Typing Correspondence Assertions for Communication Protocols

Andrew D. Gordon

Microsoft Research, Cambridge

Alan Jeffrey

DePaul University, Chicago

Abstract

Woo and Lam propose correspondence assertions for specifying authenticity properties of security protocols. The only prior work on checking correspondence assertions depends on model-checking and is limited to finite-state systems. We propose a dependent type and effect system for checking correspondence assertions. Since it is based on type-checking, our method is not limited to finite-state systems. This paper presents our system in the simple and general setting of the π-calculus. We show how to type-check correctness properties of example communication protocols based on secure channels. In a related paper, we extend our system to the more complex and specific setting of checking cryptographic protocols based on encrypted messages sent over insecure channels.

1 Introduction

Correspondence Assertions To a first approximation, a correspondence assertion about a communication protocol is an intention that follows the pattern:

If one principal ever reaches a certain point in a protocol, then some other principal has previously reached some other matching point in the protocol.

We record such intentions by annotating the program representing the protocol with labelled assertions of the form $\mathsf{begin}\,L$ or $\mathsf{end}\,L$. These assertions have no effect at runtime, but notionally indicate that a principal has reached a certain point in the protocol. The following more accurately states the intention recorded by these annotations:

If the program embodying the protocol ever asserts $\mathsf{end}\ L$, then there is a distinct previous assertion of $\mathsf{begin}\ L$.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Woo and Lam [WL93] introduce correspondence assertions to state intended properties of authentication protocols based on cryptography. Consider a protocol where a principal a generates a new session key k and transmits it to b. We intend that if a run of b ends a key exchange believing that it has received key k from a, then a generated k as part of a key exchange intended for b. We record this intention by annotating a's generation of k by the label $\mathsf{begin}\ \langle a, b, k\rangle$, and b's reception of k by the label $\mathsf{end}\ \langle a, b, k\rangle$.

A protocol can fail a correspondence assertion because of several kinds of bug. One kind consists of those bugs that cause the protocol to go wrong without any external interference. Other kinds are bugs where an unreliable or malicious network or participant causes the protocol to fail.

This Paper We show in this paper that correctness properties expressed by correspondence assertions can be proved by type-checking. We embed correspondence assertions in a concurrent programming language (the π-calculus of Milner, Parrow, and Walker [Mil99]) and present a new type and effect system that guarantees safety of well-typed assertions. We show several examples of how correspondence assertions can be proved by type-checking.

Woo and Lam's paper introduces correspondence assertions but provides no techniques for proving them. Clarke and Marrero [CM00] use correspondence assertions to specify properties of e-commerce protocols, such as authorizations of transactions. To the best of our knowledge, the only previous work on checking correspondence assertions is a project by Marrero, Clarke, and Jha [MCJ97] to apply model-checking techniques to finite state versions of security protocols. Since our work is based on type-checking, it is not limited to finite state systems. Moreover, type-checking is compositional: we can verify components in isolation, and know that their composition is safe, without having to verify the entire system. Unlike Marrero, Clarke, and Jha's work, however, the system of the present paper does not deal with cryptographic primitives, and nor does it deal with an arbitrary opponent. Still, in another paper [GJ01], we adapt our type and effect system to the setting of the spi-calculus [AG99], an extension of the π-calculus with abstract cryptographic primitives. This adaptation can show, moreover, that properties hold in the presence of an arbitrary untyped opponent.

Review of The Untyped π-Calculus Milner, Parrow, and Walker's π-calculus is a concurrent formalism to which many kinds of concurrent computation may be reduced. Its simplicity makes it an attractive vehicle for developing the ideas of this paper, while its generality suggests they may be widely applicable. Its basic data type is the *name*, an unguessable identifier for a communications channel. Computation is based on the exchange of messages, tuples of names, on named channels. Programming in the π-calculus is based on the following constructs (written, unusually, with keywords, for the sake of clarity). The rest of the paper contains many examples. An output process $\mathsf{out}\ x\langle y_1, \ldots, y_n\rangle$ represents a message $\langle y_1, \ldots, y_n\rangle$ sent on the channel x. An input process $\mathsf{inp}\ x(z_1, \ldots, z_n); P$ blocks till it finds a message sent on the channel x, reads the names in the message into the variables $z_1, \ldots, z_n$, and then runs P. The process $P \mid Q$ is the parallel composition of the two processes P and Q; the two may run independently or communicate on shared channels. The name generation process $\mathsf{new}(x); P$ generates a fresh name, calls it x, then runs P. Unless P reveals x, no other process can use this fresh name. The replication process $\mathsf{repeat}\ P$ behaves like an unbounded parallel array of replicas of P. The process stop represents inactivity; it does nothing. Finally, the conditional $\mathsf{if}\ x = y\ \mathsf{then}\ P\ \mathsf{else}\ Q$ compares the names x and y. If they are the same it runs P; otherwise it runs Q.

2 Correspondence Assertions, by Example

This section introduces the idea of defining correspondence assertions by annotating code with begin- and end-events. We give examples of both safe code and of unsafe code, that is, of code that satisfies the correspondence assertions induced by its annotations, and of code that does not.

A transmit-acknowledge handshake is a standard communications idiom, easily expressed in the π-calculus: along with the actual message, the sender transmits an acknowledgement channel, upon which the receiver sends an acknowledgement. We intend that:

During a transmit-acknowledge handshake, if the sender receives an acknowledgment, then the receiver has obtained the message.

Correspondence assertions can express this intention formally. Suppose that a and b are the names of the sender and receiver, respectively. We annotate the code of the receiver b with a begin-assertion at the point after it has received the message msg. We annotate the code of the sender a with an end-assertion at the point after it has received the acknowledgement. We label both assertions with the names of the principals and the transmitted message, $\langle a, b, msg\rangle$. Hence, we assert that if after sending msg to b, the sender a receives an acknowledgement, then a distinct run of b has received msg.

Suppose that c is the name of the channel on which principal b receives messages from a. Here is the π-calculus code of the annotated sender and receiver:

$$\begin{array}{ll}
\mathit{Rcver}(a,b,c) \triangleq & \mathit{Snder}(a,b,c) \triangleq \\
\quad \mathsf{inp}\ c(\mathit{msg}, \mathit{ack}); & \quad \mathsf{new}(\mathit{msg}); \mathsf{new}(\mathit{ack}); \\
\quad \mathsf{begin}\ \langle a, b, \mathit{msg}\rangle; & \quad \mathsf{out}\ c\langle \mathit{msg}, \mathit{ack}\rangle; \mathsf{inp}\ \mathit{ack}(); \\
\quad \mathsf{out}\ \mathit{ack}\langle\rangle & \quad \mathsf{end}\ \langle a, b, \mathit{msg}\rangle
\end{array}$$

The sender creates a fresh message msg and a fresh acknowledgement channel ack, sends the two on the channel c, waits for an acknowledgement, and then asserts an end-event labelled $\langle a, b, msg\rangle$.

The receiver gets the message msg and the acknowledgement channel ack off c, asserts a begin-event labelled $\langle a, b, msg\rangle$, and sends an acknowledgement on ack.

We say a program is safe if it satisfies the intentions induced by the begin- and end-assertions. More precisely, a program is *safe* just if for every run of the program and for every label L, there is a distinct begin-event labelled L preceding every end-event labelled L. (We formalize this definition in Section 5.)

Here are three combinations of our examples: two safe, one unsafe.

$$\begin{array}{ll} \mathsf{new}(c); & \text{(Example 1: \textit{safe})} \\ \quad Snder(a, b, c) \mid & \\ \quad Rcver(a, b, c) & \end{array}$$

Example 1 uses one instance of the sender and one instance of the receiver to represent a single instance of the protocol. The restriction $\mathsf{new}(c)$; makes the channel c private to the sender and the receiver. This assembly is safe; its only run correctly implements the handshake protocol.

$$\begin{array}{ll} \mathsf{new}(c); & \text{(Example 2: \textit{safe})} \\ \quad Snder(a, b, c) \mid & \\ \quad Snder(a, b, c) \mid & \\ \quad \mathsf{repeat}\ Rcver(a, b, c) & \end{array}$$

Example 2 uses two copies of the sender—representing two attempts by a single principal a to send a message to b—and a replicated copy of the receiver—representing the principal b willing to accept an unbounded number of messages. Again, this assembly is safe; any run consists of an interleaving of two correct handshakes.

$$\begin{array}{ll} \mathsf{new}(c); & \text{(Example 3: \textit{unsafe})} \\ \quad Snder(a, b, c) \mid & \\ \quad Snder(a', b, c) \mid & \\ \quad \mathsf{repeat}\ Rcver(a, b, c) & \end{array}$$

Example 3 is a variant on Example 2, where we keep the replicated receiver b, but change the identity of one of the senders, so that the two senders represent two different principals a and a'. These two principals share a single channel c to the receiver. Since the identity a of the sender is a parameter of $Rcver(a, b, c)$ rather than being explicitly communicated, this assembly is unsafe. There is a run in which a' generates msg and ack, and sends them to b; b asserts a begin-event labelled $\langle a, b, msg\rangle$ and outputs on ack; then a' asserts an end-event labelled $\langle a', b, msg\rangle$. This end-event has no corresponding begin-event so the assembly is unsafe, reflecting the possibility that the receiver can be mistaken about the identity of the sender.

3 Typing Correspondence Assertions

3.1 Types and Effects

Our type and effect system is based on the idea of assigning types to names and effects to processes. A type describes what operations are allowed on a name, such as what messages may be communicated on a channel name. An effect describes the collection of labels of events the process may end while not itself beginning. We compute effects based on the intuition that end-events are accounted for by preceding begin-events; a begin-event is a credit while an end-event is a debit. According to this metaphor, the effect of a process is an upper bound on the debt a process may incur. If we can assign a process the empty effect, we know all of its end-events are accounted for by begin-events. Therefore, we know that the process is safe, that is, its correspondence assertions are true.

An essential ingredient of our typing rules is the idea of attaching a *latent effect* to each channel type. We allow any process receiving off a channel to treat the latent effect as a credit towards subsequent end-events. This is sound because we require any process sending on a channel to treat the latent effect as a debit that must be accounted for by previous begin-events. Latent effects are at the heart of our method for type-checking events begun by one process and ended by another.

The following table describes the syntax of types and effects. As in most versions of the π-calculus, we make no lexical distinction between names and variables, ranged over by a, b, c, x, y, z. An *event label*, L, is simply a tuple of names. Event labels identify the events asserted by begin- and end-assertions. An *effect*, e, is a multiset, that is, an unordered list, of event labels, written as $[L_1, \ldots, L_n]$. A *type*, T, takes one of two kinds. The first kind, Name, is the type of pure names, that is, names that only support equality operations, but cannot be used as channels. We use Name as the type of names that identify principals, for instance. The second kind, $\mathsf{Ch}(x_1{:}T_1, \ldots, x_n{:}T_n)e$, is a type of a channel communicating n-tuples of names, of types $T_1, \ldots, T_n$, with latent effect e. The names $x_1, \ldots, x_n$ are bound; the scope of each x_i consists of the types $T_{i+1}, \ldots, T_n$, and the latent effect e. We identify types up to the consistent renaming of bound names.

Names, Event Labels, Effects, and Types:

a, b, c, x, y, z	names, variables
$L ::= \langle x_1, \ldots, x_n \rangle$	event label: tuple of names
$e ::= [L_1, \ldots, L_n]$	effect: multiset of event labels
$T ::=$	type
$\quad \mathsf{Name}$	pure name
$\quad \mathsf{Ch}(x_1{:}T_1, \ldots, x_n{:}T_n)e$	channel with latent effect e

For example:

- $\mathsf{Ch}()[\,]$, a synchronization channel (that is, a channel used only for synchronization) with no latent effect.
- $\mathsf{Ch}(a{:}\mathsf{Name})[\langle b\rangle]$, a channel for communicating a pure name, costing $[\langle b\rangle]$ to senders and paying $[\langle b\rangle]$ to receivers, where b is a fixed name.
- $\mathsf{Ch}(a{:}\mathsf{Name})[\langle a\rangle]$, a channel for communicating a pure name, costing $[\langle a\rangle]$ to senders and paying $[\langle a\rangle]$ to receivers, where a is the name communicated on the channel.
- $\mathsf{Ch}(a{:}\mathsf{Name}, b{:}\mathsf{Ch}()[\langle a\rangle])[\,]$, a channel with no latent effect for communicating pairs of the form a, b, where a is a pure name, and b is the name of a synchronization channel, costing $[\langle a\rangle]$ to senders and paying $[\langle a\rangle]$ to receivers.

The following is a convenient shorthand for the lists of typed variable declarations found in channel types:

Notation for Typed Variables:

$\vec{x}{:}\vec{T} \triangleq x_1{:}T_1, \ldots, x_n{:}T_n$	where $\vec{x} = x_1, \ldots, x_n$ and $\vec{T} = T_1, \ldots, T_n$
$\epsilon \triangleq ()$	the empty list

The following equations define the the sets of free names of our syntax as follows: variable declarations, $\mathsf{fn}(\epsilon{:}\epsilon) \triangleq \varnothing$ and $\mathsf{fn}(\vec{x}{:}\vec{T}, x{:}T) \triangleq \mathsf{fn}(\vec{x}{:}\vec{T}) \cup (\mathsf{fn}(T) - \{\vec{x}\})$; types, $\mathsf{fn}(\mathsf{Name}) \triangleq \varnothing$ and $\mathsf{fn}(\mathsf{Ch}(\vec{x}{:}\vec{T})e) \triangleq \mathsf{fn}(\vec{x}{:}\vec{T}) \cup (\mathsf{fn}(e) - \{\vec{x}\})$; event labels, $\mathsf{fn}(\langle x_1, \ldots, x_n\rangle) \triangleq \{x_1, \ldots, x_n\}$; and events, $\mathsf{fn}([L_1, \ldots, L_1]) \triangleq \mathsf{fn}(L_1) \cup \cdots \cup \mathsf{fn}(L_n)$.

For any of these forms of syntax, we write $-\{x{\leftarrow}y\}$ for the operation of capture-avoiding substitution of the name y for each free occurrence of the name x. We write $-\{\vec{x}{\leftarrow}\vec{y}\}$, where $\vec{x} = x_1, \ldots, x_n$ and $\vec{y} = y_1, \ldots, y_n$ for the iterated substitution $-\{x_1{\leftarrow}y_1\} \cdots \{x_n{\leftarrow}y_n\}$.

3.2 Syntax of our Typed π-Calculus

We explained the informal semantics of begin- and end-assertions in Section 2, and of the other constructs in Section 1.

Processes:

$P, Q, R ::=$	process
$\mathsf{out}\ x\langle y_1, \ldots, y_n\rangle$	polyadic asynchronous output
$\mathsf{inp}\ x(y_1{:}T_1, \ldots, y_n{:}T_n); P$	polyadic input
$\mathsf{if}\ x = y\ \mathsf{then}\ P\ \mathsf{else}\ Q$	conditional
$\mathsf{new}(x{:}T); P$	name generation
$P \mid Q$	composition
$\mathsf{repeat}\ P$	replication
stop	inactivity
$\mathsf{begin}\ L; P$	begin-assertion
$\mathsf{end}\ L; P$	end-assertion

There are two name binding constructs: input and name generation. In an input process $\mathsf{inp}\ x(y_1{:}T_1, \ldots, y_n{:}T_n); P$, each name y_i is bound, with scope consisting of T_{i+1}, $\ldots$, T_n, and P. In a name restriction $\mathsf{new}(x{:}T); P$, the name x is bound; its scope is P. We write $P\{x{\leftarrow}y\}$ for the outcome of a capture-avoiding substitution of the name y for each free occurrence of the name x in the process P. We identify processes up to the consistent renaming of bound names. We let $\mathsf{fn}(P)$ be the set of free names of a process P. We sometimes write an output as $\mathsf{out}\ x\langle\vec{y}\rangle$ where $\vec{y} = y_1, \ldots, y_n$, and an input as $\mathsf{inp}\ x(\vec{y}{:}\vec{T}); P$, where $\vec{y}{:}\vec{T}$ is a variable declaration written in the notation introduced in the previous section. We write $\mathsf{out}\ x\langle\vec{y}\rangle; P$ as a shorthand for $\mathsf{out}\ x\langle\vec{y}\rangle \mid P$.

3.3 Intuitions for the Type and Effect System

As a prelude to our formal typing rules, we present the underlying intuitions. Recall the intuition that end-events are costs to be accounted for by begin-events. When we say a process P has effect e, it means that e is an upper bound on the begin-events needed to precede P to make the whole process safe. In other words, if P has effect $[L_1, \ldots, L_n]$ then $\mathsf{begin}\,L_1; \cdots; \mathsf{begin}\,L_n; P$ is safe.

Typing Assertions An assertion $\mathsf{begin}\,L; P$ pays for one end-event labelled L in P; so if P is a process with effect e, then $\mathsf{begin}\ L; P$ is a process with effect $e{-}[L]$, that is, the multiset e with one occurrence of L deleted. So we have a typing rule of the form:

$$P : e \quad \Rightarrow \quad \mathsf{begin}\,L; P : e{-}[L]$$

If P is a process with effect e, then $\mathsf{end}\,L; P$ is a process with effect $e{+}[L]$, that is, the concatenation of e and $[L]$. We have a rule:

$$P : e \quad \Rightarrow \quad \mathsf{end}\ L; P : e{+}[L]$$

Typing Name Generation and Concurrency The effect of a name generation process $\mathsf{new}(x{:}T); P$, is simply the effect of P. To prevent scope confusion, we forbid x from occurring in this effect.

$$P : e,\ x \notin \mathsf{fn}(e) \quad \Rightarrow \quad \mathsf{new}(x{:}T); P : e$$

The effect of a concurrent composition of processes is the multiset union of the constituent processes.

$$P : e_P,\ Q : e_Q \quad \Rightarrow \quad P \mid Q : e_P{+}e_Q$$

The inactive process asserts no end-events, so its effect is empty.

$$\mathsf{stop} : [\,]$$

The replication of a process P behaves like an unbounded array of replicas of P. If P has a non-empty effect, then its replication would have an unbounded effect, which could not be accounted for by preceding begin-assertions. Therefore, to type repeat P we require P to have an empty effect.

$$P : [\,] \quad \Rightarrow \quad \mathsf{repeat}\ P : [\,]$$

Typing Communications We begin by presenting the rules for typing communications on monadic channels with no latent effect, that is, those with types of the form $\mathsf{Ch}(y{:}T)[\,]$. The communicated name has type T. An output $\mathsf{out}\ x\langle z\rangle$ has empty effect. An input $\mathsf{inp}\ x(y{:}T); P$ has the same effect as P. Since the input variable in the process and in the type are both bound, we may assume they are the same variable y.

$$x : \mathsf{Ch}(y{:}T)[\,],\ z : T \quad \Rightarrow \quad \mathsf{out}\ x\langle z\rangle : [\,]$$
$$x : \mathsf{Ch}(y{:}T)[\,],\ P : e,\ y \notin \mathsf{fn}(e) \quad \Rightarrow \quad \mathsf{inp}\ x(y{:}T); P : e$$

Next, we consider the type $\mathsf{Ch}(y{:}T)e_\ell$ of monadic channels with latent effect e_ℓ. The latent effect is a cost to senders, a benefit to receivers, and is the scope of the variable y. We assign an output $\mathsf{out}\ x\langle z\rangle$ the effect $e_\ell\{y{\leftarrow}z\}$, where we have instantiated the name y bound in the type of the channel with z, the name actually sent on the channel. We assign an input $\mathsf{inp}\ x(y{:}T); P$ the effect $e - e_\ell$, where e is the effect of P. To avoid scope confusion, we require that y is not free in $e - e_\ell$.

$$x : \mathsf{Ch}(y{:}T)e_\ell,\ z : T \quad \Rightarrow \quad \mathsf{out}\ x\langle z\rangle : e_\ell\{y{\leftarrow}z\}$$
$$x : \mathsf{Ch}(y{:}T)e_\ell,\ P : e,\ y \notin \mathsf{fn}(e - e_\ell) \quad \Rightarrow \quad \mathsf{inp}\ x(y{:}T); P : e - e_\ell$$

The formal rules for input and output in the next section generalize these rules to deal with polyadic channels.

Typing Conditionals When typing a conditional if $x = y$ then P else Q, it is useful to exploit the fact that P only runs if the two names x and y are equal. To do so, we check the effect of P after substituting one for the other. Suppose then process $P\{x{\leftarrow}y\}$ has effect $e_P\{x{\leftarrow}y\}$. Suppose also that process Q has effect e_Q. Let $e_P \vee e_Q$ be the least upper bound of any two effects e_P and e_Q. Then $e_P \vee e_Q$ is an upper bound on the begin-events needed to precede the conditional to make it safe, whether P or Q runs. An example in Section 4.2 illustrates this rule.

$$P\{x{\leftarrow}y\} : e_P\{x{\leftarrow}y\},\ Q : e_Q \quad \Rightarrow \quad \mathsf{if}\ x = y\ \mathsf{then}\ P\ \mathsf{else}\ Q : e_P \vee e_Q$$

3.4 Typing Rules

Our typing rules depend on several operations on effect multisets, most of which were introduced informally in the previous section. Here are the formal definitions.

Operations on effects: $e + e'$, $e \leq e'$, $e - e'$, $L \in e$, $e \vee e'$

$$[L_1, \ldots, L_m] + [L_{m+1}, \ldots, L_{m+n}] \triangleq [L_1, \ldots, L_{m+n}]$$

$e \leq e'$ if and only if $e' = e + e''$ for some e''
$e - e' \triangleq$ the smallest e'' such that $e \leq e' + e''$
$L \in e$ if and only if $[L] \leq e$
$e \vee e' \triangleq$ the smallest e'' such that $e \leq e''$ and $e' \leq e''$

The typing judgments of this section depend on an environment to assign a type to all the variables in scope.

Environments:

$E ::= \vec{x}{:}\vec{T}$	environment
$\mathsf{dom}(\vec{x}{:}\vec{T}) \triangleq \{\vec{x}\}$	domain of an environment

To equate two names in an environment, needed for typing conditionals, we define a name fusion function. We obtain the fusion $E\{x{\leftarrow}x'\}$ from E by turning all occurrences of x and x' in E into x'.

Fusing x with x' in E: $E\{x{\leftarrow}x'\}$

$$(x_1{:}T_1, \ldots, x_n{:}T_n)\{x{\leftarrow}x'\} \triangleq$$
$$(x_1\{x{\leftarrow}x'\}){:}(T_1\{x{\leftarrow}x'\}); \ldots; (x_n\{x{\leftarrow}x'\}){:}(T_n\{x{\leftarrow}x'\})$$
$$\text{where } E; x{:}T \triangleq \begin{cases} E & \text{if } x \in \mathsf{dom}(E) \\ E, x{:}T & \text{otherwise} \end{cases}$$

The following table summarizes the five judgments of our type system, which are inductively defined by rules in subsequent tables. Judgment $E \vdash \diamond$ means environment E is well-formed. Judgment $E \vdash T$ means type T is well-formed. Judgment $E \vdash x : T$ means name x is in scope with type T. Judgment $E \vdash \langle\vec{x}\rangle : \langle\vec{y}{:}\vec{T}\rangle$ means tuple $\langle\vec{x}\rangle$ matches the variable declaration $\vec{y}{:}\vec{T}$. Judgment $E \vdash P : e$ means process P has effect e.

Judgments:

$E \vdash \diamond$	good environment
$E \vdash T$	good type T
$E \vdash x : T$	good name x of type T
$E \vdash \langle\vec{x}\rangle : \langle\vec{y}{:}\vec{T}\rangle$	good message $\vec{x}$ matching $\vec{y}{:}\vec{T}$
$E \vdash P : e$	good process P with effect e

The rules defining the first three judgments are standard.

Good environments, types, and names:

(Env $\varnothing$) (Env x) (Type Name)

$$\frac{}{\varnothing \vdash \diamond} \qquad \frac{E \vdash T \quad x \notin \mathsf{dom}(E)}{E, x{:}T \vdash \diamond} \qquad \frac{E \vdash \diamond}{E \vdash \mathsf{Name}}$$

(Type Chan)
$$\frac{E,\vec{x}{:}\vec{T} \vdash \diamond \quad \mathsf{fn}(e) \subseteq \mathsf{dom}(E) \cup \{\vec{x}\}}{E \vdash \mathsf{Ch}(\vec{x}{:}\vec{T})e}$$

(Name x)
$$\frac{E', x{:}T, E'' \vdash \diamond}{E', x{:}T, E'' \vdash x : T}$$

The next judgment, $E \vdash \langle\vec{x}\rangle : \langle\vec{y}{:}\vec{T}\rangle$, is an auxiliary judgment used for typing output processes; it is used in the rule (Proc Output) to check that the message $\langle\vec{x}\rangle$ sent on a channel of type $\mathsf{Ch}(\vec{y}{:}\vec{T})e$ matches the variable declaration $\vec{y}{:}\vec{T}$.

Good message:

(Msg $\langle\rangle$)
$$\frac{E \vdash \diamond}{E \vdash \langle\rangle : \langle\rangle}$$

(Msg x) (where $y \notin \{\vec{y}\} \cup \mathsf{dom}(E)$)
$$\frac{E \vdash \langle\vec{x}\rangle : \langle\vec{y}{:}\vec{T}\rangle \quad E \vdash x : (T\{\vec{y}\leftarrow\vec{x}\})}{E \vdash \langle\vec{x}, x\rangle : \langle\vec{y}{:}\vec{T}, y{:}T\rangle}$$

Finally, here are the rules for typing processes. The effect of a process is an upper bound; the rule (Proc Subsum) allows us to increase this upper bound. Intuitions for all the other rules were explained in the previous section.

Good processes:

(Proc Subsum) (where $e \leq e'$ and $\mathsf{fn}(e') \subseteq \mathsf{dom}(E)$)
$$\frac{E \vdash P : e}{E \vdash P : e'}$$

(Proc Output)
$$\frac{E \vdash x : \mathsf{Ch}(\vec{y}{:}\vec{T})e \quad E \vdash \langle\vec{x}\rangle : \langle\vec{y}{:}\vec{T}\rangle}{E \vdash \mathsf{out}\ x\langle\vec{x}\rangle : (e\{\vec{y}\leftarrow\vec{x}\})}$$

(Proc Input) (where $\mathsf{fn}(e - e') \subseteq \mathsf{dom}(E)$)
$$\frac{E \vdash x : \mathsf{Ch}(\vec{y}{:}\vec{T})e' \quad E, \vec{y}{:}\vec{T} \vdash P : e}{E \vdash \mathsf{inp}\ x(\vec{y}{:}\vec{T}); P : e - e'}$$

(Proc Cond)
$$\frac{E \vdash x : T \quad E \vdash y : T \quad E\{x\leftarrow y\} \vdash P\{x\leftarrow y\} : e_P\{x\leftarrow y\} \quad E \vdash Q : e_Q}{E \vdash \mathsf{if}\ x = y\ \mathsf{then}\ P\ \mathsf{else}\ Q : e_P \vee e_Q}$$

(Proc Res) (where $x \notin \mathsf{fn}(e)$)
$$\frac{E, x{:}T \vdash P : e}{E \vdash \mathsf{new}(x{:}T); P : e}$$

(Proc Par)
$$\frac{E \vdash P : e_P \quad E \vdash Q : e_Q}{E \vdash P \mid Q : e_P + e_Q}$$

(Proc Repeat)
$$\frac{E \vdash P : [\,]}{E \vdash \mathsf{repeat}\ P : [\,]}$$

(Proc Stop)
$$\frac{E \vdash \diamond}{E \vdash \mathsf{stop} : [\,]}$$

(Proc Begin) (where $\mathsf{fn}(L) \subseteq \mathsf{dom}(E)$)

$$\frac{E \vdash P : e}{E \vdash \mathsf{begin}\ L; P : e - [L]}$$

(Proc End) (where $\mathsf{fn}(L) \subseteq \mathsf{dom}(E)$)

$$\frac{E \vdash P : e}{E \vdash \mathsf{end}\ L; P : e + [L]}$$

Section 5 presents our main type safety result, Theorem 5.2, that $E \vdash P : [\,]$ implies P is safe. Like most type systems, ours is incomplete. There are safe processes that are not typeable in our system. For example, we cannot assign the process $\mathsf{if}\ x = x\ \mathsf{then\ stop\ else}\ (\mathsf{end}\ x; \mathsf{stop})$ the empty effect, and yet it is perfectly safe.

4 Applications

In this section, we present some examples of using correspondence assertions to validate safety properties of communication protocols. For more examples, including examples with cryptographic protocols which are secure against external attackers, see the companion paper [GJ01].

4.1 Transmit-Acknowledge Handshake

Recall the untyped sender and receiver code from Section 2. Suppose we make the type definitions:

$$\begin{array}{ll} \mathit{Msg} \triangleq \mathsf{Name} & \mathit{Ack}(a, b, \mathit{msg}) \triangleq \mathsf{Ch}()[\langle a, b, \mathit{msg}\rangle] \\ \mathit{Host} \triangleq \mathsf{Name} & \mathit{Req}(a, b) \triangleq \mathsf{Ch}(\mathit{msg}{:}\mathit{Msg}, \mathit{ack}{:}\mathit{Ack}(a, b, \mathit{msg}))[\,] \end{array}$$

Suppose also that we annotate the sender and receiver code, and the code of Example 1 as follows:

```
$\mathit{Snder}(a{:}\mathit{Host}, b{:}\mathit{Host}, c{:}\mathit{Req}(a, b)) \triangleq$
  $\mathsf{new}(\mathit{msg}{:}\mathit{Msg});$
  $\mathsf{new}(\mathit{ack}{:}\mathit{Ack}(a, b, \mathit{msg}));$
  $\mathsf{out}\ c\langle \mathit{msg}, \mathit{ack}\rangle;$
  $\mathsf{inp}\ \mathit{ack}();$
  $\mathsf{end}\ \langle a, b, \mathit{msg}\rangle$

$\mathit{Rcver}(a{:}\mathit{Host}, b{:}\mathit{Host}, c{:}\mathit{Req}(a, b)) \triangleq$
  $\mathsf{inp}\ c(\mathit{msg}{:}\mathit{Msg}, \mathit{ack}{:}\mathit{Ack}(a, b, \mathit{msg}));$
  $\mathsf{begin}\ \langle a, b, \mathit{msg}\rangle;$
  $\mathsf{out}\ \mathit{ack}\langle\rangle$

$\mathit{Example1}(a{:}\mathit{Host}, b{:}\mathit{Host}) \triangleq$
  $\mathsf{new}(c{:}\mathit{Req}(a, b));$
    $\mathit{Snder}(a, b, c)\ |$
    $\mathit{Rcver}(a, b, c)$
```

We can then check that $a{:}\mathit{Host}, b{:}\mathit{Host} \vdash \mathit{Example1}(a, b) : [\,]$. Since the system has the empty effect, by Theorem 5.2 it is safe. It is routine to check that

Example 2 from Section 2 also has the empty effect, but that Example 3 cannot be type-checked (as to be expected, since it is unsafe).

4.2 Hostname Lookup

In this example, we present a simple hostname lookup system, where a client b wishing to ping a server a can contact a name server *query*, to get a network address *ping* for a. The client can then send a ping request to the address *ping*, and get an acknowledgement from the server. We shall check two properties:

- When the ping client b finishes, it believes that the ping server a has been pinged.
- When the ping server a finishes, it believes that it was contacted by the ping client b.

We write "a was pinged by b" as shorthand for $\langle a, b\rangle$, and "b tried to ping a" for $\langle b, a, a\rangle$. These examples are well-typed, with types which we define later in this section.

We program the ping client and server as follows.

```
$PingClient(a{:}Hostname, b{:}Hostname, query{:}Query) \triangleq$
  new($res : Res(a)$);
  out $query\langle a, res\rangle$;
  inp $res(ping : Ping(a))$;
  new($ack : Ack(a, b)$);
  begin "$b$ tried to ping $a$";
  out $ping\langle b, ack\rangle$;
  inp $ack()$;
  end "$a$ was pinged by $b$"

$PingServer(a : Hostname, ping : Ping(a)) \triangleq$
  repeat
    inp $ping(b : Hostname, ack : Ack(a, b))$;
    begin "$a$ was pinged by $b$";
    end "$b$ tried to ping $a$";
    out $ack\langle\rangle$
```

If these processes are safe, then any ping request and response must come as matching pairs. In practice, the name server would require some data structure such as a hash table or database, but for this simple example we just use a large if-statement:

$$
\begin{array}{l}
\mathit{NameServer}(\\
\quad \mathit{query}{:}\mathit{Query}, \\
\quad h_1{:}\mathit{Hostname}, \ldots, h_n{:}\mathit{Hostname}, \\
\quad \mathit{ping}_1{:}\mathit{Ping}(h_1), \ldots, \mathit{ping}_n{:}\mathit{Ping}(h_n) \\
) \triangleq \\
\quad \mathsf{repeat} \\
\quad\quad \mathsf{inp}\ \mathit{query}(h, \mathit{res}); \\
\quad\quad \mathsf{if}\ h = h_1\ \mathsf{then\ out}\ \mathit{res}\langle \mathit{ping}_1 \rangle\ \mathsf{else}\ \cdots \\
\quad\quad \mathsf{if}\ h = h_n\ \mathsf{then\ out}\ \mathit{res}\langle \mathit{ping}_n \rangle\ \mathsf{else\ stop}
\end{array}
$$

To get the system to type-check, we use the following types:

$$
\begin{aligned}
\mathit{Hostname} &\triangleq \mathsf{Name} \\
\mathit{Ack}(a, b) &\triangleq \mathsf{Ch}()[\text{“}a\text{ was pinged by }b\text{”}] \\
\mathit{Ping}(a) &\triangleq \mathsf{Ch}(b{:}\mathit{Hostname}, \mathit{ack}{:}\mathit{Ack}(a, b))[\text{“}b\text{ tried to ping }a\text{”}] \\
\mathit{Res}(a) &\triangleq \mathsf{Ch}(\mathit{ping}{:}\mathit{Ping}(a))[\,] \\
\mathit{Query} &\triangleq \mathsf{Ch}(a{:}\mathit{Hostname}, \mathit{res}{:}\mathit{Res}(a))[\,]
\end{aligned}
$$

The most subtle part of type-checking the system is the conditional in the name server. A typical branch is:

$$
\begin{array}{l}
h_i : \mathit{Hostname}, \mathit{ping}_i : \mathit{Ping}(h_i), h : \mathit{Hostname}, \mathit{res} : \mathit{Res}(h) \\
\quad \vdash \mathsf{if}\ h = h_i\ \mathsf{then\ out}\ \mathit{res}\langle \mathit{ping}_i \rangle\ \mathsf{else}\ \cdots : [\,]
\end{array}
$$

When type-checking the then-branch, (Proc Cond) assumes $h = h_i$ by applying a substitution to the environment:

$$
\begin{array}{l}
(h_i : \mathit{Hostname}, \mathit{ping}_i : \mathit{Ping}(h_i), h : \mathit{Hostname}, \mathit{res} : \mathit{Res}(h))\{h \leftarrow h_i\} \\
\quad = (h_i : \mathit{Hostname}, \mathit{ping}_i : \mathit{Ping}(h_i), \mathit{res} : \mathit{Res}(h_i))
\end{array}
$$

In this environment, we can type-check the then-branch:

$$
\begin{array}{l}
h_i : \mathit{Hostname}, \mathit{ping}_i : \mathit{Ping}(h_i), \mathit{res} : \mathit{Res}(h_i) \\
\quad \vdash \mathsf{out}\ \mathit{res}\langle \mathit{ping}_i \rangle : [\,]
\end{array}
$$

If (Proc Cond) did not apply the substitution to the environment, this example could not be type-checked, since:

$$
\begin{array}{l}
h_i : \mathit{Hostname}, \mathit{ping}_i : \mathit{Ping}(h_i), h : \mathit{Hostname}, \mathit{res} : \mathit{Res}(h) \\
\quad \nvdash \mathsf{out}\ \mathit{res}\langle \mathit{ping}_i \rangle : [\,]
\end{array}
$$

4.3 Functions

It is typical to code the λ-calculus into the π-calculus, using a return channel k as the destination for the result. For instance, the hostname lookup example of the previous section can be rewritten in the style of a remote procedure call. The client and server are now:

```
$PingClient(a{:}Hostname, b{:}Hostname, query{:}Query) \triangleq$
  let $(ping : Ping(a)) = query\ \langle a \rangle$;
  begin "$b$ tried to ping $a$";
  let $() = ping\ \langle b \rangle$;
  end "$a$ was pinged by $b$"

$PingServer(a : Hostname, ping : Ping(a)) \triangleq$
  fun $ping(b{:}Hostname)$ {
    begin "$a$ was pinged by $b$";
    end "$b$ tried to ping $a$";
    return $\langle\rangle$
  }
```

The name server is now:

```
$NameServer($
  $query{:}Query,$
  $h_1{:}Hostname, \ldots, h_n{:}Hostname,$
  $ping_1{:}Ping(h_1), \ldots, ping_n{:}Ping(h_n)$
$) \triangleq$
  fun $query(h{:}Hostname)$ {
    if $h = h_1$ then return $\langle ping_1 \rangle$ else $\cdots$
    if $h = h_n$ then return $\langle ping_n \rangle$ else stop
  }
```

In order to provide types for these examples, we have to provide a function type with *latent effects*. These effects are *precondition/postcondition* pairs, which act like Hoare triples. In the type $(\vec{x}{:}\vec{T})e \rightarrow (\vec{y}{:}\vec{U})e'$ we have a precondition e which the callee must satisfy, and a postcondition e' which the caller must satisfy. For example, the types for the hostname lookup example are:

$$\begin{aligned} Ping(a) &\triangleq (b{:}Hostname)[\text{“}b\text{ tried to ping }a\text{”}] \rightarrow ()[\text{“}a\text{ was pinged by }b\text{”}] \\ Query &\triangleq (a{:}Hostname)[\,] \rightarrow (ping{:}Ping(a))[\,] \end{aligned}$$

which specifies that the remote ping call has a precondition "b tried to ping a" and a postcondition "a was pinged by b".

This can be coded into the π-calculus using a translation [Mil99] in continuation passing style.

$$\begin{aligned} \mathsf{fun}\ f(\vec{x}{:}\vec{T})\ \{P\} &\triangleq \mathsf{repeat\ inp}\ f(\vec{x}{:}\vec{T}, k{:}\mathsf{Ch}(\vec{y}{:}\vec{U})e'); P \\ \mathsf{let}\ (\vec{y}{:}\vec{U}) = f\ \langle \vec{x} \rangle; P &\triangleq \mathsf{new}(k{:}\mathsf{Ch}(\vec{y}{:}\vec{U})e'); \mathsf{out}\ f\langle \vec{x}, k \rangle; \mathsf{inp}\ k(\vec{y}{:}\vec{U}); P \\ \mathsf{return}\ \langle \vec{z} \rangle &\triangleq \mathsf{out}\ k\langle \vec{z} \rangle \\ (\vec{x}{:}\vec{T})e \rightarrow (\vec{y}{:}\vec{U})e' &\triangleq \mathsf{Ch}(\vec{x}{:}\vec{T}, k{:}\mathsf{Ch}(\vec{y}{:}\vec{U})e')e \end{aligned}$$

This translation is standard, except for the typing. It is routine to verify its soundness.

5 Formalizing Correspondence Assertions

In this section, we give the formal definition of the trace semantics for the π-calculus with correspondence assertions, which is used in the definition of a safe process. We then state the main result of this paper, which is that effect-free processes are safe.

We give the trace semantics as a labelled transition system. Following Berry and Boudol [BB92] and Milner [Mil99] we use a structural congruence $P \equiv Q$, and give our operational semantics up to $\equiv$.

Structural Congruence: $P \equiv Q$

$P \equiv P$	(Struct Refl)
$Q \equiv P \Rightarrow P \equiv Q$	(Struct Symm)
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$	(Struct Trans)
$P \equiv Q \Rightarrow \mathsf{inp}\ x(\vec{y}{:}\vec{T}); P \equiv \mathsf{inp}\ x(\vec{y}{:}\vec{T}); Q$	(Struct Input)
$P \equiv Q \Rightarrow \mathsf{new}(x{:}T); P \equiv \mathsf{new}(x{:}T); Q$	(Struct Res)
$P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$	(Struct Par)
$P \equiv Q \Rightarrow \mathsf{repeat}\ P \equiv \mathsf{repeat}\ Q$	(Struct Repl)
$P \mid \mathsf{stop} \equiv P$	(Struct Par Zero)
$P \mid Q \equiv Q \mid P$	(Struct Par Comm)
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	(Struct Par Assoc)
$\mathsf{repeat}\ P \equiv P \mid \mathsf{repeat}\ P$	(Struct Repl Par)
$\mathsf{new}(x{:}T); (P \mid Q) \equiv P \mid \mathsf{new}(x{:}T); Q$	(Struct Res Par) (where $x \notin \mathsf{fn}(P)$)
$\mathsf{new}(x_1{:}T_1); \mathsf{new}(x_2{:}T_2); P \equiv$	(Struct Res Res)
$\quad \mathsf{new}(x_2{:}T_2); \mathsf{new}(x_1{:}T_1); P$	(where $x_1 \neq x_2, x_1 \notin \mathsf{fn}(T_2), x_2 \notin \mathsf{fn}(T_1)$)

There are four actions in this labelled transition system:

- $P \xrightarrow{\mathsf{begin}\ L} P'$ when P reaches a $\mathsf{begin}\ L$ assertion.
- $P \xrightarrow{\mathsf{end}\ L} P'$ when P reaches an $\mathsf{end}\ L$ assertion.
- $P \xrightarrow{\mathsf{gen}\ \langle x \rangle} P'$ when P generates a new name x.
- $P \xrightarrow{\tau} P'$ when P can perform an internal action.

For example:

$$\begin{aligned}(\mathsf{new}(x{:}\mathsf{Name}); \mathsf{begin}\ \langle x \rangle; \mathsf{end}\ \langle x \rangle; \mathsf{stop}) &\xrightarrow{\mathsf{gen}\ \langle x \rangle} (\mathsf{begin}\ \langle x \rangle; \mathsf{end}\ \langle x \rangle; \mathsf{stop})\\ &\xrightarrow{\mathsf{begin}\ \langle x \rangle} (\mathsf{end}\ \langle x \rangle; \mathsf{stop})\\ &\xrightarrow{\mathsf{end}\ \langle x \rangle} (\mathsf{stop})\end{aligned}$$

Next, we define the syntax of actions α, and their free names and generated names.

Actions:

$\alpha, \beta ::=$	actions
$\mathsf{begin}\ L$	begin-event
$\mathsf{end}\ L$	end-event
$\mathsf{gen}\ \langle x \rangle$	name generation
τ	internal

Free names, $\mathsf{fn}(\alpha)$, and generated names, $\mathsf{gn}(\alpha)$, of an action α:

$$\mathsf{fn}(\tau) \triangleq \varnothing \quad \mathsf{fn}(\mathsf{begin}\ L) \triangleq \mathsf{fn}(L) \quad \mathsf{fn}(\mathsf{end}\ L) \triangleq \mathsf{fn}(L) \quad \mathsf{fn}(\mathsf{gen}\ \langle x \rangle) \triangleq \{x\}$$
$$\mathsf{gn}(\tau) \triangleq \varnothing \quad \mathsf{gn}(\mathsf{begin}\ L) \triangleq \varnothing \quad \mathsf{gn}(\mathsf{end}\ L) \triangleq \varnothing \quad \mathsf{gn}(\mathsf{gen}\ \langle x \rangle \triangleq \{x\}$$

The labelled transition system $P \xrightarrow{\alpha} P'$ is defined here.

Transitions: $P \xrightarrow{\alpha} P'$

$\mathsf{out}\ x\langle \vec{x} \rangle \mid \mathsf{inp}\ x(\vec{y}); P \xrightarrow{\tau} P\{\vec{y}\leftarrow\vec{x}\}$	(Trans Comm)
$\mathsf{if}\ x = x\ \mathsf{then}\ P\ \mathsf{else}\ Q \xrightarrow{\tau} P$	(Trans Match)
$\mathsf{if}\ x = y\ \mathsf{then}\ P\ \mathsf{else}\ Q \xrightarrow{\tau} Q$	(Trans Mismatch) (where $x \neq y$)
$\mathsf{begin}\ L; P \xrightarrow{\mathsf{begin}\ L} P$	(Trans Begin)
$\mathsf{end}\ L; P \xrightarrow{\mathsf{end}\ L} P$	(Trans End)
$\mathsf{new}(x{:}T); P \xrightarrow{\mathsf{gen}\ \langle x \rangle} P$	(Trans Gen)
$P \xrightarrow{\alpha} P' \Rightarrow P \mid Q \xrightarrow{\alpha} P' \mid Q$	(Trans Par) (where $\mathsf{gn}(\alpha) \cap \mathsf{fn}(Q) = \varnothing$)
$P \xrightarrow{\alpha} P' \Rightarrow \mathsf{new}(x{:}T); P \xrightarrow{\alpha} \mathsf{new}(x{:}T); P'$	(Trans Res) (where $x \notin \mathsf{fn}(\alpha)$)
$P \equiv P', P' \xrightarrow{\alpha} Q', Q' \equiv Q \Rightarrow P \xrightarrow{\alpha} Q$	(Trans $\equiv$)

From this operational semantics, we can define the traces of a process, with reductions $P \xrightarrow{s} P'$ where s is a sequence of actions.

Traces:

$s, t ::= \alpha_1, \ldots, \alpha_n$	trace

Free names, $\mathsf{fn}(s)$, and generated names, $\mathsf{gn}(s)$, of a trace s:

$$\mathsf{fn}(\alpha_1, \ldots, \alpha_n) \triangleq \mathsf{fn}(\alpha_1) \cup \cdots \cup \mathsf{fn}(\alpha_n)$$
$$\mathsf{gn}(\alpha_1, \ldots, \alpha_n) \triangleq \mathsf{gn}(\alpha_1) \cup \cdots \cup \mathsf{gn}(\alpha_n)$$

Traced transitions: $P \xrightarrow{s} P'$

$P \equiv P' \Rightarrow P \xrightarrow{\varepsilon} P'$	(Trace $\equiv$)
$P \xrightarrow{\alpha} P'', P'' \xrightarrow{s} P' \Rightarrow P \xrightarrow{\alpha,s} P'$	(Trace Action) (where $\mathsf{fn}(\alpha) \cap \mathsf{gn}(s) = \varnothing$)

We require a side-condition on (Trace Action) to ensure that generated names are unique, otherwise we could observe traces such as

$$(\mathsf{new}(x); \mathsf{new}(y); \mathsf{stop}) \xrightarrow{\mathsf{gen}\ \langle x\rangle, \mathsf{gen}\ \langle x\rangle} (\mathsf{stop})$$

Having formally defined the trace semantics of our π-calculus, we can define when a trace is a correspondence: this is when every $\mathsf{end}\ L$ has a distinct, matching $\mathsf{begin}\ L$. For example:

$$\begin{array}{rl} \mathsf{begin}\ L, \mathsf{end}\ L & \text{is a correspondence} \\ \mathsf{begin}\ L, \mathsf{end}\ L, \mathsf{end}\ L & \text{is not a correspondence} \\ \mathsf{begin}\ L, \mathsf{begin}\ L, \mathsf{end}\ L, \mathsf{end}\ L & \text{is a correspondence} \end{array}$$

We formalize this by counting the number of $\mathsf{begin}\ L$ and $\mathsf{end}\ L$ actions there are in a trace.

Beginnings, $\mathsf{begins}\,(\alpha)$, and endings, $\mathsf{ends}\,(\alpha)$, of an action α:

$$\begin{array}{rlrl} \mathsf{begins}\,(\mathsf{begin}\ L) \triangleq [L] & & \mathsf{ends}\,(\mathsf{begin}\ L) \triangleq [\,] \\ \mathsf{begins}\,(\mathsf{end}\ L) \triangleq [\,] & & \mathsf{ends}\,(\mathsf{end}\ L) \triangleq [L] \\ \mathsf{begins}\,(\mathsf{gen}\ \langle x\rangle) \triangleq [\,] & & \mathsf{ends}\,(\mathsf{gen}\ \langle x\rangle) \triangleq [\,] \\ \mathsf{begins}\,(\tau) \triangleq [\,] & & \mathsf{ends}\,(\tau) \triangleq [\,] \end{array}$$

Beginnings, $\mathsf{begins}\,(s)$, and endings, $\mathsf{ends}\,(s)$, of a trace s:

$$\begin{array}{r} \mathsf{begins}\,(\alpha_1, \ldots, \alpha_n) \triangleq \mathsf{begins}\,(\alpha_1) + \cdots + \mathsf{begins}\,(\alpha_n) \\ \mathsf{ends}\,(\alpha_1, \ldots, \alpha_n) \triangleq \mathsf{ends}\,(\alpha_1) + \cdots + \mathsf{ends}\,(\alpha_n) \end{array}$$

Correspondence:

A trace s is a *correspondence* if and only if $\mathsf{ends}\,(s) \leq \mathsf{begins}\,(s)$.

A process is safe if every trace is a correspondence.

Safety:

A process P is *safe* if and only if for all traces s and processes P'
if $P \xrightarrow{s} P'$ then s is a correspondence.

A subtlety of this definition of safety is that although we want each end-event of a safe process to be preceded by a distinct, matching begin-event, a trace st may be a correspondence by virtue of a later begin-event in t matching an earlier end-event in s. For example, a trace like $\mathsf{end}\ L, \mathsf{begin}\ L$ is a correspondence.

To see why our definition implies that a matching begin-event must precede each end-event in each trace of a safe process, suppose a safe process has a trace $s, \mathsf{end}\ L, t$. By definition of traces, the process also has the shorter trace

$s, \mathsf{end}\, L$, which must be a correspondence, since it is a trace of a safe process. Therefore, the end-event $\mathsf{end}\, L$ is preceded by a matching begin-event in s.

We can now state the formal result of the paper, Theorem 5.2, that every effect-free process is safe. This gives us a compositional technique for verifying the safety of communications protocols. It follows from a subject reduction result, Theorem 5.1. The most difficult parts of the formal development to check in detail are the parts associated with the (Proc Cond) rule, because of its use of a substitution applied to an environment.

Theorem 5.1 (Subject Reduction) *Suppose $E \vdash P : e$.*

(1) *If $P \xrightarrow{\tau} P'$ then $E \vdash P' : e$.*

(2) *If $P \xrightarrow{\mathsf{begin}\, L} P'$ then $E \vdash P' : e + [L]$.*

(3) *If $P \xrightarrow{\mathsf{end}\, L} P'$ then $E \vdash P' : e - [L]$, and $L \in e$.*

(4) *If $P \xrightarrow{\mathsf{gen}\, \langle x \rangle} P'$ and $x \notin \mathsf{dom}(E)$ then $E, x{:}T \vdash P' : e$ for some type T.*

Theorem 5.2 (Safety) *If $E \vdash P : []$ then P is safe.*

6 Related Work

Correspondence assertions are not new; we have already discussed prior work on correspondence assertions for cryptographic protocols [WL93,MCJ97]. A contribution of our work is the idea of directly expressing correspondence assertions by adding annotations to a general concurrent language, in our case the π-calculus.

Gifford and Lucassen introduced type and effect systems [GL86,Luc87] to manage side-effects in functional programming. There is a substantial literature; recent applications include memory management for high-level [TT97] and low-level [CWM99] languages, race-condition avoidance [FA99], and access control [SS00].

Early type systems for the π-calculus [Mil99,PS96] focus on regulating the data sent on channels. Subsequent type systems also regulate process behaviour; for example, session types [THK94,HVK98] regulate pairwise interactions and linear types [Kob98] help avoid deadlocks. A recent paper [DG00] explicitly proposes a type and effect system for the π-calculus, and the idea of latent effects on channel types. This idea can also be represented in a recent general framework for concurrent type systems [IK01]. Still, the types of our system are dependent in the sense that they may include the names of channels; to the best of our knowledge, this is the first dependent type system for the π-calculus. Another system of dependent types for a concurrent language is Flanagan and Abadi's system [FA99] for avoiding race conditions in the concurrent object calculus of Gordon and Hankin [GH98].

The rule (Proc Cond) for typing name equality $\mathsf{if}\ x = y\ \mathsf{then}\ P\ \mathsf{else}\ Q$ checks P under the assumption that the names x and y are the same; we formalize this by substituting y for x in the type environment and the process P. Given that names are the only kind of value, this technique is simpler than the standard technique from dependent type theory [NPS90,Bar92] of defining typing judgments with respect to an equivalence relation on values. Honda, Vasconcelos, and Yoshida [HVY00] also use the technique of applying substitutions to environments while type-checking.

7 Conclusions

The long term objective of this work is to check secrecy and authenticity properties of security protocols by typing. This paper introduces several key ideas in the minimal yet general setting of the π-calculus: the idea of expressing correspondences by begin- and end-annotations, the idea of a dependent type and effect system for proving correspondences, and the idea of using latent effects to type correspondences begun by one process and ended by another. Several examples demonstrate the promise of this system. Unlike a previous approach based on model-checking, type-checking correspondence assertions is not limited to finite-state systems.

A companion paper [GJ01] begins the work of applying these ideas to cryptographic protocols as formalized in Abadi and Gordon's spi-calculus [AG99], and has already proved useful in identifying known issues in published protocols. Our first type system for spi is specific to cryptographic protocols based on symmetric key cryptography. Instead of attaching latent effects to channel types, as in this paper, we attach them to a new type for nonces, to formalize a specific idiom for preventing replay attacks. Another avenue for future work is type inference algorithms.

The type system of the present paper has independent interest. It introduces the ideas in a more general setting than the spi-calculus, and shows in principle that correspondence assertions can be type-checked in any of the many programming languages that may be reduced to the π-calculus.

Acknowledgements We had useful discussions with Andrew Kennedy and Naoki Kobayashi. Tony Hoare commented on a draft of this paper. Alan Jeffrey was supported in part by Microsoft Research during some of the time we worked on this paper.

References

[AG99] M. Abadi and A.D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148:1–70, 1999.

[Bar92] H. Barendregt. Lambda calculi with types. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume II.* Oxford University Press, 1992.

[BB92] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, April 1992.

[CM00] E. Clarke and W. Marrero. Using formal methods for analyzing security. Available at *http://www.cs.cmu.edu/~marrero/abstract.html*, 2000.

[CWM99] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *26th ACM Symposium on Principles of Programming Languages*, pages 262–275, 1999.

[DG00] S. Dal Zilio and A.D. Gordon. Region analysis and a π-calculus with groups. In *Mathematical Foundations of Computer Science 2000 (MFCS2000)*, volume 1893 of *Lectures Notes in Computer Science*, pages 1–21. Springer, 2000.

[FA99] C. Flanagan and M. Abadi. Object types against races. In J.C.M. Baeten and S. Mauw, editors, *CONCUR'99: Concurrency Theory*, volume 1664 of *Lectures Notes in Computer Science*, pages 288–303. Springer, 1999.

[GH98] A.D. Gordon and P.D. Hankin. A concurrent object calculus: Reduction and typing. In *Proceedings HLCL'98*, ENTCS. Elsevier, 1998.

[GJ01] A.D. Gordon and A. Jeffrey. Authenticity by typing for security protocols. In *14th IEEE Computer Security Foundations Workshop.* IEEE Computer Society Press, 2001. To appear.

[GL86] D.K. Gifford and J.M. Lucassen. Integrating functional and imperative programming. In *ACM Conference on Lisp and Functional Programming*, pages 28–38, 1986.

[HVK98] K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *European Symposium on Programming*, volume 1381 of *Lectures Notes in Computer Science*, pages 122–128. Springer, 1998.

[HVY00] K. Honda, V. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In *European Symposium on Programming*, Lectures Notes in Computer Science. Springer, 2000.

[IK01] A. Igarashi and N. Kobayashi. A generic type system for the pi calculus. In *28th ACM Symposium on Principles of Programming Languages*, pages 128–141, 2001.

[Kob98] N. Kobayashi. A partially deadlock-free typed process calculus. *ACM Transactions on Programming Languages and Systems*, 20:436–482, 1998.

[Luc87] J.M. Lucassen. *Types and effects, towards the integration of functional and imperative programming.* PhD thesis, MIT, 1987. Available as Technical Report MIT/LCS/TR–408, MIT Laboratory for Computer Science.

[MCJ97] W. Marrero, E.M. Clarke, and S. Jha. Model checking for security protocols. In *DIMACS Workshop on Design and Formal Verification of Security Protocols*, 1997. Preliminary version appears as Technical Report TR–CMU–CS–97–139, Carnegie Mellon University, May 1997.

[Mil99] R. Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999.

[NPS90] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory: An Introduction.* Oxford University Press, 1990.

[PS96] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996.

[SS00] C. Skalka and S. Smith. Static enforcement of security with types. In P. Wadler, editor, *2000 ACM International Conference on Functional Programming*, pages 34–45, 2000.

[THK94] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *Proceedings 6th European Conference on Parallel Languages and Architecture*, volume 817 of *Lectures Notes in Computer Science*, pages 398–413. Springer, 1994.

[TT97] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[WL93] T.Y.C. Woo and S.S. Lam. A semantic model for authentication protocols. In *IEEE Symposium on Security and Privacy*, pages 178–194, 1993.