsPrec 0 left) (showsPrec 0 right)

instance Show Term where
  showsPrec _ (Variable var) = showString var
  showsPrec _ (Function fun subterms) =
    formatTerm (showString fun) (map (showsPrec 0) subterms)
```

```
{- *Formatting* -}

-- from Haskell wiki
compose :: [a -> a] -> a -> a
compose fs v = foldl (.) id fs $ v

joinShows :: String -> [ShowS] -> ShowS
joinShows delim = compose . intersperse (showString delim)

formatTheory :: [ShowS] -> ShowS
formatTheory rules = joinShows "" rules

formatRule lhs rhs =
  lhs . showString " -> " . rhs . showString "\n"

formatExistential [] body = body
formatExistential vars body =
  showString "Exists " .
  joinShows ", " vars . showString ": " . body

formatConj conjuncts = joinShows " & " conjuncts

formatEquation lhs rhs = lhs . showString " = " . rhs

formatAtom rel vars =
  rel . showString "(" . joinShows ", " vars . showString ")"

formatTerm fun subterms =
  fun . showString "<" . joinShows ", " subterms . showString ">"
```