Modelling and Proving Cryptographic Protocols in the Spi Calculus using Coq

Adam Tonet

Submitted in partial fulfilment
of the requirements for the degree of

Master of Science

Department of Computer Science
Faculty of Mathematics and Science
Brock University
St. Catharines, Ontario

Abstract

The spi calculus is a process algebra used to model cryptographic protocols. A process calculus is a means of modelling a system of concurrently interacting agents, and provides tools for the description of communications and synchronizations between those agents. The spi calculus is based on Robin Milners pi calculus, which was itself based upon his Calculus of Communicating Systems (CCS). It was created by Martin Abadi and Andrew D. Gordon as an expansion of the pi calculus intended to focus on cryptographic protocols, and adds features such as the encryption and decryption of messages using keys. The Coq proof system is an interactive theorem prover that allows for the definition of types and functions, and provides means by which to prove properties about them. The spi calculus has been implemented in Coq and subsequently used to model and show an example proof of a property about a simple cryptographic protocol. This required the implementation of both the syntax and the semantics of the calculus, as well as the rules and axioms used to manipulate the syntax in proofs. We discuss the spi calculus in detail as defined by Abadi and Gordon, then the various challenges faced during the implementation of the calculus and our rationale for the decisions made in the process.

Acknowledgements

Major thanks go to my supervisor, Dr. Michael Winter, for his invaluable advice, guidance and assistance over the course of this work, as well as for his contributions to the code regarding process equivalence, basic barbs and all the other minor lemmas. I have learned so much, mostly about how much I do not know yet.

Thanks as well to my fellow MSc. candidate, Tyler Collins, for starting me on this roller coaster of a project by telling me about the spi calculus when I probably could have picked an easier topic. I will pay you back for it in full.

Finally, thank you to all the undergraduate students around the department that helped me maintain some semblance of a social life over the past year. Your putting up with my constant blathering and griping about spi calculus was immensely helpful, even if it didn't seem like it.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In today's society, there are countless computer programs being used to do countless tasks, ranging from as mundane as entertainment to as important as financial management or missile guidance. When dealing with applications of the latter sort, there exists an inherent need for guarantees that software will do what is intended to do without error. Even in entertainment this can be the case; the accuracy of a braking system for a roller coaster suddenly seems much more important when it is hurtling passengers over hills at nigh-on unsafe speeds. Just as computer programs can benefit from (and often require) verification to ensure they meet certain standards, it can be vital for communication protocols used by computers to be guaranteed safe. The spi calculus was proposed by Martin Abadi and Andrew D. Gordon for exactly this purpose [1]. However, such proofs always contain a potential for human error when crafted by hand. As such, it would be beneficial to anyone looking to verify their protocol to be able to generate their proofs in a strict automated proof environment.

To do this, we need a system for modelling and reasoning about protocols. Since protocols are essentially just a series of instructions, a process algebra is suitable for our purposes. A process calculus is a type of calculus that focuses on representing the interactions between concurrent entities, such as two computers communicating over a network. Process calculi also provide rules for the manipulation of these models. CSP, developed by C.A.R. Hoare, is one of the earliest examples of a process calculus [7]. We want to use a process calculus that is specialized toward the modelling of cryptographic protocols, known as the spi calculus. The spi calculus is heavily based on Robin Milner's CCS [11], another early process calculus, as well as its successor pi calculus [12], while incorporating aspects of the chemical abstract machine developed by Gérard Berry and Gérard Boudol [3] and ideas from the work of De Nicola and

Hennessy [5].

There has been previous work done in the area of formal verification of cryptographic protocols prior to the creation of the spi calculus. A method of machine-aided protocol verification was presented by R. A. Kemmerer utilizing theorems and state invariants [8]. Other works include that of Gray and McLean, which utilizes temporal logic [6], and of Lampson et al. [9] and Liebl [10] which discusses reasoning about authority based on relations and channels in the context of various security mechanisms.

The intention of this thesis is to produce an implementation of the spi calculus as it is defined the original paper by Abadi and Gordon, show that the implementation works for the purpose of modelling and proving properties of protocols, and discuss the work done so that anyone can use it to perform protocol verification. The implementation involved several stages and required several major readjustments of the basic definitions of the syntax and semantics of the calculus. The overall methodology of the work and individual motivation for each of these major adjustments will be covered, with the intention of explaining why all the parts of the implementation have been designed the way they have. In general, an attempt was made to produce a version of the spi calculus that is usable in a proof system and also leverages the strengths of the Coq proof system wherever possible.

1.1 Terminology

There is a significant amount of terminology inherent to working with any calculus, as well as with regard to the spi calculus specifically and to the Coq proof system in which the implementation was written. This section will attempt to cover all of the terminology necessary for understanding the spi calculus and its implementation in the system.

General Concepts

One of the most basic concepts that must be considered is variables. Variables, whether in math, computer science or elsewhere, are identifiers that stand for other values. In the spi calculus a variable does just this, existing as a placeholder to be replaced later via a substitution. Substitution is the replacement of a variable with some other element. In the spi calculus, this element is a term. While this term could (based on the spi calculus definition of a term) be another variable, generally

the value substituted in is more meaningful than this.

One exception to this is the case of renaming conflicting variables. A variable is considered free if it has no ties to any syntax around it, and is considered bound otherwise. In the simple case of a mathematical function $f(x)$, any occurrences of the variable x would be bound within the body of the function f, as whatever input is provided to f must replace the occurrences of x within the body. To put it another way, a bound variable cannot be changed individually; all occurrences of a bound variable must be changed within the context it is bound in. Bound variables also cannot be substituted for terms as they are tied to their binding operation. Free variables are essentially the opposite, and can be substituted for at any time. In the case of $f(x)$, substitution can occur for the bound x when a value is provided because substitution only considers the body of the function, in which the binding operation $f(x)$ is not present. This lengthy explanation of free and bound variables ties substitution to the concept of renaming, in which a term contains a bound variable as a free variable and is substituted in. In this case, the currently bound variable must have all instances replaced with some new identifier, hence it is renamed.

Relations

Relations are another simple but necessary concept that pertains to the spi calculus. A relation is an abstract connection between elements; the elements can be said to be related if their pairing fulfills whatever condition the relation represents (which could be entirely arbitrary). Relations are usually defined on specific types, such as the set of natural numbers. "*Less than*" is a common relation, and the pair $(1, 2)$ is within that relation assuming we interpret the pair as "1 is *less than* 2". In the spi calculus, the elements being related are terms, processes and agents, the three main types in the calculus.

Two other concepts that appear frequently in the thesis are closedness and stuckness, for lack of a better term. Closedness applies to terms, processes and agents, while only processes are capable of being stuck. Closedness will be discussed in detail as part of the implementation, but having defined free variables above, it can be defined as the state of having no free variables. All the relations defined in the spi calculus are for closed constructs. When a process is stuck, it has reached a state where it can no longer perform any actions. This could be because it has executed until reaching a nil process (which does nothing by design) or because a deterministic

process such as a term match or case contains invalid or unexpected input.

Coq

It is also necessary to consider some terminology of the Coq proof system. Common keywords in the system include *Definition*, *Fixpoint*, *Lemma*, and *Proposition*. *Definition* and *Fixpoint* are used to declare non-recursive and recursive functions respectively, while *Lemma* and *Proposition* are used to indicate the declaration and subsequent proving of a fact. When the system is in proof mode, we refer to the current fact being proven as the *goal* (there can be multiple goals, called *subgoals*) and the information above the goal as *assumptions*. Also common are the keywords *Proof*, *Qed*, and *Defined*. *Proof* is a statement that a proof is beginning (generally after a *Lemma* or similar), while *Qed* and *Defined* are used to end a completed proof. *Defined* tells the system that the internals of the proof are relevant while *Qed* discards them and simply accepts the proven fact as true. There are some intricacies to these keywords, such as the fact that *Lemma* and *Proposition* have the same effect and so only differ aesthetically, but their meanings as have been stated are sufficient for the purposes of this thesis.

Another commonly-used and important keyword in Coq is *Inductive*. An inductive declaration can be colloquially seen as a data type declaration, but can be considered more literally as the ability to define a series of constructors for that new type in the sense that each constructor takes in some other types (or nothing at all for terminal forms) and produces an element of the stated inductive type. Examples of inductives in the implementation of spi calculus include the simple Name (containing one constructor N that takes string and produces a Name) and the multi-faceted Term (which has six different constructors representing the six kinds of terms).

In Coq, it is possible to define a type which is all the elements of a given type such that they fulfill a given property. These types are called dependent types. The implementation makes use of these dependent types to define closed versions of terms, processes and agents, though this results in some complications. Coq is very strict about the proof component of a dependent type (that is, the proof that the property is met by the element) and does not always consider two proofs of the same fact equal. It is important to take this into account when discussing the implementation, particularly regarding the topics of setoid equivalences and the calculus rules.

Coq also possesses type classes, a means of defining a set of properties that a type of that class should satisfy. A good example of this is the Equivalence class, which requires that the provided type is a relation on some other type, and that the relation is reflexive, symmetric and transitive. A type can be stated as a member of a class using the *Instance* keyword, at which point the system requires a proof that the type possesses all the properties of that class. We use the Equivalence class several times in the implementation, and the Proper class as well.

Tactics

When proving facts in the implementation, we make use of many different proof tactics available in the Coq system. It is important to have a general idea of what these tactics are doing to be able to understand the proofs themselves. The descriptions of these tactics will not be representative of every possible usage, but should give some indication of what is happening when they are used. The first tactic commonly seen at the start of a proof is *intros*. This tactic strips off any instances of a *forall* quantifier at the start of a goal and converts them into assumptions, along with any premises stated prior to the final implication. The *intros* tactic is used any time we have an assumption in the goal that needs to be "moved up". It is almost universally used at the start of proofs in the implementation, but is sometimes used elsewhere when new assumptions are produced in the goal. It may also be used after an induction, so that an important variable remains general in the induction.

As its name implies, the *induction* tactic performs an induction on a target variable or assumption assuming it is of an inductive type. This amounts to a case analysis with the addition of an induction hypothesis for non-terminal constructor cases. There are several other tactics for case analysis: *case*, *case_eq*, *destruct*, and *inversion*. *Case* is a very basic form of case analysis; proofs in the implementation tend to use destruct for simple case distinction or when no induction hypothesis is needed. *Case_eq* is used in place of *destruct* when we want to guarantee the addition of the current case to our assumptions. *Destruct* often achieves this on its own (for example, when used on a decidability function), but not in all cases. *Inversion* is used for case analysis on an inductive assumption; it produces a case for each constructor of the inductive that fits based on the values in the current instance. In layman's terms, it is a means of analyzing "how we got here" with regard to an inductive.

Other common tactics in Coq include *apply*, *rewrite*, and *simpl*. These tactics allow

us to transform the goal based on already proven facts, assumptions, and definitions. *Apply* is used in the context of one-way lemmas (ie. $P \rightarrow Q$) to move in the appropriate direction based on whether we apply the lemma to an assumption (P becomes Q) or the goal (Q becomes P). *Rewrite* is similar, but is bidirectional. It is used in the case of equalities and equivalences. This includes logical equivalence, or if-and-only-if (also represented as *iff* or $\leftrightarrow$). If we have the fact $P = Q$ we can use rewrite to turn P into Q, or rewrite $\leftarrow$ to do the converse. *Simpl* is a very useful tactic; it simplifies any syntax by referencing the definition of functions and lemmas. If the parameter of such a definition can match a case of the implementation, it will replace the current syntax with the appropriate case. One use of *simpl* will perform as much simplification as possible, but in the event this is not desired we can specify the definition to simplify.

When we have used the above tactics and others to reach our stated goal, there are several tactics that can be used to close the goal. Akin to *simpl*, *trivial* will finish the current goal if it simplifies to an assumption or a reflexive equality. It will not close a reflexive equivalence; for any goal in which both sides match, we can use *reflexivity* as long as the relation is proven to be reflexive. Equality, equivalence and if-and-only-if have this property by default. These are the most common tactics for closing a goal, but there are a few specialized tactics such as *tauto* and *contradiction*. The former closes a goal if it is a logical tautology (like $\neg False$) while the latter checks for the presence of a contradiction in the assumptions. It is a rather strict check, generally requiring that the contradictory assumptions simplify to P and its exact negation $\neg P$. *Inversion* can also be used to close a goal in a manner similar to contradiction. If an inductive constructor in the assumption contains an impossibility, *inversion* will end the current goal but produce no new cases as it is impossible to get to this assumption from any previous assumptions.

This concludes the discussion on tactics in Coq. Any tactics present in the implementation that have not been discussed here can be referenced at the Coq website's extensive tactic index, which contains explanations of every tactic in the system along with their parameters and optional extensions.

1.2 Structure

The structure of the thesis is as follows: we begin with a brief primer on terminology regarding the Coq proof system and other basic concepts, followed by an outline of the

works that led to the development of the spi calculus. We will proceed to outline the spi calculus as created by Abadi and Gordon, then address the syntax, semantics and rules of the calculus in detail to give context to our discussion on its implementation. After all of this background information, we will provide an in-depth discussion on the features of the implementation, how they were developed, and the challenges we faced in doing so. We will then provide and discuss an example of the implementation's usage, and conclude with the major lessons learned over the course of this work. The completed work can be found online at the address provided at the end of this thesis [16].

Chapter 2

Background

While the focus of this thesis is on the spi calculus developed by Abadi and Gordon, it was developed with pi calculus as a base. In this chapter we will discuss pi calculus and its predecessor CCS, both developed by Robin Milner. An understanding of these systems is necessary to highlight what the spi calculus introduces, as well as what it inherits from Milner's work.

2.1 CCS

CCS was originally created by Robin Milner in the early 1980s [11]. It is a process calculus centered around communications between processes in a system, whether those processes are true computer processes or simply another type of actor in a system of related agents. The calculus can be viewed as a set of processes (i.e. P_1, P_2, P_3, etc.) connected to each other through actions or "ports" that correspond to what each process can do. Actions are generally labeled as lower case letters (p, q, r, etc.) and belong to the set of all names in the universe of discourse. Variables are also used in CCS (and are labeled as x, y, z, etc.). The set of names also has a complement: the set of co-names (denoted by the same action with a bar over it) indicating a reaction to the initial action. The nameless action τ (tau) is also important, denoting an action internal to the process that possesses no name (and thus no connection to another process). In addition to these symbols, the syntax also features five combinators that can be used to model these systems on a low level. These combinators are shown in Table 2.1. Prefix is a basic operation, indicating that the system can take action a and then continue with process P_1. Summation and composition indicate that the involved processes can be chosen between or are running simultaneously, respectively. Renaming is the action of replacing all instances

of action x with action y, and restriction internalizes the action a to the process P_1, changing the free name a into a bound name within that context. Free names can be used anywhere in a definition, while bound names are restricted to the scope they have been limited to. Beyond these symbols and combinators, CCS also includes a null process 0, which indicates that nothing is to be done.

Table 2.1: CCS Syntax

Operation	Syntax
Prefix	$a.P_1$
Summation	$P_1 + P_2$
Composition	$P_1 \mid P_2$
Renaming	$P_1[y/x]$
Restriction	$P_1 \backslash a$

As a simple example of CCS, we could define a process with two actions p and q and further subprocesses A and B. A process with these components could look like this:

$$S \stackrel{\text{def}}{=} p.A + q.B \tag{2.1}$$

This system definition can execute by performing action p then running A or by performing action q and then running B. It is noteworthy that this choice is non-deterministic; no mechanism exists to decide which action is performed. By abstracting out components of a system and defining them separately in this manner, much more complex interactions can be modeled at a very low level. As a side note, while CCS and pi calculus both use $\stackrel{\text{def}}{=}$ for definitions, spi calculus instead uses $\triangleq$. The rules of CCS can be seen in Figure 2.1 [11].

2.2 Pi Calculus

The pi calculus is largely based on the previous definition of CCS [12]. It was developed by Robin Milner a decade later, with the intention of being a more accurate model of modern systems by taking mobility of connections into account. This notion of mobility is the focus of pi calculus, taking CCS and introducing the ability to pass process connections, or channels, as a fundamental part of the calculus. Put simply, this allows a static CCS model to not only pass messages, but also pass a connection

$$\textbf{Act}\ \frac{}{\alpha.E \xrightarrow{\alpha} E} \qquad \textbf{Sum}_j\ \frac{E_j \xrightarrow{\alpha} E'_j}{\sum_{i\in I} E_i \xrightarrow{\alpha} E'_j}\ (j \in I)$$

$$\textbf{Com}_1\ \frac{E \xrightarrow{\alpha} E'}{E|F \xrightarrow{\alpha} E'|F} \qquad \textbf{Com}_2\ \frac{F \xrightarrow{\alpha} F'}{E|F \xrightarrow{\alpha} E|F'}$$

$$\textbf{Com}_3\ \frac{E \xrightarrow{\ell} E' \quad F \xrightarrow{\bar{\ell}} F'}{E|F \xrightarrow{\tau} E'|F'}$$

$$\textbf{Res}\ \frac{E \xrightarrow{\alpha} E'}{E\backslash L \xrightarrow{\alpha} E'\backslash L}\ (\alpha, \overline{\alpha} \notin L) \qquad \textbf{Rel}\ \frac{E \xrightarrow{\alpha} E'}{E[f] \xrightarrow{f(\alpha)} E'[f]}$$

$$\textbf{Con}\ \frac{P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'}\ (A \stackrel{\text{def}}{=} P)$$

Figure 2.1: CCS Rules

like the action p to alter the topography of the model. This adjustment to the calculus results in several changes to the syntax.

Most importantly, the syntax $p(x)$ represents a reception of the message over the channel p, with the result being bound to variable x. For example, $p(x).A$ receives the message on p and then runs process A with all instances of x in A replaced by the message contents. To complement this new use of the syntax, $\bar{p}\langle \mathrm{x} \rangle$ now denotes the sending of what is contained within x on the channel p. Similarly to with CCS, the syntax $P \mid Q$ indicates the running of P and Q simultaneously. Prefix, summation and renaming operations are also carried over from CCS. Restriction has new syntax, creating a new channel x bound to P and then running P. Replication is a new operation that will repeatedly spawn copies of P. It was added to allow for the defining of parametric processes using recursive equations without the use of $\stackrel{\text{def}}{=}$. The new syntax for the pi calculus is shown in Table 2.2.

Table 2.2: New Pi Calculus Operations

Operation	Syntax
Send	$\bar{p}\langle \mathrm{x} \rangle.A$
Receive	$p(x).A$
Restriction (new syntax)	$(new\, x)P$
Replication	$!P$

A simple example of pi calculus provided by Milner involves three agents P, Q, and R, with P connected to Q by channel x and to R by channel z [12]. Initially z is

exclusive to the involved agents P and R, thus the system can be defined as:

$$S \stackrel{\text{def}}{=} new\, z\, (P \mid R) \mid Q \tag{2.2}$$

To illustrate the passing of channels, we can define P as $\bar{x}\langle \mathrm{z} \rangle.P'$, meaning that running P will send z along x and leave P in new state P'. Continuing with this idea, we can say that Q is defined as $x(y).Q'$, and so Q will receive the channel z sent by P and thus become capable of communicating with R in any further operations. If P does not contain any other means of using z, it loses access to the channel following this exchange. Thus, it can be said that the channel z was passed from between P and R to between Q and R. This basic system exemplifies one of pi calculus' most notable features. To conclude the example, the transfer of channel z results in the new system

$$S' \stackrel{\text{def}}{=} P' \mid new\, z\, (R \mid Q'') \tag{2.3}$$

where Q'' is the result of replacing all instances of y (a variable in Q) in Q' with z (the actual "value" received by Q).

There are several means of testing equivalence in both CCS and pi calculus, with each one implying a differing level of equivalence. Milner speaks at length on the concept of observational equivalence, the idea that two agents are essentially equivalent if all their external interactions with the rest of the system are identical. For the purposes of these calculi, the idea of observational equivalence is realized through bisimulation. Two agents are equivalent in their external functions if for each state of each agent, there exists a state in the other agent that simulates it (i.e. has at least as much capability) [15]. A bisimulation can be represented by a binary relation containing a set of pairs of states. However, on its own this is only a simulation (one-way); a bisimulation must also be a valid simulation if all the pairs are reversed. Beyond this, there are also two types of bisimulation: strong and weak. Strong bisimulation states that for two agents P and Q, each result of using action α with P must map to an equivalent result using the same action on Q. As stated before, all capabilities of P must be matched by Q, as must the converse. Weak bisimulation loosens this requirement, allowing for the use of silent actions (τ) to achieve a simulation.

There are rules in pi calculus for reaction and for transition; these rules can be seen in Figures 2.2 and 2.3 [12].

REACTION RULES

$$\text{TAU}: \quad \tau.P + M \rightarrow P$$

$$\text{REACT}: \quad (a.P + M) \mid (\overline{a}.Q + N) \rightarrow P \mid Q$$

$$\text{PAR}: \quad \frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q} \qquad \text{RES}: \quad \frac{P \rightarrow P'}{\text{new } a\, P \rightarrow \text{new } a\, P'}$$

$$\text{STRUCT}: \quad \frac{P \rightarrow P'}{Q \rightarrow Q'} \ \textit{if} \ P \equiv Q \ \textit{and} \ P' \equiv Q'$$

Figure 2.2: Pi Calculus Reaction Rules

TRANSITION RULES

$$\text{SUM}_t: \quad M + \alpha.P + N \xrightarrow{\alpha} P \qquad \text{REACT}_t: \quad \frac{P \xrightarrow{\lambda} P' \qquad Q \xrightarrow{\overline{\lambda}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\text{L-PAR}_t: \quad \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \text{R-PAR}_t: \quad \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'}$$

$$\text{RES}_t: \quad \frac{P \xrightarrow{\alpha} P'}{\text{new } a\, P \xrightarrow{\alpha} \text{new } a\, P'} \ \textit{if} \ \alpha \notin \{a, \overline{a}\}$$

$$\text{IDENT}_t: \quad \frac{\{\vec{b}/\vec{a}\}P_A \xrightarrow{\alpha} P'}{A\langle\vec{b}\rangle \xrightarrow{\alpha} P'} \ \textit{if} \ A(\vec{a}) \stackrel{\text{def}}{=} P_A$$

Figure 2.3: Pi Calculus Transition Rules

Chapter 3

The Spi Calculus

This section will explain the spi calculus as it pertains to the previous works of Robin Milner, as well as basic facts about the calculus which are important for covering its implementation later. Spi calculus expands on pi calculus, adding new capabilities that make it better suited to modelling cryptographic protocols [1].

3.1 Additions

The main new feature of the spi calculus is the ability to encrypt and decrypt messages. Encryption and sending of a message M using a specific key K is performed using the syntax $\bar{p}\langle\{M\}_K\rangle$. Decryption is performed similarly, but utilizes a syntax new to the calculus. $q(x).case\, x\, of\, \{y\}_K\, in\, F(y)$ will receive the encrypted message on q and substitute it into $\{y\}_K$. If the key K successfully decrypts the message, then the process F will run using the decrypted message. Otherwise the process is stuck. The new syntax $case\, M\, of\, 0 :\ P$ will run process P if M is 0, and is stuck otherwise. Other comparisons to M can be included following the first one; for example, $case\, M\, of\, 0 :\ P\, suc(x) :\ Q$. Although the syntax's lack of separation makes this a bit unclear, the case statement now also has the option of running Q if M is $suc(x)$. Regarding $suc(x)$, spi calculus also adds constructs for pairing and numbers. Specifically, (M, N) indicates a pair and 0 and $suc(x)$ are used when working with numbers. It is important to note that the number 0 and the nil process 0 are different, but identifiable based on the context. A pair splitting operation is also introduced; it functions as $P[N/x][L/y]$ if $M = (N, L)$, substituting the left and right members of the pair in place of x and y in P. Finally, a syntax for matching is provided: $match\, [M\, is\, N]\, P$ runs P if M and N are the same and does nothing otherwise. Abadi and Gordon treat this operation as part of the base pi calculus, although it is

not used in the version originally introduced by Milner [12]. All other operations are the same as was defined by Milner, although restriction has again been adjusted to use the symbol ν (nu) instead of the word *new*. Table 3.1 lists the syntax that differs from pi calculus.

Table 3.1: New Spi Calculus Operations

Operation	Syntax
Encryption	$\{M\}_K$
Decryption	$case\, L\, of\, \{x\}_K\, in\, P$
Restriction (new syntax)	$(\nu\, x)P$
Matching	$[M\, is\, N]\, P$
Pair Splitting	$let\, (x, y) = M\, in\, P$

The first examples provided for spi calculus demonstrate a simple sending and receiving of an encrypted message from A to B, with B then performing a function F using the decrypted message. From this, two important properties can be obtained. For the definition of these properties, it is important to note that $\simeq$ indicates something similar to but not identical to bisimilarity, and means that both sides of the equation behave identically to an outside observer. Specifically, it means that $P(M)$ behaves identically to $P(M')$ where pi calculus' bisimilarity would distinguish them.

The secrecy property is the guarantee that the message M cannot be intercepted by a third party in-transit, while the authenticity property is the guarantee that an attacker cannot force B to perform F using anything other than what is sent by A [14]. The secrecy property can be represented as such: if $F(M) \simeq F(M')$ for all M and M', then secrecy is guaranteed. Authenticity is slightly more involved, and requires the introduction of a "magic" version of the system in which B knows the message M from A in advance. If $inst(M)$ is the initializing process that sets up the channel for A and B to communicate, as well as A and B themselves, then $inst_{spec}(M)$ is the "magic" version. The key difference between them is the latter providing M to both A and B, rather than just A. With these two versions defined, authenticity can be guaranteed if $inst(M) \simeq inst_{spec}(M)$ for all M. The property shows that the whole system's performance in the specification is the same as in the original. In other words, nothing suspicious can occur between the version that must send M over a channel and the version that provides it directly to B.

Spi calculus functions similarly to pi calculus, so it is more effective to demonstrate its use by modelling a pre-existing cryptographic protocol as is provided by Abadi and Gordon [1]. The Wide-Mouthed Frog protocol is a means of communication facilitated through a central server. Clients send the server the key which they will use to communicate with another client. The server then provides that key to the intended recipient of the forthcoming communication. In the spi calculus, the clients A and B and server S can be represented as follows:

$$A(M) \triangleq (\nu\, K_{AB})(\overline{c_{AS}}\langle\{K_{AB}\}_{K_{AS}}\rangle.\overline{c_{AB}}\langle\{M\}_{K_{AB}}\rangle) \tag{3.1}$$

$$B \triangleq c_{SB}(x).case\, x\, of\, \{y\}_{SB}\, in\, c_{AB}(z).case\, z\, of\, \{w\}_y\, in\, F(w) \tag{3.2}$$

$$S \triangleq c_{AS}(x).case\, x\, of\, \{y\}_{K_{AS}}\, in\, \overline{c_{SB}}\langle\{y\}_{K_{SB}}\rangle \tag{3.3}$$

Put simply, A creates a new key to communicate with B, sends that key to S using the pre-existing client-server key, and then sends its message to B using the sent encryption key. S receives the key sent by A, attempts to decrypt it using the pre-existing client-server key and passes the decrypted key on using the separate client-server key known to S and B. Finally, B receives the encryption key sent by S, decrypts it using the appropriate key and then receives the message from A, attempting to decrypt it using the received key. If all keys were correct, B will finally run F using the sent message. Although the syntax can seem complex, it allows for very specific low-level representation of protocols where precision is of the utmost importance.

The main construct in the spi calculus is the process, with terms serving as data. For the most part, process equality is determined based on whether two processes are composed of the exact same subprocesses and terms. There is one caveat to this: since bound variables (such as the x in $c(x).P$) serve only as placeholders for later results of computation, the exact symbol does not matter so long as it occurs in all the same positions. In this respect, the processes $p(x).\overline{p}\langle x\rangle.0$ and $p(y).\overline{p}\langle y\rangle.0$ are equal, as they both receive a message on channel p and then send that message back on the same channel. What the placeholder is called does not matter, since x and y are both bound variables that appear in all the same places.

The spi calculus also possesses two different kinds of semantics: one which is based on the notion of reaction, and one which is based on a labelled transition system akin

to Milner's previous works. The former is introduced first and relies solely on terms and processes, while the latter is introduced as a supplemental semantics and utilizes an extra construct known as an agent. While terms and processes are defined recursively, agents are all made up of exactly one process at minimum with no recursion. The distinction between these semantics will be further addressed later on during discussion of the implementation.

3.2 Rules

The spi calculus has many rules for manipulation of its syntax. These rules fall under six categories: reduction, structural equivalence, reaction, barb, convergence, and commitment. We include these rules in Figures 3.1, 3.2 and 3.3, 3.4, 3.5, 3.6, and 3.7 respectively [1]. We will discuss these rules in detail in Chapter 6.

(Red Repl)	$!P > P \mid !P$
(Red Match)	$[M \text{ is } M]\ P > P$
(Red Let)	$let(x, y) = (M, N)\ in\ P > P[M/x][N/y]$
(Red Zero)	$case\ 0\ of\ 0: P\ suc(x): Q > P$
(Red Suc)	$case\ suc(M)\ of\ 0: P\ suc(x): Q > Q[M/x]$
(Red Decrypt)	$case\ \{M\}_N\ of\ \{x\}_N\ in\ P > P[M/x]$

Figure 3.1: Reduction Rules

$$\text{(Struct Nil)} \qquad P \mid \mathbf{0} \equiv P$$

$$\text{(Struct Comm)} \qquad P \mid Q \equiv Q \mid P$$

$$\text{(Struct Assoc)} \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R$$

$$\text{(Struct Switch)} \qquad (vm)(vn)\ P \equiv (vn)(vm)\ P$$

$$\text{(Struct Drop)} \qquad (vn)\ \mathbf{0} \equiv \mathbf{0}$$

$$\text{(Struct Extrusion)} \quad (vn)(P \mid Q) \equiv P \mid (vn)\ Q \qquad \text{if} \quad n \notin fn(P)$$

Figure 3.2: Structural Equivalence Axioms

$$\text{(Struct Red)} \quad \frac{P > Q}{P \equiv Q} \qquad \text{(Struct Refl)} \quad \frac{}{P \equiv P} \qquad \text{(Struct Symm)} \quad \frac{P \equiv Q}{Q \equiv P}$$

$$\text{(Struct Trans)} \quad \frac{P \equiv Q \quad Q \equiv R}{P \equiv R} \qquad \text{(Struct Par)} \quad \frac{P \equiv P'}{P \mid Q \equiv P' \mid Q} \qquad \text{(Struct Res)} \quad \frac{P \equiv P'}{(vm)\ P \equiv (vm)\ P'}$$

Figure 3.3: Structural Equivalence Rules

$$\text{(React Inter)} \quad \bar{m}\langle N \rangle . P \mid m(x) . Q \rightarrow P \mid Q[N/x]$$

$$\text{(React Struct)} \quad \frac{P \equiv P' \quad P' \rightarrow Q' \quad Q' \equiv Q}{P \rightarrow Q}$$

$$\text{(React Par)} \quad \frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q} \qquad \text{(React Res)} \quad \frac{P \rightarrow P'}{(vn)\ P \rightarrow (vn)\ P'}$$

Figure 3.4: Reaction Rules

(Barb In) $m(x).P \downarrow m$ (Barb Out) $\bar{m}\langle M\rangle.P \downarrow \bar{m}$

(Barb Par) $\dfrac{P \downarrow \beta}{P \mid Q \downarrow \beta}$ (Barb Res) $\dfrac{P \downarrow \beta \quad \beta \notin \{m, \bar{m}\}}{(\nu m)\, P \downarrow \beta}$ (Barb Struct) $\dfrac{P \equiv Q \quad Q \downarrow \beta}{P \downarrow \beta}$

Figure 3.5: Barb Rules

(Conv Barb) $\dfrac{P \downarrow \beta}{P \Downarrow \beta}$ (Conv React) $\dfrac{P \to Q \quad Q \Downarrow \beta}{P \Downarrow \beta}$

Figure 3.6: Convergence Rules

(Comm In) $\dfrac{}{m(x).P \xrightarrow{m} (x)\, P}$ (Comm Out) $\dfrac{}{\bar{m}\langle M\rangle.P \xrightarrow{\bar{m}} (\nu)\langle M\rangle\ P}$

(Comm Inter 1) $\dfrac{P \xrightarrow{m} F \quad Q \xrightarrow{\bar{m}} C}{P \mid Q \xrightarrow{\tau} F@C}$ (Comm Inter 2) $\dfrac{P \xrightarrow{\bar{m}} C \quad Q \xrightarrow{m} F}{P \mid Q \xrightarrow{\tau} C@F}$

(Comm Par 1) $\dfrac{P \xrightarrow{\alpha} A}{P \mid Q \xrightarrow{\alpha} A \mid Q}$ (Comm Par 2) $\dfrac{Q \xrightarrow{\alpha} A}{P \mid Q \xrightarrow{\alpha} P \mid A}$

(Comm Res) $\dfrac{P \xrightarrow{\alpha} A \quad \alpha \notin \{m, \bar{m}\}}{(\nu m)\, P \xrightarrow{\alpha} (\nu m)\, A}$ (Comm Red) $\dfrac{P > Q \quad Q \xrightarrow{\alpha} A}{P \xrightarrow{\alpha} A}$

Figure 3.7: Commitment Rules

Chapter 4

Syntax

The syntax of the spi calculus consists of terms, processes, agents, and various relations. In this section we will outline the syntax for the basic constructs of the calculus and briefly cover the symbols used for the relations on those constructs; the relations themselves will be discussed later. It is important to keep in mind that the spi calculus is based off a process calculus intended to model interacting systems in a mainly non-deterministic manner. The syntax of the spi calculus builds on this to model deterministic protocols and bears some resemblance to a programming language as a result. All of the basic constructs and relations in the implementation are defined as inductives; we will first discuss the original syntax proposed by Abadi and Gordon [1]. Their actual definition and usage in the implementation will follow. Notation choices and discussion will be deferred to Chapter 8.

4.1 Terms

Terms in the spi calculus act as data values. They are the elements that are sent and received by channels, and can be encrypted before transfer. The set of terms is defined by the grammar in figure 4.1. Consequently, the syntax of terms is fairly simple. There are three terminal kinds: names, variables, and zero. There are no visual distinctions between names and variables as they are both represented as strings of characters. The zero term appears as the numeral 0. Regarding the natural number representation of the calculus, the successor term $suc(M)$ can build up further terms, and the pair term bundles two terms together with a standard syntax of parentheses and a comma. Encryptions are represented as a target term within curly braces and a shared-key term in subscript beside.

$L, M, N ::=$	terms
n	name
(M, N)	pair
0	zero
$suc(M)$	successor
x	variable
$\{M\}_N$	shared-key encryption

Figure 4.1: Spi Calculus Terms

Terms are defined inductively in the implementation and by default do not mirror the syntax of the actual calculus. As with any inductive type in Coq, they are represented as a specific constructor for that inductive followed by the parameters for the type. The constructors for a term are, in order: TName, Pair, Zero, Suc, TVar, and Encryption. Names and variable constructors are more specific as names and variables can exist in contexts where other terms are invalid. These constructors emphasize that the values are treated as terms when they appear as such. Outside the context of a term, names and variables are aliases for the string type. As an example of constructor usage, a pair of two terms M and N would be represented as $PairMN$.

We define notations for each of these constructors to better resemble the original syntax. Our choices for term notation can be found in Table 8.4.

4.2 Processes

Processes in the spi calculus, as their name implies, are the various operations that can be performed on terms (data) using the calculus. Terms can be sent from place to place, used for branching and matching, or for other purposes. Processes can also be used to arrange parallel composition (processes that run at the same time). The set of processes is defined by the grammar in Figure 4.2.

As stated previously, the process syntax resembles that of a programming language. Operations can be done in sequence (or parallel) and each process executes in series so long as it is capable of doing so. Hence, all processes except for the nil process require a further process to run after they complete; this is reflected in the syntax for processes. Each process takes the information it needs to perform the task and a process

$P, Q, R ::=$	processes
$\bar{M}\langle N \rangle . P$	output
$M(x) . P$	input
$P \mid Q$	composition
$(\nu n)\ P$	restriction
$!\, P$	replication
$[\, M \textit{ is } N \,]\ P$	match
$\mathbf{0}$	nil
$\textit{let } (x, y) = M \textit{ in } P$	pair splitting
$\textit{case } M \textit{ of } 0 : P \textit{ suc}(x) : Q$	integer case
$\textit{case } L \textit{ of } \{x\}_N \textit{ in } P$	shared-key decryption

Figure 4.2: Spi Calculus Processes

to run afterward. Figure 4.2 does a reasonable job outlining the ten kinds of processes, so we will skip the discussion of individual syntax as defined by Abadi and Gordon [1].

The inductive constructors for a process in the implementation are mostly identical to the names of the kinds of processes. The only differences are in the pair splitting, integer case and shared-key decryption processes, which have constructors Split, Case and Decryption for brevity's sake. The parameter order for process constructors reflects the left-to-right appearance of elements within processes (for example, output is $Output M N P$) except for the Split and Decryption constructors. A pair split is implemented as $Split\ M\ x\ y\ P$ and a decryption is implemented as $Decryption\ M\ N\ x\ P$ to better represent the fact that the variables are being used in the next process.

We define notations for each of these constructors to better resemble the original syntax. Our choices for process notation can be found in Table 8.1.

4.3 Agents

The last kind of spi calculus construct is an agent. Agents are represented by three syntactic forms: abstractions, concretions, and processes. Agents are used alongside a semantics which is different from the main semantics of the calculus, but can be used in reasoning about processes as well. The set of agents is defined by the grammar in Figure 4.3. Note that agents are not defined recursively. An agent contains

at minimum a process, and no agent is made up of other agents.

$A, B ::=$	agents
$(x)\ P$	abstraction
$(\nu \vec{m})\langle M \rangle\ P$	concretion
P	process

Figure 4.3: Spi Calculus Agents

Abstractions and concretions are semantically analogous to input and output processes, and as a result they are represented with a similar syntax. Abstractions are input processes without the channel preceding the variable in parentheses, while concretions drop the channel and also include a list of names where it would be. The list does not replace the channel semantically because it is required for use with the alternate semantics of agents while channels are not. In the implementation, abstractions and concretions can exist on their own as well as in the context of an agent. We define the single-constructor inductives for these types as Abs and Con. For agents the constructors are AAbs, ACon and AProc to represent their being considered as agents instead of as their own type.

Composition and restriction are also defined for agents. We use the same syntax for agent composition and restriction as we do for processes. The only unique operation for agents is interaction between an abstraction and a concretion. We denote an interaction by an infix @ symbol.

We define notations for each of these constructors to better resemble the original syntax; our choices for agent notation can be found in Table 8.3

4.4 Relations

There are many relations in the spi calculus. We outline the syntax for them in Table 4.1. All of the relations in the calculus are written infix, between the related constructs. Most of these relations are defined for closed processes. The exceptions are barb and convergence which relate a closed process and a name, and commitment which relates a closed process and a closed agent as well as the action bridging them. Regarding the latter, the label α above the commitment arrow stands for a general

action and may change depending on the commitment. Reaction and commitment both use a right arrow symbol because they have similar meaning but are part of separate semantics. A commitment can always be distinguished by the action label above the arrow.

Table 4.1: Relation Symbols

Relation	Symbol
Reduction	$>$
Structural Equivalence	$\equiv$
Reaction	$\rightarrow$
Barb	$\downarrow$
Convergence	$\Downarrow$
Testing Equivalence	$\simeq$
Commitment	$\xrightarrow{\alpha}$
Strong Bisimilarity	$\sim_s$
Barbed Equivalence	$\dot{\sim}$
Barbed Congruence	$\sim$

Chapter 5

Semantics

This section will provide a general understanding of the meaning and intent behind the constructs of the spi calculus. As stated previously, the spi calculus is designed around the ability to model and prove properties about cryptographic protocols. As a result, the constructs of the calculus are intended to allow for the definition of logical agents (in the sense that they act autonomously, rather than as agents per the calculus definition) that operate in parallel and communicate via channels. Of course, it is also possible to model a single logical unit operating on its own and communicating only with its internals, but this would be ignoring the capabilities the spi calculus offers. A detailed examination of the different kinds of constructs in the calculus follows.

5.1 Terms

Now we will outline the purpose of each kind of term. Names are a basic form of data that can represent channels (the connections between nodes in a system) but also pure data, as a name is just a string of characters. They are used as part of input and output operations to indicate which channel a piece of data is being transmitted on. Pair terms are self-explanatory; they are used to package two separate terms into one logical unit. Zero is necessary for the use of a natural number system as the terminal value, and $suc(M)$ (successor of M where M is any term) is used in conjunction with Zero to complete the natural number representation in the calculus. With this in mind, it follows that any term can be provided as the parameter for $suc(M)$, but ostensibly the only terms that fit with this construct are Zero and recursive uses of $suc(M)$. Any other kind of term inserted in this construct would be valid syntax, but would not make semantic sense. The next kind of term is a variable. Variables work

like they do in any mathematical or computational context. Variables can be substituted for any other term at appropriate times during calculus operations. Finally, encryption takes a term M and a term N, and uses N to encrypt M. This would usually be a name (which would function as a shared key), but as with the successor term any other term is a valid option.

Terms in the calculus serve as data to be passed around via processes or agents, but there are some intricacies to their usage. As stated previously, the term $suc(M)$ is intended to be used as a natural number representation in concert with the term Zero. However, the statement $suc(n)$ (where n is a name) is valid syntax. It is not meaningful in any way, and such cases are not considered in the implementation even though they are possible. Encryption terms can also have odd possibilities, such as trying to encrypt a term M (which is meaningful regardless) using Zero, a pair, or another encryption. These usages are less outlandish than the previous example and could have some real applications, since decryption succeeds as long as the term used for decryption matches the term used for encryption. However, it can be said for certain that encrypting a message with Zero is not a secure approach.

5.2 Processes

Now we will outline the purpose of each kind of process. An output process takes two terms M and N and a process P, and sends the term N using term M before running P. Obviously, the term M should be a name (channel) or a variable that can be instantiated to a name. Input works analogously, but the payload must be a variable, which becomes bound in P as the actual value of the received term. Parallel composition takes two processes and sets them to run concurrently. Restriction binds a name n within a process P. Replication takes a process P and spawns a theoretically infinite number of copies of it. Matching takes terms M and N and runs P only if the terms are identical; the matching operation becomes stuck otherwise. 0 (henceforth known as Nil) is the process that does nothing. Pair splitting binds x and y in process P as the left and right element of term M assuming M is a pair term; the process is stuck otherwise. A case statement works directly with the natural number representation: given a term M and processes P and Q, the operation runs P if M is Zero and Q if M is $suc(x)$ (a successor term). The case operation is stuck otherwise, and in the successor case the variable x is bound in Q as the subterm of the successor. Finally, decryption takes terms L and N as well as a variable x and process P. The

decryption succeeds if L is an encryption of some term M that was encrypted using N. The process P is then run with x bound as the result of the decryption. The decryption is stuck if any of these requirements are not met.

Processes also have some cases of valid syntax with invalid semantics. The simplest case is that of input and output processes. The first parameter of an input or output is a term indicating the channel to send the message on. A channel is indicated by a name, but the parameter indicating the channel can be any term. This is due to the potential for a channel to be represented by a variable which will later be instantiated to a name via substitution. As a result, any term that is not a name or variable would result in invalid semantics here. As with invalid term semantics, nothing is done in the calculus to address these possibilities. It is expected that the user will not attempt to prove things about semantically invalid processes just as one would not prove things about meaningless programs.

There are several other cases of processes that are invalid semantically, but these cases are addressed in the rules in the calculus. Match, Case, Split and Decryption processes are used to do checking or branching on specific kinds of terms. Match looks for two identical terms before continuing, Case branches based on whether a term is Zero or $suc(M)$, Split takes a pair term and separates the left and right elements in a further process, and Decryption will decrypt an encrypted term (ie. Encryption M N) using the provided term as a key before continuing. In each of these cases only specific terms make sense. A Match can only proceed if the terms are identical, a Case can only proceed if the term is Zero or $suc(M)$, a Split can only proceed if the term is a pair, and decryption can only proceed if the first term is an Encryption and the second term matches the term used to encrypt it. These situations are addressed by the Reduction relation, which has rules that can only be applied if the terms in these processes are as the processes expect. Otherwise, no rules can be applied and the processes are considered to be stuck.

5.3 Agents

The process form of agent is identical to the previously defined process; in other words, all processes are agents. This is why agent semantics will work for reasoning about any process as well. An abstraction can be seen as an alternate form of input operation. Where an input operation can be written as $c(x).P$, an abstraction is

written as $(x)P$. The obvious difference between abstractions and inputs is that the channel is dropped from an abstraction. Concretions are similar, but expand more so upon output operations. An output operation can be written as $c\{M\}.P$, while a concretion is written as $(\nu n)\{M\}P$. In addition to dropping the channel from the syntax, a concretion also includes a restriction on a list of names. It is important to note that this construct restricts a list of names, where a normal restriction only restricts a single name. Multi-name restrictions only occur in agent-related aspects of the calculus, and necessitated the definition of such a function in the implementation of the calculus. Composition and (single) restriction, originally defined as process constructs, are also defined for agents with some side conditions on a per-agent form basis (ie. abstractions and concretions each have their own unique side conditions while processes do not have such side conditions).

Agents do not have any situations in which their valid syntax does not make semantic sense, aside from in the case that an agent is a process. Abstractions and concretions omit the channel that input and output processes use, so they do not have any general terms as parameters. There are side conditions on agent compositions and restrictions that can invalidate such constructions, but these cases would have to be evaluated based on the specific side conditions of each operation. In the implementation, these side conditions are automatically handled by the system.

Composition and restriction, initially defined for processes, can also be used with agents. Since agents are not defined recursively and all have an internal process, these operations generally apply to the internal process as long as no side conditions are broken. Additionally, we do not compose agents with other agents but instead compose agents with processes only. For compositions on the left and the right, we compose the internal process of the agent with the incoming process. The incoming process cannot contain any names or variables free that are bound in the agent. Restriction of agents is simple for abstractions and processes, but since concretions have an inherent list of bound names, it is necessary to consider this list when restricting a new name. If the new restricted name occurs free in the term of the concretion, the name is added to the concretion's list. Otherwise, it can be ignored by the concretion and passed into the internal process as a normal restriction. Restriction of names which are bound within the agent already is allowed, but is not meaningful.

Interactions are also possible with agents, since they act as the labelled transition

semantics equivalent of reaction between an input and an output process. The result of two agents interacting is a process, and the process is restricted by the list of bound names of the concretion. As such, the abstraction cannot contain any of the concretion's bound names free. This would result in a different process semantically.

It may be confusing as to what the difference is between abstractions and concretions versus input and output processes. It ultimately comes down to a choice of semantics. It is possible to prove facts regarding spi calculus constructs using either the reaction based semantics, or the labelled transition semantics. Abadi and Gordon discuss some of these differences in their introduction of the commitment relation (the labelled transition semantics). Structural equivalence is stated to make proofs more awkward, so the labelled transition semantics uses commitment and agents without appeal to structural equivalence to allow for a different approach to these proofs [1].

Chapter 6

Rules

In this section, we will discuss the rules of the spi calculus and how they have been implemented. The rules of the spi calculus are divided into six kinds: reduction, structural equivalence, reaction, barbs, convergences, and commitment. Each set of rules can be said to constitute a relation, except for barbs and convergences which are essentially predicates. Regardless, each set of rules relates elements to other elements and they will be generally referred to as relations from here on. All the relations in the calculus are defined only on closed constructs if such a property exists for that type. Reduction, structural equivalence and reaction are relations between two closed processes, while barbs and convergences are relations between a closed process and a name. Commitment is a relation between a closed process, a closed agent, and an action.

All of the relations in the spi calculus are implemented as inductives with a type signature reflecting the relation. For example, structural equivalence is of the type $ClosedProcess \rightarrow ClosedProcess \rightarrow Prop$. Each relation has a set of constructors that accept the specific forms (of processes, terms, etc.) that the rule requires. Some rules are more general and only require that elements be part of a relation. An example of the former rule is StructNil, which states that the process $P \mid 0$ is equivalent to P. In Coq, this rule would be implemented as:

$$StructNil\ :\ forall\, P,\, Structural\, (Composition\, P\, Nil)\, P \tag{6.1}$$

An example of the latter rule is StructSymm, the symmetry property for structural equivalence. It would be implemented as:

$$StructSymm\ :\ forall\, P\, Q,\ Structural\, P\, Q\ \rightarrow\ Structural\, Q\, P \tag{6.2}$$

It is important to note that due to the nature of dependent types in Coq and their usage in the work for requiring closedness of terms, processes and agents, the actual implementation of these rules is slightly more complex than is shown here. Barring treatment of closedness, the rules would look as they do here.

Taking into account the definition of a closed process as a dependent type, the rules of the calculus must be adjusted slightly to compensate. A closed process in the implementation is the pair of a process and a proof that that process is closed (no free variables). In Coq, two proofs of the same fact are not necessarily equal, creating the situation that a rule may be applicable but cannot be applied due to the rule inductive's proof not matching the proof of the process in the goal or assumptions. Semantically we do not care about the proof, but due to Coq's strict rules regarding dependent types and proofs we are unable to use traditional equality to determine whether two closed processes are equal. We instead use a user-defined equivalence class via a setoid to state that two closed processes are equal if the internal processes are equal, bypassing the proof. This approach will be covered in detail later, but is necessary at this point to understand the final implementation of a rule such as StructNil:

$$StructNil\ :\ forall\, P\, Q\, R,\ Q\ ==\ P\,|\,0\ \rightarrow\ R\ ==\ P\ \rightarrow\ Structural\, Q\, R \tag{6.3}$$

A double equal sign is Coq's syntax for an equivalence, which we use instead of equality. This rule now states that two closed processes Q and R are structurally equivalent if they are themselves equivalent (per our definition) to the original left and right sides of the structural equivalence. With this consideration, any closed processes purportedly related by structural equivalence can have this rule applied, after which it remains to show that the internal process of the process/proof pair is of the correct form without appeal to whether it is closed or not. All rules in the implementation follow this convention to allow for ignoring of the closedness proof, and subsequently allow the proving of a series of lemmas stating that any relation in the calculus can have a closed process or closed agent replaced by an equivalent element.

6.1 Reduction

The reduction relation is intended to allow for certain processes to be "executed" and simplified to reach a point where a reaction can occur between input and output processes [1]. Such processes include replications, matches, pair splitting, case distinctions and decryptions. Reduction is a one-way relation between closed processes, and the resulting process is the act of reflecting the semantic meaning behind the process. For example, RedRepl turns the process $!P$ into $!P \mid P$, as replicating a process spawns a new parallel instance. The other rules are similar. RedMatch completes the match comparison as long as the two terms are identical, and the rules regarding pair splitting, case and decryption substitute the involved terms into the next process if the conditions are met. In RedSplit, this condition is simply that the term given to the split must be a pair term; the process does not make semantic sense otherwise. The other conditions are similar. RedZero involves no substitution, as the semantics of a zero case process indicate that the left process runs without any changes.

It is worth noting that, on paper, all of the rules of reduction are axioms. Their implementations in Coq are not technically axioms due to accounting for closedness, but semantically the rules are all still axioms.

6.2 Structural Equivalence

The rules regarding structural equivalence allow us to consider two closed processes equivalent even if they are not equal, as long as they will function identically when executed. Structural equivalence focuses mainly on parallel composition and restriction as a result. By the definition of the word, processes composed in parallel all run at the same time, so two processes that have the same subprocesses arranged in a different parallel order will perform the same. For restriction, as is the case in the declaring of new variables in a programming language, the order of declaration does not matter so long as none of the new identifiers are used in between the declarations. Hence, structural equivalence rules allow us to consider two processes with differently-ordered compositions or restrictions to be equivalence. There are also rules to allow such operations on the nil process to be ignored. For example, StructDrop states that $(\nu\, n)\, 0 \equiv 0$. Obviously the processes $(\nu\, n)\, 0$ and 0 will perform identically, so they are considered structurally equivalent. Colloquially, structural equivalence can be seen as having identical functionality when looking at the internals of two systems.

It is a similar idea to the concept of testing equivalence (which is covered later), but is a stronger statement about two processes. A relationship can be inferred between these equivalences through parts four and five of Proposition 7, namely that strong bisimilarity implies barbed congruence and barbed congruence implies testing equivalence [1].

Structural equivalence also has a rule StructRed, which states that any two processes related by a reduction are also structurally equivalent. This holds with the semantic intention behind structural equivalence; both sides of a reduction will perform the same since the left process of the reduction (the "before" state) is a check for some condition before the right process runs. The exception to this is RedRepl, which is structurally equivalent because it spawns infinite copies of the same process which will always do the same thing. Consequently, any fact proven about structural equivalence must also be true for reduction. This is important to consider when proving something about structural equivalence. Reduction may be a "one-way" relation, but when considered as a structural equivalence it becomes reversible via StructSymm. This means that any fact about reduction that is only true from left to right becomes untrue in the context of structural equivalence.

6.3 Reaction

The reaction relation represents a core aspect of the spi calculus. It relates the before and after states of a communication between closed processes. The notion of reaction was present in Milner's previous works, but was not represented in the same way; the commitment relation more closely resembles Milner's approach. The main focus of reaction is the result of an input process and an output process executing in parallel. The output process sends a message which is received by the input process, and this translates into the single axiom of the reaction relation, ReactInter. Aside from this axiom, there are three other simple rules. They allow for dropping of parallel processes and restrictions, as well as the replacement of the components of a reaction with structurally equivalent processes. ReactInter works exactly as reaction has been described: an output process and an input process communicating on the same channel in parallel interact to become a new parallel process composed of the output subprocess and the input subprocess, with the sent term substituted into the latter.

Aside from adjustments for consideration of closedness, reaction has a fairly straight-

forward implementation.

6.4 Barb

The barb relation functions as a predicate which indicates whether a process can communicate on a given channel. This check occurs via two axioms for the input and output processes, which are the only processes that perform actual communication. We can use these axioms to show that for a given closed process and name, the process is capable of communicating immediately via that named channel. When this is the case, it is said that the process exhibits a barb for that channel. The barb rules work similarly to those of reaction, allowing for the dropping of parallel processes and restriction, as well as the replacing of the process in the barb with one that is structurally equivalent.

There is an alternate version of the Barb predicate in the implementation known as BBarb, or "basic barb". This version arises from an issue with the Coq definition of Barb that prevents the closing of contradictory instances via inversion. Since the BarbStruct rule states that any structurally equivalent process can be substituted for the process in a barb, an attempt to do inversion on an impossible barb will result in the possibility that the process was structurally equivalent to some other process. Of course, the hypothesis was inherently contradictory so this is impossible, but the system cannot infer this. BBarb is defined as a separate logically equivalent predicate to Barb which keeps its version of BarbStruct separate from the rest of the rules. Consequently it is necessary to include versions of the Reduction rules in the context of Barb, since reduction implies structural equivalence and we want to continue to take these cases into account when dealing with barbs. However, this approach avoids the possibility of an infinite cycle of barbs in the event of a contradiction that requires an inversion.

6.5 Convergence

Convergence works very similarly to the barb predicate, being based entirely upon it with one addition. The convergence predicate holds if the closed process exhibits the barb after some reactions. Hence, if the process under a convergence was obtained via a reaction, we can instead consider the state of the process prior to that reaction. This is the only new rule of convergence. The other rule states that any barb is also

a convergence.

6.6 Commitment

The commitment relation exists almost entirely separate from the other aspects of the calculus. It is an alternate semantics relative to the previous relations, but is essentially equivalent and can be used in proofs about the previous semantics as well. Commitment is based on the concept of a labelled transition system instead of the notion of reaction, focusing on the idea of closed processes becoming closed agents via some *action*. While actions can be viewed as channels, they are not the same as names (which suffice to represent channels in the previous semantics). An action can be one of three things: a name, a co-name (indicated by a name with a bar above it), or the distinguished action τ (tau) [1]. Names and co-names are used for receiving and sending, respectively, when an external communication occurs. τ is used when the communication is internal to the system. In the implementation, names and co-names are not distinguished because there is no need to do so. It is possible to infer from the context what kind of name is involved and doing so is unnecessary. As a result, in the implementation an action is an inductive type which is either a name or τ.

Since commitment is an alternate form of the semantics given by the other five relations, it follows that its definition is fairly involved even on paper. The rules contain axiomatic cases for abstractions (input agents) and concretions (output agents) as well as the common rules for dropping of parallel elements and restrictions. Apart from these, there are two rules that represent the agent equivalent of ReactInter: CommInter1 and CommInter2 serve to produce an interaction between an abstraction and a concretion. Finally, there is a rule that allows the process of a commitment to be reverted to a pre-reduction state, if such a state exists. This rule is necessary to account for the intentional absence of structural equivalence in commitment, and is the only reference to the previous semantics in commitment. This also means that structural equivalence is not considered in the definition of commitment, which is an intentional design decision [1]. Structural equivalence is considered separately using the concept of bisimulation. It is inherent to the definition of bisimulation, and by proving that structural equivalence is a strong bisimulation, we can incorporate structural equivalence when considering the process of a commitment. Abadi and Gordon prove a lemma for this in the original paper [1].

Chapter 7

Equivalences

In the spi calculus, there are several different notions of equality on processes. These multiple notions arise from a desire to consider processes equal beyond a perfect matching of terms, names and variables. For example, if two processes differ only by the messages being sent, they still execute the same subprocesses to do so and could both be considered as the same protocol. This section will outline the different notions of equality in the spi calculus as defined by Abadi and Gordon [1].

7.1 Basic Equivalences

The spi calculus has three core types of constructs: terms, processes, and agents. While traditional equality suffices for a significant portion of our definitions on processes, Abadi and Gordon define a slightly weaker notion of equality on processes: we identify processes up to the renaming of bound names and variables. Since bound names and variables can be renamed at any time without changing the inherent semantics of the process, it does not matter if two processes are identical but have different bound elements. This notion of equality is implemented as a function ProcessEquiv, which checks that two given processes are made up of all the same constructors and terms with bound names and variables in one process substituted for those of the other process.

While we discuss basic equivalences, it is also relevant to cover the notion of equality for closed elements in the calculus. In the original paper, there is no distinction between a process and a closed process (or other such types); some definitions are only for closed elements, but there is no separate consideration of equality for closed elements. In the implementation, we have defined such notions not with a semantic

intent, but instead as a means of circumventing the strict nature of Coq's dependent typing. Since dependent types are a pair made of an element and a proof that the element possesses a certain property, the definition of equality for closed elements in the implementation simply takes the left members (the element) from the dependent pairs and considers whether they are equivalent.

Since agents are based on processes, we can convert any relation on closed processes to also apply to closed agents. Abadi and Gordon define an agent version of a closed process relation for each kind of agent. For abstractions, we consider if the internal processes of the two abstractions are related by the given relation after the bound variable of each abstraction is filled with the same arbitrary term M. For concretions, we consider whether the lists of bound names for the concretions are permutations of each other, and whether the internal processes are related by the given relation. For processes, we simply consider if they are related by the given relation. This definition is called ARelation in the implementation. Representing this definition in the implementation requires some amendments. We discuss this in further detail in Chapter 8.

7.2 Testing Equivalence

Testing equivalence is the core equivalence relation of the spi calculus. The idea for testing equivalence comes from the work of De Nicola and Hennessy [5]. It is intended as a weaker relation than strong bisimulation, which is used as a central relation in both CCS and pi calculus. The problem with strong bisimulation is that it distinguishes two processes which perform identically aside from their messages. The spi calculus requires two processes to be considered equivalent if they function the same but for the messages [1]. As such, testing equivalence works by running each process in parallel with an arbitrary third process. If both processes' interactions with the arbitrary third process are identical, then the two processes will behave identically in any context and are exchangeable [4].

The convergence predicate is the foundation for testing equivalence, and with it Abadi and Gordon define the notion of a *test*. A test consists of any closed process R and barb (name) n, and can be used to test some closed process P. If P in parallel with R satisfies the convergence predicate with n (ie. if $P \mid R$ can communicate on channel n) then P is said to have passed the test. Process testing is intended to resemble the

experiments an attacker might perform to gain an understanding of a system's behavior. With testing, Abadi and Gordon define a testing preorder $\sqsubseteq$ (ie. for $P \sqsubseteq Q$, if P passes the test then Q passes the test) and use this to define testing equivalence $\simeq$ as $P \sqsubseteq Q \wedge Q \sqsubseteq P$.

7.3 Strong Bisimilarity

To understand strong bisimilarity, it is necessary to first discuss the base concept of simulation. A simulation relation is one in which one construct "simulates" another in terms of their capabilities. For a fairly literal example, a modern computer could most likely perform all the functions of a computer from the early 1990's with significant ease. In this situation, a modern computer simulates all the capabilities of an older machine. Focusing specifically in terms of process calculi like CCS, a system A is said to simulate another system B if every state transition (called an action in CCS) in B is matched by a transition in A. In this respect, A has at least as much capability as B [11]. Robin Milner utilizes both strong and weak simulation for CCS. Strong simulation requires the simulation to be composed of named actions for state transitions while weak simulation allows a simulation to make use of the silent action τ. A bisimulation of any kind is a relation in which A simulates B in some specific manner and B simulates A via the converse of that simulation. If the simulation is represented as a set of pairs of states, the set obtained by reversing the pairs must also be a simulation. It is not sufficient for A to simulate B in one way and for the reverse to happen in some other manner. We say that two constructs are strongly bisimilar if a bisimulation relation exists that relates them. In other words, strong bisimilarity (represented as $\sim_s$) is the union of all strong bisimulations. For the spi calculus, we only consider strong bisimilarity with no reference to the weak variety.

In the spi calculus strong bisimulation is used in conjunction with the commitment relation, which is intended to act as a labelled transition semantics for the calculus. Consequently, the definition of simulation makes direct use of commitment as the mechanism for state transitions. A simulation is a relation R such that when it relates some closed processes P and Q, and P transitions (commits) to some agent A via an action, then there exists a corresponding transition from Q to some agent B via the same action and R relates A and B as well. From this definition we say that R is a strong bisimulation if both R and its converse are simulations. Finally, we define strong bisimilarity as relating two closed processes P and Q if there exists an

R which relates them and meets the criteria for being a strong bisimulation.

As stated previously, strong bisimilarity is not suitable as the core notion of equivalence for processes in the spi calculus since it distinguishes two processes that differ solely by the messages being sent. In other words, it is too strong a relation for our purposes.

7.4 Barbed Equivalence

If strong bisimilarity does not suffice due to distinguishing processes with different messages, it stands to reason that one possible solution would be to consider only the channels used and not the messages being sent while still keeping with the idea of strong bisimilarity. This gives rise to the concept of barbed simulations, bisimulations and bisimilarity. The concept was previously introduced by Milner and Sangiorgi using CCS as an example [13]. A barbed simulation draws on the previous definition of simulation: it is a relation R such that for all processes P and Q, if R relates P and Q then any barb for P is matched by Q and if P reacts to some P' (ie. $P \rightarrow P'$) then there exists some Q' such that Q reacts to Q' and R relates P' and Q' up to structural equivalence [1]. To elaborate on the latter part of the definition, P' and Q' are related by R *up to* structural equivalence if there exist some P'' and Q'' such that $P' \equiv P''$, $Q' \equiv Q''$ and R relates P'' and Q''.

Barbed bisimulation is analogous to strong bisimulation: a relation is a barbed bisimulation when both the relation and its converse are barbed simulations. Barbed bisimilarity is referred to henceforth as barbed equivalence, and two processes are barbed equivalent (represented as $\dot{\sim}$) if there exists a relation which relates them and is a barbed bisimulation. Abadi and Gordon also define a version of barbed simulation up to barbed equivalence. Where barbed simulation normally considers up to structural equivalence, this version instead considers up to barbed equivalence. Going further, there is also barbed simulation up to barbed equivalence and restriction. These simulations also have bisimulation and bisimilarity (equivalence) definitions analogous to those previously discussed, and all of these definitions are represented and can be referenced in the implementation.

7.5 Barbed Congruence

Barbed equivalence is still insufficient as a relation; there exist processes which are barbed equivalent but not strongly bisimilar or testing equivalent, and barbed equivalence is also not closed under composition. It is necessary to strengthen the relation in a manner that resembles testing equivalence: two processes are barbed congruent (represented as $\sim$) if their respective compositions with an arbitrary third process R are barbed equivalent [1]. The definition for barbed congruence is as follows:

$$P \sim Q \triangleq \forall R, (P \mid R \dot{\sim} Q \mid R) \tag{7.1}$$

Barbed congruence suffices as an equivalence relation on closed processes for the purposes of the spi calculus, and actually implies testing equivalence. As a result, demonstrating barbed congruence is sufficient to show testing equivalence.

Chapter 8

Implementation

Up to this point we have outlined how the spi calculus works, along with a minor commentary on how some of these pieces (syntax, semantics, rules, etc.) work within the implementation when it was relevant to the topic. In this chapter, we will focus entirely on the process of implementing the spi calculus. Specifically, we will cover how the calculus was implemented as well as the rationale for the wide variety of design decisions that had to be made. We begin with a brief discussion of the Coq proof system in which the implementation was written, and move from there to the many different parts of the calculus and the various challenges encountered and decisions made.

8.1 Coq

Coq is an interactive theorem prover that allows for the definition of types and functions, and provides means by which to prove properties about these constructs. It is able to check the correctness of a proof and extract a certifiably correct program from a verified proof in a number of functional programming languages. Programs and proofs are treated as interchangeable, and the system requires guaranteed termination of all functions. Proofs can be constructed using various tactics which can be applied in a manner akin to a sequential program, adding to the notion that programs and proofs are equal in the eyes of the system.

The Coq proof system has a number of features suitable for the implementation of the spi calculus. Dependent types (elements that satisfy a specific property) appear fairly often in the implementation; Coq provides a built-in definition of these dependent types. As has been mentioned, the system is incredibly strict about such types

to the point of differentiating two proofs of the same fact. The calculus has been implemented with this in mind. As a result, this strictness adds a degree of rigor to the implementation that helps guarantee accuracy of both the implementation itself and any usage by other parties.

Even without inclusion of dependent types, the word "rigor" does an admirable job describing the system as a whole; a large motivator for the use of Coq was the unyielding nature of the system. Barring the assumption of additional axioms (which the spi calculus implementation does not do) or programmer error (which was unfortunately far more common in this case), it is very difficult to make mistakes in defining or proving functions and facts in the system. Utilizing the spi calculus in a strict system such as Coq should help to guarantee the accuracy of any definitions or proofs made using it.

8.2 Methodology

There is a significant body of pre-existing work done in the Coq proof system, including some on CCS and pi calculus. This implementation was done from the ground up with no reference to these packages; the only pre-existing libraries used will be discussed here, and are included in the imports section at the beginning of the implementation file [16]. We use the List and ListNotations packages to deal with lists and more convenient syntax for those lists. The String package contains the data type of the same name which is used as the internal representation for names and variables. The Relations, Setoid, and SetoidClass packages are used for dealing with relations and the defining of them, mainly for creating instances of the Proper and Setoid classes. Finally, we use the Sorting and Permutation packages for the Permutation predicate used with the Sets package, the latter of which was created for this thesis. All other code was newly written for this thesis.

8.2.1 Basic Definitions

The absolute basics of the spi calculus implementation involve the definition of the syntax of the calculus. This includes the representations of names and variables, as well as terms, processes and agents. Names and variables are defined separately from terms because they must be considered both as themselves (like in restriction, which needs a name specifically) and generally as a possible form of a term (in most in-

stances such as for the payload of an output process). Internally, names and variables are implemented as strings of characters. As names and variables are simply text identifiers, this approach is sufficient and also allows for the definition of decidable equalities for them. This in turn allows for the use of names and variables with the many functions and predicates Coq provides for decidable types.

Terms are defined inductively, with the six kinds (Name, Variable, Pair, Zero, Successor, Encryption) each having their own constructor. Since names and variables are already defined, the term versions of these types are called TName and TVar. Other term constructors are consistent with what the original paper defines them as (for example, Pair M N where M and N are terms). Processes are defined in the same way, with the ten constructors Input, Output, Composition, Restriction, Replication, Match, Nil, Split, Case, and Decryption. Each constructor corresponds to one of the kinds of process shown in Figure 4.2. Agents are defined similarly, but abstractions and concretions are referred to specifically in the same manner as names and variables. As a result, abstractions and concretions have their own inductive type defined, and the agent constructors have the prefix A prepended to the expected name to indicate whether the element is being considered as itself, or as one possible kind of agent (ie. Abstraction is defined as its own type with constructor Abs, while the agent constructor for abstractions is called AAbs). As the third kind of agent, a process, is already defined (ie. any process is an agent), the constructor for an agent which is a process follows the same convention but is shortened to AProc for brevity.

For each construct in the spi calculus, we also define free name and free variable functions *fn* and *fv*. These functions return the set of all free names or variables of the construct. Coq does not allow for overloading of function names, so we use the aforementioned function names for the process version of the function and append t, a, Abs, and Con to the function name (ex. *fvt*) for terms, agents, abstractions and concretions, respectively. The *fna* and *fva* functions for agents leverage the pre-existing functions for its constituent kinds.

8.2.2 Substitution

Substitution is a vital component of any calculus with free variables, and in the spi calculus pertains to the substituting of a term for all instances of a given variable in a term, process, or agent. Aside from the actual substitution functions, there

are numerous lemmas that have been proven regarding substitution for dealing with occurrences of substitution in proofs. We define the functions TSub and Sub for substituting into terms and processes respectively. Although a traditional type signature for a substitution would involve a term, a target variable and a construct to substitute within, the substitution functions in the implementation are slightly more involved. We follow the same approach of substituting a term for a variable so long as the variable is not bound, but we do so not for a single term and variable but for an entire general list of term and variable pairs. Specifically, the implementation uses a list of pairs in which the variable is the first element and the term is the second. The necessity for this "queue" of substitutions stems from Coq's requirement for well-foundedness of functions.

Coq requires guaranteed termination of all functions. If it cannot automatically deduce that a function is defined in a manner that will always reach a base case, it will complain accordingly. We see this issue if the substitution function is implemented naively: when an incoming term contains a free variable matching a bound variable in the process, standard procedure is to rename that bound variable leaving the semantics of the process unchanged. To do this in a single substitution function, we must perform a variable-for-variable substitution of the new identifier before performing the original substitution. By trying to perform the original substitution on the renamed process, we are no longer calling substitution recursively on a smaller process; we are calling substitution on the result of another substitution. As a result, Coq cannot infer that the function will terminate even though a variable renaming ostensibly will not change the size of the original process. We could go through the ordeal of demonstrating to Coq that this is okay and will not affect termination, but it becomes infinitely easier to adopt a queue approach to substitution.

With our queue approach, we define a function *lookupDefault* for searching the queue of variable/term pairs and returning the first existing match based on a given variable "key". If nothing is found, we return a default value given when the lookup is called. Although it only returns the first hit in the queue, substitution queues should only ever contain the original variable/term pair and any renaming events added to the queue, so queues will never contain duplicate keys. Using this lookup, we go through the process and check any free variables within it using the lookup function, replacing them with the term if they are found and leaving them alone via the default value otherwise. To avoid variables which are bound, when recursing into a process with

a bound variable we remove all pairs in the queue that target that bound variable. The removal always takes place, so if the bound variable is not a substitution target then the removal will not change the queue.

In many proofs, often involving closedness, we encounter substitutions within a free name or variable function. To deal with occurrences of substitution in such situations, we prove lemmas indicating the general outcome of such occurrences. Given the use of a list as a substitution queue, we first prove general list versions of these lemmas and then a trivial version for a substitution of a single term. We call the latter versions of these proofs fnt_TSub, fn_Sub, fvt_TSub, and fv_Sub. The free variables of a substitution are a subset of the union of two sets: the free variables of the original term or process with all targets removed via difference (as they either did not exist or will be replaced) and all of the free variables present in the terms to be substituted in. The free name version of this lemma is simpler; since substituting for a variable cannot result in the loss of any free names, it is only a subset of the union of the free names of the original term or process and the free names in the terms to be substituted in. These lemmas should suffice in the event that substitution occurs within a free name or variable check (most commonly during proofs of closedness). It is worth noting that these lemmas consider subsets. If the goal is a set equality, Coq will not allow rewriting using these lemmas. It is necessary to first use the antisymmetry tactic defined in the set package; this will separate the set equality into two goals showing that both sides are subsets of each other.

8.2.3 Setoids

In the implementation, we define our own notions of equivalence for several types, which are known as setoids in Coq. Here we will discuss why these notions are necessary and give a general idea of how they work in the system. The first major application of setoids as mentioned above is their usage for equating closed constructs. We can circumvent strict proof equalities by using our own setoid equivalence and incorporating it throughout the implementation. Closedness is significant enough to warrant its own section however, so we will leave its discussion for elsewhere and focus on the other setoids used.

There are two other setoids in the implementation, used for sets and processes. Conventional sets do not care about ordering of elements, but since sets are implemented

as a dependent type of list we need a setoid for this. The setoid itself is simple: two sets are equivalent if their internal lists are permutations of each other. This covers the requirement that ordering of sets does not matter. The process setoid is far more involved.

In the spi calculus, Abadi and Gordon state that processes are considered equal up to renaming of bound variables. This makes sense, as bound variables can always be renamed without changing the semantics of a process. Hence, two processes that are identical save for choice of bound variable identifier should be equal. Implementing this consideration is non-trivial. One possible approach is to substitute the bound variables of one process for those of the other and check traditional equality. This is a valid approach but tedious in the context of proving other facts about it in Coq.

We opt instead to use a function that takes two processes and builds lists of corresponding bound names and variables respectively. If the processes have matching constructors that bind variables, we pair the bound identifiers together and add them to the list of name pairs or list of variable pairs according to the type being bound. When we encounter a terminal name or variable in both processes, we consider the lists of name and variable pairs to determine if the elements are equal and return an appropriate result. Of course, if at any point the two processes do not have matching constructors, the processes are not equal.

8.2.4 Closedness

Closedness of elements, as was covered briefly in the introduction, is the state of having no free variables. Closedness is important in the spi calculus, as all the relations defined by the calculus are defined solely for closed processes and agents. Unclosed elements can be created, but are generally not considered in the source paper. Hence, it is important to be able to determine closedness of these constructs and also of the type constructors and functions the constructs can be used with, to ensure that these transformations of closed constructs also result in closed constructs. To do this, every constructor of the three main syntactic constructions in the spi calculus is shown to preserve closedness when provided with closed elements. Significant consideration has been given to representing closedness in the implementation. We use dependent types to do so, in conjunction with the free variable functions for each type. We will

use processes as an example. In Coq, we define a closed process as follows:

$$Definition\, ClosedProcess : Type := \{P : Process \mid fv\, P == \emptyset\} \tag{8.1}$$

Although this makes ClosedProcess a pair of an element and the proof that it has no free variables, we do not create an element of a dependent type with the same syntax. To do this, the constructor *exist* is used. Assuming we have a process P and some proof that it is closed, we can create the closed process as follows:

$$exist\, (fun\, P => fv\, P == \emptyset)\, P\, proof \tag{8.2}$$

The first parameter of the exist constructor is a function taking the element and inserting it into the predicate, while the second and third parameters are the element and the proof that it satisfies the property. In almost all cases, the first parameter can be left as an underscore. An underscore represents a field that Coq will fill in automatically by inference, and it can do so here since only one dependent type is defined for processes resulting in one possible predicate.

Utilizing dependent types presents its own issues, however. Since two proofs of the same fact are not equal, two equal processes with differing proofs are also not equal. We must define our own notion of equality between closed elements which takes two such "pairs" and equates the left element of each pair. We define this for terms, processes, abstractions, concretions, and agents. Extracting the left and right elements of a dependent type can be done with the functions *proj1_sig* and *proj2_sig*. We alias these functions for improved readability. The former is aliased per-type (*getTerm*, *getProc*, *getAgent*, etc.) while the latter is universally aliased as *getProof*. We include type-specific aliases for *proj1_sig* for typing convenience when equating the results. Since proofs are rarely equal and are never considered as such in this implementation, a universal alias of *proj2_sig* suffices.

This notion of equivalence does not immediately solve the problem of proof inequality, however. If we attempt to apply a rule of the calculus to some related closed processes, we may encounter a situation in which an assumption and the goal have the same internal processes but different proofs. In these cases, we cannot close the goal and we cannot show that the proofs are equal. To account for this, we have to include our notion of equivalence in every rule of the calculus. Put simply, the process or agent that results from application of the rule can be substituted for some equivalent

process. Our equivalence states that the internal processes must always be equal, so this does not change the semantics of the calculus and allows us to bypass differing proofs in all aspects of closedness. Adjusting the rules in this way complicates their application slightly: Coq has a much harder time inferring obvious parameters of rule applications since it does not know which closed process we want to consider equivalent. We must almost always provide these equivalent closed processes and subsequently show that they are equivalent, but doing so should be trivial if the rule is applied properly.

8.2.5 Sets Package

In the spi calculus, the free names and free variables of a process are represented as sets with no duplicates. The Coq library contains a lightweight implementation of sets based on lists, but is not sufficient for our purposes. We have implemented a robust package for sets based on dependent types, defining sets as lists that contain no duplicates. The various operations on sets are also defined, partially utilizing the operations from the lightweight library implementation. While the set package is used often in the implementation, its contents strongly resemble that of conventional set theory so we will forgo an in-depth discussion of its development.

8.2.6 NewVar Package

Substitution of a term for a variable with a process can result in the requirement that a bound variable be renamed to avoid conflicts with incoming variables. When similar conflicts occur in the Coq system, a number is appended to the name of the identifier to create a unique new identifier, starting at zero. For example, if an identifier x exists already, a new distinct x will be renamed to $x0$, then another to $x1$, and so on. We elected to follow this simple convention when performing conflicting substitutions and so required a function that would create a unique identifier based on an old identifier and a context of other identifiers that exist already.

This is easier said than done in Coq, which relies entirely on recursion. We require a function that continually tries higher numbers until one results in a unique string, but recursion requires that the function decrease towards a base case. It was necessary to come up with a structure in which there was a decreasing parameter to recurse on. This required the definition of several functions, resulting in the function newVar which takes a string to rename and a list of strings that exist already (a context).

The function removes all numerals at the end of the target string before finding the first number it can append that creates a unique string based on the context.

8.2.7 Calculus Rules

Implementation of the rules of the calculus was fairly simple. The initial version of the rule inductives was a virtually identical reproduction of the rules as defined in the paper. Complications arose when closedness was formalized, as was discussed previously. This necessitated adjustment of the rules to incorporate a means of ignoring proofs of closedness that were essentially never equal. An initial attempt was made to show that closedness was irrelevant to the relations. In other words, after the initial relating of two closed processes, the closedness aspect could be dropped entirely to simplify proofs. This would allow for consideration of closedness with minimal adjustment to the rule inductives. However, this did not work in some cases. For example, StructTrans required a guarantee that the middle element was closed when we only knew that the first and last processes were. This was not possible to prove without additional assumptions. It was also necessary to show some consistent property regarding the free variables of processes related by reduction, but since reductions imply structural equivalence and structural equivalence is symmetric, this would require proving that the free variables of the processes in a reduction are equal. This is not the case, and so based on these issues this attempt could not succeed.

We instead opted for the method used at present: we augment the rules in the implementation such that any process in the conclusion of a rule can be swapped for an equivalent closed process. In this manner, applying rules to equivalent closed processes becomes possible in all cases, but also becomes a little more awkward. Without closedness, rules could almost always be applied with minimal specification of parameters; Coq was able to infer the parameters on its own. With the equivalences present in the current rules, it is universally necessary to provide the equivalent closed processes that the rule will result in, even if they are identical. It is also necessary to prove that the processes are equivalent, but this is trivial if the rule is correctly applied. Following an application of a calculus rule with a semicolon and *spitrivial* should address the extra subgoals without presenting them to the user. The tactic *spitrivial* is a sequence of several fail-proof tactics that attempt to resolve the trivial equivalences produced by a rule application. Using it after a semicolon applies it to all subgoals generated by the previous tactic.

We also prove several lemmas to allow for replacement of equivalent closed terms. These lemmas mostly follow the naming convention of the rules and are called RedProper, StructProper, ReactProper, BarbProper′, ConvProper, and CommProper. BarbProper′ is named as such as it clashes with the naming convention for the actual instances of Proper, which we define to allow for rewriting of equivalent closed elements within these relations. These instances use the full name of each relation (ex. StructuralProper).

8.2.8 Basic Barb

Basic barb, or BBarb, is an alternate version of the Barb relation that is logically equivalent to the original definition. This basic barb arose as a result of attempts to prove a barb assumption to be contradictory. The resulting inversion (ie. reverse case-analysis) on the assumption created an infinite loop in which the system always claimed that there may exist a structurally equivalent process and it was impossible to show otherwise. BBarb avoids this by separating the BarbStruct rule from the rest of the Barb relation. This way, we can convert from a normal Barb to the alternative Barb′ which contains only the BarbStruct rule. This version of the rule uses BBarb in its premise, so performing inversion on it will not infinitely attempt to find a structurally equivalent process. Since we separate structural equivalence in this manner, it is necessary to include new cases in BBarb for each of the rules of reduction. We do so because reduction implies structural equivalence, and we need these new rules to cover that implication without direct access to BarbStruct. All of these reduction-based rules correspond to one of the cases of reduction, but we adjust them slightly to better facilitate the purpose of a barb check. Specifically, the RedRepl rule states that a replication reduces to a single copy of the replicating process in parallel with the original replication. This is not particularly helpful as a transformation for checking barbs; we would rather consider only the process being replicated. As a result, BBarbRepl is instead defined as follows.

$$BBarbRepl\ :\ forall\, P\, \beta,\, BBarb\, P\, \beta \rightarrow BBarb\ !P\, \beta \tag{8.3}$$

To implement BBarb and Barb′, it was necessary to prove the latter equivalent to Barb. We have done so by induction, considering every case of each inductive. The proof is fairly long due to the many cases for BBarb added by the incorporation

of reduction-based rules, but since the two predicates are logically equivalent they each possess an equivalent rule or combination of rules to facilitate a matching transformation of the barb. For the forward implication ($Barb \rightarrow Barb'$) we utilize the BBarb rules corresponding to the original Barb rules, and BarbStruct$'$ for the case of BarbStruct. For the reverse implication ($Barb' \rightarrow Barb$), we prove a lemma stating that $BBarb\, P\, \beta \rightarrow Barb\, P\, \beta$. Since Barb$'$ only has one rule (which uses BBarb), we perform inversion on Barb$'$ to get a BBarb and then apply this lemma to finish the proof. Based on this proof, it is always possible to change a Barb into a Barb$'$ for the purposes of demonstrating a contradiction.

8.2.9 Equivalences

The many notions of equivalence present in the spi calculus ended up being fairly straightforward translations of their definitions in the original paper. Specifically, we refer to the equivalences discussed in chapter 7. Since these equivalences are mostly defined in terms of previous relations, they are short and require little in the way of implementation. The main equivalences are testing equivalence, strong bisimilarity, barbed equivalence, and barbed congruence. Abadi and Gordon provide many slight variations on barbed equivalence; we define these but do not use or focus on them anywhere presently. As mentioned previously, barbed equivalence is analogous to barbed bisimilarity.

Testing equivalence is based on testing preorder, which is in turn based on the definition of a test. A test checks if a process exhibits a given barb when placed in parallel with a given process (generally an arbitrary one). This definition uses the convergence predicate, while preorder uses two tests in an implication. Testing equivalence subsequently checks that both directions of a preorder pass the test. Strong bisimilarity and barbed equivalence are based on different relations, but follow a similar buildup. We first define a simulation predicate that determines if a given relation is that kind of simulation. We then define a bisimulation predicate that checks if the relation and its converse are both simulations. From this, we define the equivalence in which there must exist a bisimulation relation that relates the given processes. Strong bisimilarity requires that all transitions of two processes match, and so uses the commitment relation to check these transitions. Barbed equivalence checks only the channels being communicated on, and so uses the barb predicate in conjunction with the reaction relation. The latter serves as the transition in the same manner

that strong bisimilarity uses commitment.

Several of the equivalence definitions also use the *UpTo* relation. We say that two elements are related by R "up to" some other relation S if we can relate each of those elements to other elements via S and those other elements are related by R. Put simply, we can "rewrite" any elements related by S if the result is still related by R. Barbed simulation uses this; we consider a transition (via reaction) valid if the elements of the reaction are structurally equivalent to some other element, so we consider reaction up to structural equivalence.

Barbed congruence is the last major equivalence and is defined largely in terms of barbed equivalence. It mirrors testing equivalence in the regard that it relates two processes P and Q by checking them while in parallel with a common process R. Where testing equivalence uses the convergence predicate at its core, barbed congruence uses barbed equivalence instead. As indicated by the difference in semantics between the two relations (the former uses the initial semantics based on reaction while the latter uses the labelled transition semantics of commitment), barbed congruence looks at the barbs as transitions in the same manner as strong bisimilarity. It is a stronger relation than testing equivalence, and as a result implies it.

8.2.10 Proper and Equivalence Instances

As discussed previously, there are two main tactics in Coq for transforming the goal and assumptions: apply and rewrite. The apply tactic works based on implications, so to apply the theorem either the conclusion must appear in the goal or all of the premises must appear in the assumptions. Rewrite is simpler, allowing for in-place swapping of equal terms without the need to consider the direction of an implication. In the case of an if-and-only-if, either tactic can be used. While rewrite can always be used if one side of an equality appears in a goal or assumption, it can also be used for the aforementioned logical equivalence (iff) as well as generally for other relations. However, to use rewrite for relations other than equality and iff, it is necessary to first prove that it is possible to perform that rewriting via application of theorems. We do this in Coq by proving an instance of Proper, meaning that the given constructor or function holds with regard to the indicated equivalences. Once we prove that a context is proper, it becomes possible to use rewrite when the relation appears in that context.

For example, in the implementation we show that structural equivalence ($\equiv$) holds under composition, so if $P \equiv Q$ and $P \mid R$, we can rewrite using the structural equivalence to get $Q \mid R$ without having to apply rules and show they are equivalent. In the long run, this becomes much more convenient. We prove many different instances of Proper for different constructors and relations, both for convenience of the user and convenience proving facts necessary to complete the implementation. We do so for the setoids in the implementation as well as for the relations defined for the calculus.

For setoids, we have sets, processes, and closed versions of terms, processes and agents. Sets are a basic data type, but because we need a setoid to judge their equality we also want to be able to replace equivalent sets in the context of operations on sets. Therefore, we show that union, intersection and set difference are all proper with regard to set equality. This helps immensely with proofs of closedness, as we often have assumptions that certain constructs have no free variables (for example, $fv\, P == \emptyset$) and goals that are unions of several constructs' free variables. Rewriting equivalent sets allows us to replace most or all of the goal with empty sets, which can be simplified via set rules. Rewriting under intersection and set difference is less common, but still possible. We prove similar morphisms for processes, but for more cases. For each constructor (Output, Input, etc.), we show that we can replace equivalent processes within them. Closed processes have similar instances of Proper; these allow for quick replacement of closed processes which are identical aside from their proofs. However, we cannot prove direct instances for all closed process constructors. In the constructors that bind variables (like CInput), we provide not a closed subprocess but a process with at most the bound variable free (to allow replacing of that variable with a term as the process executes). As a result, the constructors CInput, CSplit, CCase, and CDecryption do not have Proper instances; they instead have equivalent lemmas that perform the same function but cannot be used with the rewrite tactic. These lemmas follow the same naming convention as Proper instances.

We also define several instances of Proper with regard to the relations in the calculus. As in the structural equivalence example above, there are several rules in the calculus that translate into allowing for replacement of a process with another process within a relation. StructRed is one such example: since P reducing to Q implies that P is structurally equivalent to Q, we can easily prove that any two closed processes in a reduction relation can replace each other within a structural equivalence.

Similar rules include ReactStruct, BarbStruct, ConvReact, and CommRed, which all directly allow for the replacement of closed processes which are related by structural equivalence, reaction and reduction, respectively.

8.2.11 Agent Semantics and ARelation

When discussing agent semantics, Abadi and Gordon define composition and restriction of agents as well as interactions. The latter works analogously to ReactInter but with an abstraction and a concretion, while the others are defined using their process counterparts. The key difference with these operations is that they have side conditions in place to avoid conflicts with bound names and variables, which abstractions and concretions almost always have. The authors continue from here by implicitly assuming such side conditions are always met. We opt to take a different approach in the implementation.

Since these side conditions are intended to avoid naming conflicts for bound identifiers, we can programmatically avoid them through the renaming of those bound identifiers. In agent restriction, we add the restricted name to the internal process except in the case of a concretion, where we must consider if it occurs free in the term. If it does not, we proceed as in the other cases. If it does, we instead add the name to the list of bound names inherent to the concretion. Essentially, we only add the element to the list if it matters to do so. Normally agent restriction has the side condition that the incoming restriction cannot be present in the list already. We ignore this in the implementation, as multiple instances of such a name in the list should not affect the concretion.

For agent composition, the bound identifiers in the abstraction or concretion cannot occur free in the incoming process. This makes sense, as they would become bound and change the meaning of the process. We avoid this side condition by checking for conflicting identifiers between those bound in the agent and those free in the process, and based on this list of conflicts create new identifiers and rename the bound instances in the agent. We can then compose the two elements without conflict.

For interaction, the bound list of names of the concretion is repurposed to cover the entire resulting process. As such, the abstraction cannot have any of these names occurring free within it. Similarly to with agent composition, we check for these con-

flicts between the concretion's list and the free names of the abstraction and generate new names accordingly. With these considerations, no explicit side conditions are necessary for agent semantics.

Since agents are a semantic adjustment of processes, it makes sense that any relation on processes can also work on agents. Abadi and Gordon define a "function" that converts any relation on closed processes into a relation on closed agents. We call this function ARelation in the implementation. For a given relation, if the two agents are abstractions, they are related if for any arbitrary closed term, it can be substituted in for the abstracted variable on both processes and the resulting processes are related. For concretions, they are related if their restricted lists are permutations of one another, the terms are the same, and the processes are related. For processes, the two must simply be related.

This is how the function is defined in the original paper [1]. However, there is a minor oversight in how it works. Abadi and Gordon define process equivalence up to renaming of bound names and variables. All of the names in the list of a concretion are bound, which means that if process equivalence as a relation is provided to ARelation, it will not accurately determine that two concretions with differing bound names (in the same positions) are equivalent. As a result, the definition of ARelation in the implementation is a bit more involved. The abstraction and process cases remain as they were originally defined. For concretions, we know that the terms must be equivalent up to renaming of bound names (and variables, but at this point no variables can be bound). Thus, we define a helper function that takes in two terms and performs a matching on them, building up a list of name pairs where they occur in the same position within the terms. From this, we can determine what names should be equivalent within the concretions' lists. If the terms are not structurally identical, the concretions cannot have equivalent terms and the relation returns False. Otherwise, we perform a renaming within the restriction lists and processes based on the result of the helper function before checking whether the processes are related by the given closed process relation. This allows ARelation to take into account renaming of bound names in concretions when checking relations such as process equivalence.

8.2.12 Proofs/Propositions

There are close to forty propositions and lemmas included in the original paper [1]. These facts range from relationships between equivalences in the calculus (such as structural equivalence implying testing equivalence) to important lemmas like structural equivalence being a strong bisimulation. A significant portion of the appendix in the paper is devoted to proving these propositions [1]; we include only a small selection of these in the implementation at present according to necessity. It is important to note that two versions of the original paper exist; there is a technical report with a longer appendix including more involved proof discussion [2]. We mention this because the numbering of propositions and lemmas differs slightly between these versions for some proofs. For example, the proof that structural equivalence is a strong bisimulation is Lemma 22 in the original paper and Lemma 25 in the technical report.

While the proofs of some of these propositions are covered by Abadi and Gordon, these descriptions are not always sufficient to make proving them in the implementation trivial. Lemma 22 is quite involved, requiring two-way analysis of every single structural equivalence rule and proofs of structural equivalence rules that work for agents rather than processes. Work was done on this proof prior to the inclusion of closedness and process equivalence but has not been updated with the rest of the implementation; it requires consideration of a massive number of cases. Abadi and Gordon discuss some but not all of the cases of this proof and do not address anywhere the concept of structural equivalence rules for agents [2]. This and other extra requirements had to be inferred while attempting to prove the propositions. As a result, we opted to focus on the functionality of the calculus itself rather than reprove all of these propositions in the implementation, many of which would likely also require proof by hand first.

8.2.13 Notations

Internally, definitions like the process constructors in Coq are represented in a straightforward manner with the name of the constructor followed by its parameters. While in isolated cases this is fine, processes in the spi calculus can become rather unwieldy with even just three constructors chained together. Abadi and Gordon provide a simple syntax based on CCS and pi calculus which came before, and Coq supports such notation replacements as long as it does not interfere with certain reserved symbols that the system uses. These reserved symbols include various kinds of parentheses,

and keywords like *if*, *then*, *else*, and *match*. While notations can often be declared using such symbols, it causes havoc when the system attempts to parse further definitions. We have attempted to implement notation that is faithful to the original syntax of the spi calculus while also avoiding conflict with Coq's pre-existing notation.

The best way to avoid such conflicts was to make use of special unicode symbols wherever the syntax clashed with that of Coq. For example, a composition process uses a vertical bar, commonly available via the | symbol. However, this symbol is used frequently by Coq, so we need an alternative. A quick fix comes in the form of combining symbols to form a single operator, such as : | : (colon-bar-colon). This is convenient for entering protocols into the system but less so for reading them during proofs. We choose to use a unicode vertical bar which is visually similar to the common | symbol, and make similar choices for other syntax. While this does slightly complicate entry of protocols into the system due to the absence of these symbols on a conventional keyboard, modelling the protocol is only a small initial portion of the work. The focus should be on the proving of properties about the protocol. Hence, we have opted to focus on using symbols that enhance readability. This may make the initial modelling of the protocols more tedious, but should help when proofs begin to accrue many assumptions (which they tend to do very quickly).

We will now outline the notations used for each construct in the implementation, starting with closed processes. We opt to use calculus-accurate notation on closed processes before general processes since notations cannot be used for multiple definitions and closed processes are far more prevalent in the spi calculus. Note that this is a visual representation of the syntax and not an exact copy of the unicode symbols used. If the notation looks identical to the original syntax, then it is reasonably approximated via said symbols.

There are a few immediately noticeable differences from spi calculus syntax. We do not use the symbol ν for restriction, since the word *new* is just as readable and easier to type. The *new names l in P* notation (and MultiRestriction as a definition) is not intended for programmer use, but is included as it will appear during proofs. The case notation uses the word *Check* rather than *Case* to avoid conflict with the constructor of the same name. Decryption is depicted verbally; Abadi and Gordon denote it as an offshoot of the integer case syntax. However, this both conflicts with the case

Table 8.1: Closed Process Notation

Constructor	Notation
COutput M N P	$M\langle N\rangle.P$
CInput M x P	$M(x).P$
CComposition M N	$M \mid N$
CRestriction n P	$new\, n\, in\, P$
CMultiRestriction l P	$new\, names\, l\, in\, P$
CReplication P	$!P$
CNil	0
CMatch M N P	$:[M\, is\, N]:\, P$
CSplit M x y P	$Let\,(x,\, y)\, be\, M\, in\, P$
CCase M P x Q	$Check\, M\, of\, zero:\, P\, suc\, x:\, Q$
CDecryption M N x P	$Decrypt\, M\, using\, N\, into\, x\, in\, P$

notation and utilizes a subscript notation unavailable to us in Coq.

For the relations between these closed processes, we have the similar issue of an absence of subscripts and superscripts. Table 8.2 contains the notations for the corresponding relations from Table 4.1. We represent the arrow for reaction and commitment as three dashes followed by a right angle bracket; this prevents it from conflicting with any similar arrow notation used by Coq. Commitment with a process P, agent A and action α is denoted as $P\ ---{>}\ A\, via\, \alpha$. We use arrows composed of the tilde symbol for barb and convergence, as they are larger and more visible than unicode downward-facing arrows. All other symbols are similar to their originals.

Table 8.2: Relation Notations

Relation	Symbol
Reduction	$>$
Structural Equivalence	$\equiv$
Reaction	$---{>}$
Barb	$\sim{>}$
Convergence	$\sim\sim{>}$
Testing Equivalence	$\sim{=}$
Commitment	$---{>}\ via\, \alpha$
Strong Bisimilarity	$\sim s$
Barbed Equivalence	$\cdot\sim$
Barbed Congruence	$\sim$

Agents have notation which is virtually identical to how they are defined in the original paper. These notations are listed in Table 8.3. We focus on closed elements here for the same reasons as with processes above. The main difference comes in the distinction between abstractions and concretions on their own and in the context of agents. Abstractions and concretions have notation that matches the paper, but the agent versions of these constructors need a notation as well. We choose to wrap the internal element (an abstraction, concretion, or process) in curly braces with an identifier to the right to indicate what kind of agent the element is. The encapsulation does not matter much, since the internal element will usually be obvious based on its notation. We also define notations for operations on agents, including composition, restriction, and interaction. Abadi and Gordon define general infix symbols for composition and interaction, but the results differ based on what side each operand is on. We denote these operations by appending L or R to the symbol.

Table 8.3: Closed Agent Notations

Constructor	Notation
CAbs x P	$(x)P$
CCon l M P	$(\nu\, l)\langle M\rangle\, P$
CAAbs F	$\{F\}Abs$
CACon C	$\{C\}Con$
CAProc P	$\{P\}Proc$
CACompositionL P A	$P \mid L\ A$
CACompositionR A P	$A \mid R\ P$
CARestriction n A	$ARes\ n\ in\ A$
CInteractionL F C	$F\ @L\ C$
CInteractionR C F	$C\ @R\ F$

We use a similar approach for terms. However, we do not have notations for all terms. As discussed previously, we cannot have a closed constructor for a TVar. We introduce a notation for it along the same lines as for agents, but it appears less so than others. Names have the same convention with arbitrary encapsulation that is mostly irrelevant. We use N and V in these notations to denote whether the internal identifier is a name or variable, as opposed to agents where we use A in all cases. Pairs have the expected notation, and we denote encryptions verbally as decryptions were due to the lack of subscript notation. Since the symbol 0 is used for the nil process, we let the constructor Zero (or CZero, the closed version) suffice for notation; the same goes for the constructor Suc and its closed counterpart.

Table 8.4: Closed Term Notations

Constructor	Notation
CTName n	$\{n\}N$
TVar x	$\{x\}V$
CPair M N	(M, N)
CEncryption M N	$Encrypt\ M\ using\ N$

Chapter 9

Example

In this chapter, we will provide and discuss an example of the spi calculus as it is represented in the implementation and a proof about that example. A note regarding channels: while Abadi and Gordon retain the concept of names and co-names from pi calculus for denoting receiving and sending on channels, the implementation does not distinguish between them. Name or co-name can still be inferred based on the context. The discussion below maintains usage of co-name notation for clarity (ex. $\overline{c}$) but the system has no such notation.

The simplest possible example of a proper protocol in the spi calculus is the sending of a message on a channel; Abadi and Gordon use this as their first example [1]. If we have a channel c and some message M to send, the protocol would look like this:

$$(\nu\, c)\,(\overline{c}\langle M\rangle.0 \mid c(x).0) \tag{9.1}$$

From this process, a reaction can occur between the left and right subprocesses to produce the process $0 \mid 0$. While this is a valid example, we would prefer to demonstrate the passing of the message as well. Currently, this protocol sends M and does nothing with it. We can create a more realistic protocol using an arbitrary process containing just the variable x free. Such a process is identical in nature to an abstraction, a kind of agent. We can instantiate an abstraction's free variable by providing a term, which makes the abstraction into a process. Instantiation is analogous to substitution, but the variable to target is that of the abstraction. With an abstraction F, we can create the new protocol:

$$(\nu\, c)\,(\overline{c}\langle M\rangle.0 \mid c(x).F(x)) \tag{9.2}$$

With this protocol, we can send M on channel c, at which point it will be received by the right subprocess and replace all further instances of x. At the end of the protocol, the result will be the process $0 \mid F(M)$. This is the first sample protocol given by Abadi and Gordon. For further simplicity the implementation proof specializes the arbitrary abstraction F to become $(x)0$. This has no bearing on the semantics of the proof, but allows the system to trivially solve some equivalences that arise in the proof. We still demonstrate authenticity via two protocols, one which must send M and another which already has M; they simply do nothing with that message once it is instantiated from abstraction to process. We will define the processes parametrically with an arbitrary F, then specify when we begin the proof.

To represent this protocol in the implementation, we utilize closed process constructors chained together. We define the left and right subprocesses separately and combine them together via a "bootstrap" process. The sending process A is defined as follows, and requires a closed term as the message parameter since the process itself must remain closed.

$$Definition\, A\, (M : ClosedTerm) := COutput\ (CTName\ \text{“}c\text{”})\ M\ CNil \tag{9.3}$$

We could also use notation, but this gives a better idea of the internal representation of a process. The receiving process B is slightly harder to represent: the input process binds the variable x below it, but when defining a process used closed process constructors, we cannot insert variables on their own as they are not, on their own, closed. The constructor CTName serves this purpose for a name since names are closed on their own, but we cannot do this for variables. Hence, we cannot use a closed process constructor like CInput to build B. We must instead define a normal process B' and show that it is closed to create B. The unclosed process is defined as follows (we omit the parameters due to length; B' takes a closed abstraction F).

$$Definition\, B' := Input\ (TName\ \text{“}c\text{”})\ \text{“}x\text{”}\ (Instantiate\, (getAbs\, F)\, (TVar\ \text{“}x\text{”})) \tag{9.4}$$

We use *getAbs* to get just the abstraction and ignore the proof of closedness. We then state and prove $B'Closed$, the fact that B' has no free variables. This is trivial for a specific process that is actually closed and only requires simplification of the free variable check and use of lemma *fv_Sub* to consider the free variables of the instantiation. From this process and this proof, we can make the closed process B

using the *exist* constructor.

$$Definition\, B\, (F : ClosedAbstraction) := exist\ \_\ (B'\, F)\ (B'Closed\, F) \tag{9.5}$$

Finally, we can define the protocol *Basic* using A and B. We compose the two processes and restrict the name c within the composition for use as a communication channel. We also include a closed term M as a parameter to the whole protocol and pass it into A. Once again we omit the parameters due to length: *Basic* takes a closed abstraction F and a closed term M.

$$Definition\, Basic := CRestriction\ \text{“}c\text{”}\ (CComposition\ (A\, M)\ (B\, F)). \tag{9.6}$$

With this, the protocol has been fully modelled and can be used in any relation on closed processes, such as testing equivalence.

While this protocol does serve as a good example of how to model protocols in the calculus, the real purpose of the system is to prove properties about protocols. We will show the authenticity property for this protocol as a demonstration of a simple proof in the implementation. Authenticity is the guarantee that a message sent via a protocol is from the expected sender, rather than an attacker substituting their own message. To prove this, we require a “specification” of the protocol. We will forgo the incremental construction of this specification and address only the differences from the original.

The specification of the protocol is a version in which everything happens in the same way, but the message is already known to the receiving process. This makes the sending somewhat redundant, but the intention is to show that this protocol is testing equivalent to the original. In this way, we demonstrate that it is impossible for an attacker to cause the protocol to use any message other than the one sent by A. We adjust B accordingly to get the full specification, which only changes the parameter of F to be M prior to the reaction.

$$(\nu\, c)\ (\overline{c}\langle M\rangle.0 \mid c(x).F(M)) \tag{9.7}$$

We call this protocol $Basic_{spec}$ and represent it analogously to *Basic*, but with M as the parameter for F. We must create an unclosed version of B_{spec} and prove it to be closed before constructing a closed process in the same manner as with its

original counterpart. With these two protocols, we can now begin to show that $Basic\,F \simeq Basic_{spec}\,F$, where $F = (x)0$. We will leave F alone for syntactic simplicity until it makes sense to consider $(x)0$.

We opt to emulate the proof given by Abadi and Gordon. They prove these protocols to be testing equivalent by instead proving the stronger property of strong bisimilarity. Colloquially, testing equivalence demonstrates a matching of all currently possible barbs as well as all barbs possible after any reactions. Strong bisimilarity demonstrates a matching of any and all transitions that two constructs can perform and the states they reach. These simple protocols can make only one transition via the silent action τ (a reaction between the input and output processes) and they both result in the same new state $0 \mid F(M)$. Hence, we should be able to perform an exhaustive analysis of their transitions and show that they all match. Strong bisimilarity utilizes the commitment relation for transitions, so we will be using it in this proof.

To show two processes are strongly bisimilar, it is necessary to show that they are both members of a strong bisimulation. In a strong bisimulation, a relation and its converse must possess pairs of states that advance via the same action and reach states that also simulate each other. In this case, the initial states of *Basic* and $Basic_{spec}$ simulate each other via τ, and their next states are both identical and make no transitions so they simulate each other as well. We begin the proof by breaking down the definition of strong bisimilarity accordingly: the notation $\sim_s$ unfolds to become the following. As shown previously, *Basic* and $Basic_{spec}$ have parameters F (specifically $(x)0$ in this case) and M; we will omit their inclusion when discussing proof terms to avoid cluttering them, but they are still present in the implementation. When we break down *Basic*, we will include these parameters again.

$$exists\,R,\ StrongBisimulation\,R \wedge R\ Basic\ Basic_{spec} \tag{9.8}$$

We must provide a relation and show it is a strong bisimulation. We do so using a function *MakeRelation* which takes in a list of pairs of processes and produces a relation checking whether the two inputs match any of these pairs. We provide the two pairs (*Basic*, $Basic_{spec}$) and $(0 \mid F(M),\ 0 \mid F(M))$ (we will refer to this relation as R for simplicity) and proceed to *split* the goal into the left and right subgoals of the logical and. The right subgoal is trivial; we will spend most of our time showing

that R is a strong bisimulation.

Since a strong bisimulation is defined as a relation R and its converse being simulations, we can again use *split* to separate the goal into these two subgoals. We can then break down the definition of simulation based on the following definition where P and Q are closed processes, Y and Z are closed agents (to avoid confusion with processes A and B from *Basic*) and α is an action. We use double arrows for implications to avoid confusion with commitment arrows here, although Coq utilizes single arrows for its syntax.

$$\forall\, P\, Q\, Y\, \alpha,\, R\, P\, Q \implies P \xrightarrow{\alpha} Y \implies \exists\, Z,\, Q \xrightarrow{\alpha} Z \wedge \overline{R}\, Y\ Z \tag{9.9}$$

The symbol $\overline{R}$ indicates a version of the closed process relation R that works equivalently on closed agents. Since our relation only deals with processes rather than abstractions or concretions, this does not add any complexity to the proof. We first use *intros* to move our two premises (some P and Q are related by R, and that P commits to some Y via α) into our assumptions. With the assumption that P and Q are related by R, we can begin considering the possible cases. Since R is just two pairs of specific related processes, we can *destruct* this assumption into the two possibilities: either $P = Basic \wedge Q = Basic_{spec}$ or $P = 0 \mid F(M) \wedge Q = 0 \mid F(M)$. There is a third possibility (P and Q are not related), but this results in an assumption of $False$, whicih is easily discarded by the tactic *contradiction*. The underlying definition of this assumption is a logical or, so each of these possibilities is a separate subgoal. It will provide the former case first, but in both situations we can destruct this logical and to get the individual facts (which will not produce new cases, as both are true). We then substitute them into the appropriate places in the assumption $P \xrightarrow{\alpha} Y$ and our goal $\exists\, Z,\, Q \xrightarrow{\alpha} Z \wedge \overline{R}\, Y\ Z$. We get the new assumption $Basic \xrightarrow{\alpha} Y$ and the new goal:

$$\exists\, Z,\, Basic_{spec} \xrightarrow{\alpha} Z \wedge \overline{R}\, Y\ Z \tag{9.10}$$

To progress from here, we must begin considering the possible commitments of our assumption. We do this by performing an *inversion*. From this the system will infer what each component must be to satisfy each rule of commitment. Due to the setoid usage in our rule inductives, Coq generally cannot discard absurd possibilities immediately; it falls to us to do so. Since our protocol begins with a restriction, the only possible case is CommRes. We can discount all other possibilities using further inversions, as in all the other cases we have an absurd setoid equality such as an output

process being equal to a restriction process. The exception to this is CommRed, in which case we must rewrite the given equality in the reduction assumption and then perform another inversion in which every single possible case is absurd. Thankfully, this can be accomplished easily using a blanket inversion after a semicolon, applying the tactic to every subgoal's contradictory hypothesis. We will have to use similar tactics on further inversions but will not cover these in as much detail since they all follow the same general approach.

Now that we have determined that our commitment was derived from CommRes, we have a new commitment based on the inner constructs of the original commitment. Specifically, we know that $A \mid B \xrightarrow{\alpha} Y'$ and $\alpha \neq c$ since c is restricted. Y' is inferred from the original Y, since CommRes tells us that $Y = (\nu\, m)\, Y'$. The name m is arbitrary and produced by the inversion but will not affect the proof, since we consider processes equivalent up to renaming of bound identifiers such as these. From here, we need to consider what Y' is via another inversion. Since the left element of the commitment is a composition, the only rules that apply are CommInter1, CommInter2, CommPar1, and CommPar2. CommRed does not apply since no reductions deal with an output/input composition. Only one of these four cases actually applies: CommInter2 deals with an interaction of an input and output composition where the output is the left process.

From here we have four major cases to prove or disprove, and we begin each one with a common series of steps. Initially, the previous inversion provided us the following system assumption in each of these cases:

$$new\ \text{“}c\text{”}\ in\ (A\, M \mid B\, F) == new\, m\, in\, (P1 \mid Q0) \tag{9.11}$$

The obvious conclusion here is that $A\, M == P1$ and $B\, F == Q0$. Unfortunately, it is not that simple. Since these processes are under a name binding, they are instead equivalent up to considering that "c" and m are equal. This makes things slightly more complex in the implementation. When dealing with trivial equalities such as this in Coq, we normally use inversion to conclude each piece on either side is equal to its counterpart. Coq cannot do this here due to how process equivalence is defined (via a function). We instead prove a lemma for each process constructor that breaks such an equivalence down with respect to the process equivalence function. Applying the corresponding lemma *RestrictionInversion* gives us the fact that these composi-

tions are equivalent with respect to "c" $== m$. From here we can break it apart into the left and right pieces of the composition, and simplify those process equivalences as much as possible. This requires some work with *destruct* and a few possible results that are handled with *try*, a keyword which does not stop evaluation of a tactic upon failure. This hassle is necessary since in cases where we have assumptions like $Output\ M\ N\ P == Q$, the system simplifies this to a case matching where Q is either an output (which makes sense) or not an output, which produces the assumption *False*. We have to manually dispose of all the contradictory possibilities for Q. We do this here for both sides of the composition equivalence, in all cases of the commitment. Once they have been fully broken down into the various trivial equalities, we can proceed to reason about each case.

We disprove CommInter1 since it requires that the left process is an input, and this produces an absurd setoid equality between an input and an output. CommPar1 and CommPar2 require a little more work, but the contradiction arises from the CommPar rules cutting the two process composition into a single input or output process. The only applicable cases for these are CommIn or CommOut, but these rules require that the transition occurs via the channel of the process. The channel of these processes is c, so α must be c. Since we have the fact that $\alpha \neq c$ from our first inversion, these cases contradict our assumptions. We are left with CommInter2, so we have the assumptions $A \xrightarrow{\overline{m}} C$ and $B \xrightarrow{m} F'$ as well as the complete interaction $A \mid B \xrightarrow{\tau} C \mathbin{@} F'$. We denote the abstraction as F' to differentiate it from our initial abstraction F, and can now conclude that the action α must be the silent action τ.

The name m is arbitrary per the inversion. We perform a further inversion on these assumptions to flesh out what C and F' must be based on CommOut, CommIn, A, and B. Since the process A sends term M on channel c and then does nothing, the concretion must be $(\nu)\,\langle M\rangle\,0$ and occurs via channel $\overline{c}$. Similarly, we conclude that the abstraction must be $(x)\,F(x)$ on the same channel.

Now, we must provide a matching transition agent Z so that we can apply CommInter2 and deal with the resulting subgoals. Our goal is now the following:

$$\exists\, Z,\ Basic_{spec} \xrightarrow{\tau} Z \wedge \overline{R}\ (\nu\, c)\,(C \mathbin{@} F')\ Z \tag{9.12}$$

Though we know what C and F' must be, we leave them "folded" here for better visual clarity until it is important to unfold them (in the implementation, they appear in full). Our choice of B is almost obvious: we want to use the result of the interaction $0 \mid F(M)$ wrapped in a restriction of c, since the sending occurs within this context. However, the difference between B and B_{spec} comes into play at this point. To be able to apply CommInter2 after this, we need a commitment that uses an interaction. Thus, we need to frame $0 \mid F(M)$ as an equivalent interaction. The obvious choice is $C \mathbin{@} F'$, but F' has x as the parameter of the instantiation of F. $Basic_{spec}$ already has M filled in for this x, so we have to provide our own abstraction with M already present. We choose the concretion $(\nu)\langle M\rangle 0$ and the abstraction $(x)(Instantiate\, F\, M)$. In this example F is the abstraction $(x)0$, so we fill all instances of x with M (in other words, we do nothing). Therefore, we now want to show the following two subgoals. To differentiate the two very similar agents in the second subgoal, we encase them in square brackets.

$$Basic_{spec} \xrightarrow{\tau} (\nu\, c)\, ((\nu)\langle M\rangle 0 \mathbin{@} (x)\, 0) \tag{9.13}$$

$$\overline{R}\, [(\nu\, c)\, (C \mathbin{@} F')]\, [(\nu\, c)\, ((\nu)\langle M\rangle 0 \mathbin{@} (x)\, 0)] \tag{9.14}$$

Once again the latter subgoal requires a trivial examination of what R relates; most of our effort goes into demonstrating the former. Using $(x)0$ simplifies things greatly here, as we provide an abstraction containing the nil process which is pre-instantiated with the message M. Instantiating the nil process will always result in the nil process, so the instantiation is technically irrelevant. However, since it would be necessary for a general abstraction, we structure it in this manner anyway. Now, we show the commitment via the calculus rules.

We begin by applying CommRes; this allows us to drop the restriction from both sides of the commitment as long as the restricted name is not the same as the commitment action. We know $\alpha \neq c$ from our work with the assumptions (technically $\tau \neq c$ since $\alpha = \tau$), so the side condition is fulfilled; we finish the subgoal $\tau <> ActName$ "c" using the tactic *discriminate*, as the two actions are of different base constructors. We now have the following goal:

$$A \mid B_{spec} \xrightarrow{\tau} (\nu)\langle M\rangle 0 \mathbin{@} (x)\, 0 \tag{9.15}$$

This fits the rule CommInter2, so we apply it and instead consider the two individual commits based on CommOut and CommIn. Our two subgoals are $A \xrightarrow{\bar{c}} (\nu)\,\langle M\rangle\, 0$ and $B_{spec} \xrightarrow{c} (x)\,(x)\,0$. These goals fit the commitment axioms CommOut and CommIn, so we can close these subgoals by applying these rules. Specifically, the channels, terms and processes must match on either side.

Now we show that our provided agent Z, the "specialized" result with M already sent is related by R to the non-specialized version which must send M. Per our relation R, we stated that the process $0 \mid F(M)$ was related to itself. If we resolve these interactions via simplification, we get a composition with a nil process on the left and F instantiated with M on the right whether it was actually important to send M or not. Of course, with the nil abstraction here, the interaction will simplify to a composition of nil processes. Hence, the two interactions have the same result which is related by R. Having closed this subgoal (and having dealt with every other contradictory case of commitment), we have finished showing that this pair of states is a simulation. The system now asks us to show simulation for the pair $(0 \mid F(M), 0 \mid F(M))$, but this comes with the assumption that $(\nu\, c)\,(0 \mid F(M)) \xrightarrow{\alpha} A$. The composition of nil and a "nil abstraction" cannot make any meaningful commitments, so this assumption is contradictory. We show this using inversion on all of the absurd cases that appear. If F were arbitrary it is possible that it may produce another composition of input and output processes, but we would assume it does not for the purposes of this example.

We have finished showing that our relation is a simulation, and must now show the converse is as well. This is largely identical to the example up to here, but with $Basic_{spec}$ and its commitments as our assumptions. It would normally necessitate a different choice of agent B when demonstrating the commitment: instead of the pre-filled abstraction $(x)\,(F(M))$ in our assumed interaction, we would provide an interaction with an abstraction $(x)\,(F(x))$ that has not yet received the message. The specific case of a nil abstraction does not require this (since both versions simplify to the same result immediately), but again we format it this way as if F was arbitrary. Specifically, we provide an instantiation of the nil process that uses the variable x. Everything else is identical in terms of assumptions and goals, and should match up to what has been done already. Now that the the converse simulation has been shown, we finally have to show that the two processes $Basic$ and $Basic_{spec}$ are related by our relation R. This is mostly trivial, and entirely so in our specific case. With nil as

the internal process for the abstraction, the system can resolve these subgoals down to a primitive $True$ immediately. With an arbitrary abstraction, it is necessary to do some work to show equivalence of the original processes of the protocol and the new arbitrary ones produced by inversion. This is a primary reason for the use of nil in the abstraction, as it greatly simplifies the example proof in a manner that does not relate to actual application of the calculus.

At this point, we have proven the fact that $Basic \sim_s Basic_{spec}$. We wanted to show that $Basic \simeq Basic_{spec}$. In other words, every barb (external communication) and reaction (internal communication) of the two protocols is the same while in parallel with a arbitrary "observer" process. We have done so by demonstrating a stronger property: every single action taken by $Basic$ is simulated by $Basic_{spec}$ and vice versa. By the definition of bisimilarity, this means the two protocols are indistinguishable from the outside [11]. It follows that if $Basic$ and $Basic_{spec}$ were placed in parallel with any arbitrary process, the process would not be able to detect a difference in behavior between the two protocols. This is the intent of testing equivalence, so strong bisimilarity is sufficient to show testing equivalence [1]. With this, we have shown the authenticity property for a very simple protocol, and have the guarantee that an attacker cannot insert their own message for whatever message is sent by A.

Chapter 10

Conclusion

At the outset of this project, there were two main goals. We wanted to implement the spi calculus as defined by Abadi and Gordon in a proof system, and we wanted to use it to model and prove properties about cryptographic protocols. At this point we consider both of these objectives to be met, with some caveats. We will discuss these points in detail below, but the main focus is on the depth of the spi calculus and the definitions necessary to facilitate its full functionality in the Coq proof system. We can see this depth and complexity in aspects like our process and closed process setoids, in which cases traditional equality as the system viewed it was not sufficient for this constructs. As such, it necessitated not only definition of custom notions of equivalence for these constructs, but also the adjustment of the entire work up to that point to accommodate these changes. Closed process equivalence required the inclusion of the setoid in all rules of the calculus. Process equivalence complicated the use of the *inversion* tactic on previously equal processes, raising the possibility that two identifiers were not exactly equal but functionally equal. The constant iterative nature of our design and development process, requiring consistent re-evaluation of the existing work, meant that some aspects of the calculus were not as easy to implement as they first seemed. Given our current understanding of the spi calculus and its intricacies, much more efficient approaches were possible. Of course, we say this with the benefit of hindsight.

10.1 Future Work

There are a large number of possible considerations for future work regarding this implementation. From refinements to the base aspects of the calculus to some less-used notions introduced later in the original paper, there are many ways that this

work can be continued. We will address these possibilities in order of importance; note that any discussion of the volume of work required is purely speculative.

Proof Flow

One of the most evident hurdles present in the calculus at time of writing is the immense amount of visual clutter that often occurs due to the size of the average process definition (even with notation) coupled with the presence of sigma types (ie. closed constructs) that often need to be destructed into larger definitions that also display the unnecessary proof component. Use of equivalences beyond basic equality also complicates the use of inversion, preventing the system from automatically discarding most contradictory cases. This requires the user to demonstrate the contradictions on their own, necessitating the presence of a bulk tactic including at least one inversion per contradictory subgoal. These problems are only tangentially related to the actual goal of proving properties about cryptographic protocols. One major goal for the future would be to put dedicated effort into improving both the visual and tactical aspects of these proofs to improve the user experience.

The best approach at present is to do a rough proof in the calculus on paper, followed by a translation of that proof into the system. In this sense it is possible to perform verification of proofs in the spi calculus which was one of our goals, but it would be immensely beneficial to be able to better understand and work with the proofs in the system. Such improvements would likely come in the form of tactics and lemmas covering common actions in the implementation, but this would require more time working with proofs of properties about actual protocols.

Minor Functionality

There are many minor definitions and lemmas that can be added for improved functionality rather than for convenience while proving properties. At several points, Abadi and Gordon use notation indicating restriction of a list of names. It follows from this that any rules regarding restriction would also apply to a general list of restrictions. Implementing this in the system required more rigor in the form of the MultiRestriction definition allowing for an arbitrary list of names to be translated into a restriction per name, added onto a given process. This in turn necessitated the proving of lemmas that emulated the single restriction rules used in structural equivalence, but for a MultiRestriction. We have proven equivalent rules for StructDrop,

StructSwitch and StructExtrusion, but similar lemmas could be necessary depending on the appearance of MultiRestriction in actual usage of the implementation. In this case and regarding others in the implementation such as the lemmas regarding free variables of a substitution, there are many minor improvements to be made that did not come up in the work up to this point.

Propositions and Lemmas

Abadi and Gordon define and prove approximately forty propositions and lemmas regarding the spi calculus, many of which have multiple parts. As discussed, we have only implemented some of the early propositions. Many more remain, and would be helpful during practical use of the implementation. The proofs of these propositions are covered briefly in the original paper, but are not comprehensive and so require some effort to reproduce. It was not our goal to re-prove every property in the paper, and such a task would likely take significant effort. Hence, we leave it as a future goal for the implementation.

The Underpinning Relation

Prior to the discussion of proofs using the spi calculus (which comprises most of the document), Abadi and Gordon introduce the underpinning relation. This relation is intended to better represent the knowledge of an attacker trying to break a cipher [1]. While a preliminary attempt was made to implement this relation, it is not used in the base calculus and only arises in proofs of propositions and lemmas from the paper. Since we have focused only on what was needed from these, no proof work was done regarding the underpinning relation. As a result, fully implementing the underpinning relation and doing proofs with it are future goals for the implementation.

Proposed Additions

Abadi and Gordon discuss some possible additions to the spi calculus throughout the original paper. One such addition is a "mismatch" construct, checking that two terms are not the same. Similar ideas are proposed for checking that a term is not a name, not a number, not a pair, or not encrypted with a particular key [1]. Chapter 7 of the paper also covers extensions to the calculus that go beyond the initial scope of shared-key cryptography to make use of public-key, hashing, and digital signature cryptography. They also propose the possibility of compatibility with Diffie-Hellman techniques and secret sharing. Each of these approaches adds

more primitives to the calculus, which in turn complicates any proof by induction on terms and processes. Thus, while these extensions can certainly be added to the implementation, probably as separate versions, they will likely require a significant amount of work to incorporate.

10.2 Closing Remarks

While the original intent of this work was only to implement the spi calculus in a proof system and use it, it is now clear that these tasks are more nuanced than they first appeared. There is still much to be done to make the implementation as efficient and as effective as possible for the average user. However, the implementation at present stands as both a proof of concept for future improvements as well as a lesson in the amount of work necessary to accurately recreate some fairly simple on-paper notions in a proof system such as Coq. Process equivalence in particular added a significant amount of complexity to the implementation, on top of requiring a large number of definitions and proofs to accurately represent. Closed process equivalence was simple, but in turn necessitated the adjustment of every single calculus rule to accommodate the ignoring of the accompanying proofs for these closed processes.

There are quite a few takeaways from this work, some specific to elements of the Coq proof system and others generally about translation of on-paper calculi and other works into an electronic format. First of all, although it may be difficult to do so without familiarity with the work, it would likely help development to produce an outline of all the aspects of the calculus (or other work) and an estimation of the impact and importance of each part. Having to adjust the many definitions and proofs in the spi calculus implementation to account for notions of equivalence midway through could have been avoided with a better understanding of the impact it would have. However, this was mainly due to a lack of experience on the part of the main programmer in the area of dependent types in Coq. On the subject of dependent types, it is important to note that while they are a powerful feature of the Coq proof system, it is vital to plan around their use due to the nature of Coq's proof equality. Since proofs are rarely equal, using a dependent type may often require definition of a custom setoid equivalence to allow for effective reasoning about that type. They are an extremely valuable tool for representing elements under a specific condition, but are inflexible as a result.

Another Coq-specific point is the importance of notation and other visual aspects

when designing a system within Coq. At numerous points during development, some semantically trivial proofs were complicated by the overload of information present on the screen. Some function applications (newVar especially) could become extremely complex syntactically, leading to difficulty in continuing the proof based on the information given onscreen. This issue can be mitigated somewhat by relying less on the system, such as by doing "rough" proofs prior to proving them in the system. However, this raises the possibility of user error and it would be much more convenient to be able to use the system without being confused by the information presented. As such, it may also be beneficial to determine notation choices early on in a project and give more consideration toward what the implementation presents to the user during proofs, instead of focusing entirely on technical accuracy. The latter is important, but not so much that the final user experience should suffer.

In conclusion, the process of implementing the spi calculus in Coq has been a valuable experience in developing for a proof system (and a very strict one, at that) that will have real world applications as well. Though further work may be necessary to make the system an attractive option to the average user looking to do proof work regarding cryptographic protocols, the implementation at present should provide a strong base for moving forward with any such effort. The spi calculus turned out to be a far more complex system than expected, and presented many engaging challenges to overcome over the course of its implementation up to this point. It is our hope that it will be of use in fulfilling the purpose Abadi and Gordon intended for it, in this version or in the future.

Bibliography

[1] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1998.

[2] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. Technical report, Digital Systems Research Center, 1998.

[3] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, 1992.

[4] M. Boreale and R. De Nicola. Testing equivalence for mobile processes. *Information and Computation*, 120(2):279–303, 1995.

[5] R. De Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[6] J. W. Gray and J. McLean. Using temporal logic to specify and verify cryptographic protocols. In *8th Computer Security Foundations Workshop*, pages 108–116. IEEE, 1995.

[7] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.

[8] R. A. Kemmerer. Analyzing encryption protocols using formal verification techniques. *IEEE Journal on Selected Areas in Communications*, 7:448 – 457, 1989.

[9] B. Lampson, M. Abadi, M. Burrows, and R. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, 1992.

[10] A. Liebl. Authentication in distributed systems: A bibliography. *ACM Operating Systems Review*, 27(4):31–41, 1993.

[11] R. Milner. *Communication and Concurrency*. Prentice Hall International, 1989.

[12] R. Milner. *Communicating and Mobile Systems: The Pi Calculus*. Cambridge University Press, 1999.

[13] R. Milner and D. Sangiorgi. Barbed bisimulation. In *Proceedings of the 19th International Colloquium on Automata, Languages and Programming*, ICALP '92, pages 685–695, London, UK, UK, 1992. Springer-Verlag.

[14] L. Paulson. Proving properties of security protocols by induction. In *10th Computer Security Foundations Workshop*, pages 70–83. IEEE, 1997.

[15] D. Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2012.

[16] A. Tonet. Spi calculus implementation source files. `https://github.com/Corryn/spicalc`.